**University of London**
**Computing and Information Systems/Creative Computing**
**CO1109 Introduction to Java and object-oriented programming**
**Coursework assignment 2 2018–19**

## Contents

Introduction ........ 1
Electronic files you should have: ........ 1
What you should hand in: very important ........ 1
Notes on the `ArrayList` class ........ 3
Notes on the `List` interface ........ 3
Notes on classes given to you for the assignment ........ 4
The Revisionator ........ 5
Coursework assignment 2 - questions ........ 5
Reading (all from volume two of the subject guide) ........ 6
Deliverables ........ 6
Appendix A: Sample output from *RevisionatorUserInterface* ........ 7
Appendix B: advice about readable code ........ 9
Appendix C: Example of renaming for readability ........ 11
Marks for CO1109 coursework assignment 2 ........ 13

## Introduction

This is coursework assignment 2 (of two coursework assignments total) for 2018–19. The assignment asks that you demonstrate an understanding of static methods, defining classes, constructors, instance methods and exceptions. You are also asked to apply rules for readability, as appropriate, when writing classes and methods. In addition this assignment introduces the `ArrayList` class and the `List` interface.

## Electronic files you should have:

*Java files*

- *Category.java*
- *QuestionParser.java*
- *Revisionator.java*
- *RevisionatorUserInterface.java*
- *TimeFormatter.java*
- *TimeKeeper.java*

*Quiz files*

- *boolean.quiz*
- *CompilationErrors.quiz*
- *ConstructorsAndInheritance.quiz*
- *Exceptions.quiz*
- *Loops.quiz*
- *Variables.quiz*

## What you should hand in: very important

There are two marks allocated for handing in uncompressed files – that is, students who hand in zipped or .tar files or any other form of compressed files can only score 98/100 marks.

There are another two marks allocated for handing in just the .java files asked for without putting them in a directory.

There are two marks for (1) not changing the names of the classes given to you, and (2) for naming any classes that you have to write as asked (in this assignment *Question*) ***exactly*** as you have been asked to name them.

At the end of the questions you will find a list of files to be handed in – **please note the hand-in requirements supersede the generic University of London instructions**. Please make sure that you give in **electronic versions** of your .java files since you cannot gain any marks without handing in the .**java** files asked for. Class files are **not** needed, and any student giving in only a class file will not receive any marks for that part of the coursework assignment, **so please be careful about what you upload as you could fail if you submit incorrectly**.

**Programs that do not compile will not receive any marks.**

**The examiners will compile and run your Java programs; students who hand in files containing their Java classes that cannot be compiled (*e.g.* PDFs) will not be given any marks for that part of the assignment.**

Please put your name and student number as a comment at the top of each .java file that you hand in.

## Notes on the `ArrayList` class

https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Chapter 12 of the subject guide, volume 2, concentrates on the `Vector` class, and its similarities to `Arrays`. `Vectors` are more flexible than `Arrays`, because they can grow and shrink dynamically, as your program requires them to. While the `Vector` class is not deprecated, it has been replaced by the `ArrayList` class; an `ArrayList` can also grow and shrink dynamically. Both classes implement the `List` interface.

`Arraylists` are considered by Java to be a collection. `Collections` is a class that has useful static methods that operate on any class that Java considers to be a collection, see https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

Section 12.3 of volume 2 of the guide lists five methods of the `Vector` class, which have equivalent methods in the `ArrayList` class as follows:

| **`Vector`**<br>*method numbering from subject guide* | **`ArrayList`** |
|---|---|
| 1. `void addElement()`<br>Adds an Object to the end of a `Vector` | `boolean add(E e)`<br>Adds the specified element to the end of the `ArrayList` |
| 2. `Object elementAt(int i)`<br>Returns the *i*th element | `get(int i)`<br>Returns the *i*th element |
| 3. `int Size()`<br>Returns the number of elements | `int size()`<br>Returns the number of elements |
| 4. void `removeElementAt(int i)`<br>Removes the *i*th element | `remove (int i)`<br>Removes the *i*th element. Shifts any subsequent elements to the left (subtracts one from their indices). |
| 5. `void setElementAt(Object x, int i)`<br>Changes the *i*th element to x. | `add(int i, E element)`<br>Adds an element at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices) |

## Notes on the `List` interface

You will note that in the files you have been given, `ArrayLists` are declared to be of type `List`, as are methods that return an `ArrayList`. `List` is an interface: http://docs.oracle.com/javase/7/docs/api/java/util/List.html

*Explanation below, modified from Effective Java, 2nd edition by Joshua Bloch*
You should use interfaces rather than classes as parameter types. More generally, you should favour the use of interfaces rather than classes to refer to objects. If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.

Get in the habit of typing this:

```
// Good - uses interface as type
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

rather than this:

```
// Bad - uses class as type!
ArrayList<Subscriber> subscribers = new ArrayList<Subscriber>();
```

If you get into the habit of using interfaces as types, your program will be much more flexible. If you decide that you want to switch implementations, all you have to do is change the class name in the constructor.

For example, the first declaration could be changed to read:
`List<Subscriber> subscribers = new LinkedList<Subscriber>();`
and all of the surrounding code would continue to work. The surrounding code was unaware of the old implementation type, so it would be oblivious to the change.

**Notes on classes given to you for the assignment**

In the *QuestionParser* class we use a `StringBuilder` in the *parseLines(List<String>)* method. We could have used a `String` instead, however we do not because `Strings` are immutable, which means that when we reassign a `String`, its original value still exists, it just isn't pointed at any longer. For example:

```
String s = "hello";
s = "goodbye"; //"hello" orphaned as nothing points to it
```

Orphaned `Strings` will be cleared away by the JVM eventually, but probably not as fast as they are made, particularly if they are being made in a loop. Hence for manipulating `Strings` in a loop `StringBuffer` or `StringBuilder` is preferred, as these classes are mutable, *i.e.* what they are pointing at can be changed, hence they use less memory.

Also, in the *QuestionParser* class we use *String.split(),* a method that splits a `String` and places the new `Strings` in an `Array`. `Strings` can be split around a space or a comma say. For example:

```
String[] splitString = "the quick brown fox jumped over the
lazy dog".split(" ");//one space between quote marks
```

will return a `String Array` containing 9 items. It is also possible to specify the number of items required, see https://docs.oracle.com/javase/7/docs/api/java/lang/String.html for more information.

Note that the *Category* and *TimeKeeper* classes have private variables and public methods. This is good practice for objects, variables are private to prevent unexpected updates. Access to variables is only through the classes' public methods. The *Category* class has a *getName()* method, such methods are very common, they return the state of the named variable and are known as getters. By convention they are named starting with 'get' and ending with the variable name, but starting with a capital letter. Hence the method that returns the value pointed at by the `String` *name* variable in the *Category* class is called *getName*.

The *TimeFormatter* class only has one static method. Such classes are common, and are usually referred to as static utility classes. That is, they provide helpful methods that other classes may wish to use. In the case of the *TimeFormatter* class the helpful method is one that takes a `long` value in milliseconds, and returns a `String` giving that value in minutes and seconds. Java may measure time in milliseconds, but minutes and seconds are much easier for a human to understand.

Since the methods in static utility classes are static, there is no need to make an object of the class in order to use the method. For example, in the *isCorrectAnswer(Question, String)* method in the *RevisionatorUserInterface* class, you will find this statement:

```
String timeTaken = TimeFormatter.millisecondsToMinutesAndSeconds(
stopwatch.getElapsedMilliseconds());
```

The statement is calling the *TimeFormatter*'s single method simply by placing the name of the class before the method name, followed by a dot. The `long` value is given to the method by the calling of the *getElapsedMilliseconds()* method of the *TimeKeeper* class, using dot notation and the stopwatch variable, which is an object of type *TimeKeeper*. Since the *getElapsedMilliseconds()* method is an instance method, it can only be called with reference to an object of the class, using dot notation. Note also that the *TimeKeeper* class does not have a constructor, even though it only has instance variables and methods. When a class does not have a constructor, the compiler will add the default no argument constructor, so that an object of the class can be made.

### The Revisionator

The *RevisionatorUserInterface* class is intended to run quizzes for the user, based on questions that have previously appeared in CO1109 examinations. The user is given a list of categories to choose from, and once they have chosen their category or categories, they can then choose the number of questions that they want. The questions are taken from the *.quiz* files that you have been given. After the quiz has run the user will get a message giving feedback about their answers. See Appendix A for a sample run of the working *RevisionatorUserInterface* class.

## Coursework assignment 2 - questions

Please answer the following questions:

1. The *Category*, *Revisionator, QuestionParser, and RevisionatorUserInterface* classes will not compile because they cannot find the *Question* class. Write the *Question* class. **[12 marks]**

2. Once you have successfully written the *Question* class, you should find that the *Revisionator* class will not compile because of an unreported `IOException`. Handle the `IOException` with try/catch. Make sure to print a message about the exception from the catch block. **[12 marks]**

3. In the *RevisionatorUserInterface* class there are 2 methods whose body is empty or mostly empty: *getValidNumberOfQuestions(List<String>)* and *runQuiz(List<Question>)*. A comment with each method tells you what the method should do. Complete both methods.

   Note that in the *getValidNumberOfQuestions(List<String>)* method there is a return statement placed there so that the empty method will not give a compilation error. While your completed method should have a return statement, it should be different to the one currently in the method. **[24 marks]**

4. Once you have written the 2 methods in question 3, the *RevisionatorUserInterface* will not do what it should, because the body of the *quiz()* method has not been written. The *quiz()* method is the method called when the user chooses option 1 from the menu. The method gets the question categories and the number of questions wanted in the quiz from the user. After that it makes a list of questions from the chosen category or categories, with the number of questions chosen by the user. It prints a message to the user confirming their selection, and then runs the quiz and gives feedback to the user at the end.

   Write the *quiz()* method. Your *quiz()* method should be a short one, and each statement should call either a method in the *RevisionatorUserInterface* class, or a method from another class. **[24 marks]**

5. Once the *RevisionatorUserInterface* class compiles, run the class. When asked for a menu choice and you enter something that cannot be parsed to an `int`, for example 52.1, the class stops with an exception. Similarly, if asked for the number of questions and you enter, for example, “hello” the class stops with an exception. Amend the *RevisionatorUserInterface* class so that if the user enters something that cannot be parsed to an `int`, the class will enter a loop and keep asking for a valid number, until it receives one. **[12 marks]**

6. Make sure that you apply the rules for readability given in Appendix B as far as possible. **[10 marks]**

**Reading (all from volume two of the subject guide)**

- Sections 5.3 and 5.4 (simple and reference variables)
- Sections 9.3 – 9.6 inclusive, sections 9.8 and 9.9 (defining classes)
- Section 10.6 (Instance methods)
- Sections 11.3 – 11.5 inclusive (exception handling)

**Deliverables**

Electronic file of:

- *Question.java (new class)*
- *Revisionator.java (amended)*
- *RevisionatorUserInterface.java (amended)*

## Appendix A: Sample output from *RevisionatorUserInterface*

```
Choose one of the following options:
1. Take a quiz
2. Quit
Enter your choice: 1
There are questions in these categories: boolean, CompilationErrors,
ConstructorsAndInheritance, Exceptions, Loops, Variables
Enter the categories you would like the questions to be selected
from. Please put a space between each category: boolean
compilationerrors
There are 23 questions available. How many questions should the quiz
have? 3

You have selected a quiz with 3 questions from the categories
Boolean, CompilationErrors. Good luck!

Question 1:
TRUE or FALSE: The following expression type checks:
"threefifty".compareTo("350");
Your answer: true

Question 2:
What compilation error will the compiler find with this method:
static int add(int x, int y){int temp = x + y;}
A missing return statement
B variable temp should be declared static
C return type required
D None of the above
Your answer: a

Question 3:
The following statement will not compile: if 3/1.2 < 0
System.out.print("error");
A Because the boolean conditional is invalid as an int is being
divided by a double
B Because the boolean conditional is invalid as an int is being
divided by a float
C Because the boolean conditional should be in round brackets
D All of the above
Your answer: a

You scored 2/3 and took 1 minutes and 55 seconds to complete the
quiz. That's 66.67%. Very good!
Choose one of the following options:
1. Take a quiz
2. Quit
Enter your choice: 12
I don't know how to do that.
Choose one of the following options:
1. Take a quiz
2. Quit
Enter your choice: hello
Please enter a valid number: no
Please enter a valid number: oh ok then
Please enter a valid number: 2
Bye!
Press any key to continue . . .
```

## Appendix B: advice about readable code

**Readable code**
This assignment is following the advice given by Robert C Martin in his book *Clean Code: A Handbook of Agile Software Craftsmanship* (published 2008 by Prentice Hall, ISBN 978-0132350884). Mr Martin describes a system for writing readable code, there are others, but this is the one this assignment will be focussing on.

Mr Martin writes:

> One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. [...] We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

Mr Martin writes that 'making your code readable is as important as making it executable'. He believes that names of variables, methods and classes are a major part of what makes our code readable:

> The name of a variable, function or class should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

Martin dislikes comments, noting that as code is updated comments are rarely updated at the same time, so however helpful a comment at the start, once a class has been in use for a while any comments are likely to be outdated and confusing. He believes that code should be written with names that make the intent clear, such that comments are redundant.

You should note that the programmer has tried to follow the advice given by Martin in writing the *Revisionator* and the *RevisionatorUserInterface* classes. However, Martin himself notes that names can always be improved, and that we should not be afraid to keep refining our code. If you spot a method or variable name that can be improved, please feel free to do so, but please do not change class names.

See Appendix C for an example of renaming a simple class to make it more readable.

When answering questions in this assignment, please refer to the following rules from Martin:

**Formatting**

> You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the layout of your code, and then you should consistently apply those rules. […] It helps to have an automated tool that can apply those formatting rules for you.

**Methods**

- **Method should do one thing only**. If your method does more than one thing, break it into separate methods.
- **Do not repeat yourself** – if you find yourself writing the same code more than once, put it into a method.
- **Too many arguments.** "No argument is best, followed by one, two and three. More than three is very questionable and should be avoided with prejudice."

**Comments**
If you do write a comment, make sure it is grammatical, short, does not state the obvious and is really needed.

**Names**

- **Choose descriptive names** "Names in software are 90% of what makes software readable”
- **Unambiguous names** “choose names that make the workings of a function or variable unambiguous”.
- **Names should describe side effects**, *e.g.* a method `getOos()` will make an `ObjectOuputStream` if one does not already exist, so should be called `createOrReturnOos()`

**General**

- **Obscured intent –** make the code as expressive as possible such that its intention is clear from a first reading.
- **Put conditional statements into a method to make their intention and effect clear** to make their operation and intention clear, *e.g.*
  **BAD** `if (guessedWord.length()< 6)`
  **GOOD** `if(guessedWordIsTooShort(guessedWord))`

## Appendix C: Example of renaming for readability

A simple example of renaming methods and variables for greater readability.

***Original***

```
public class Calculator{

      public static void calc(int x){
            if (x >= 70){
                  System.out.println("grade = A");
                  return;
            }
            if (x >= 60){
                  System.out.println("grade = B");
                  return;
            }
            if (x >= 50){
                  System.out.println("grade = C");
                  return;
            }
            if (x >= 40){
                  System.out.println("grade = D");
                  return;
            }
            if (x<40) System.out.println("grade = F");
      }

      public static void main(String[] args) {
            calc(90);
            calc(53);
            calc(30);
      }
}
```

***Renamed***

```
public class GradeCalculator {

      public static void calculateAndPrintGrade(int finalMark){
            if (finalMark >= 70){
                  System.out.println("grade = A");
                  return;
            }
            if (finalMark >= 60){
                  System.out.println("grade = B");
                  return;
            }
            if (finalMark >= 50){
                  System.out.println("grade = C");
                  return;
            }
            if (finalMark >= 40){
                  System.out.println("grade = D");
                  return;
            }
            if (finalMark<40) System.out.println("grade = F");
      }

      public static void main(String[] args) {
            calculateAndPrintGrade(90);
            calculateAndPrintGrade(53);
            calculateAndPrintGrade(30);
      }
}
```

**Marks for CO1109 coursework assignment 2**

The marks for each section of coursework assignment 2 are clearly displayed against each question and add up to 94. There are another two marks available for giving in uncompressed .Java files, two marks for giving in files that are not contained in a directory and two marks for files with correct names. This amounts to 100 marks altogether. There were another 100 marks available from coursework assignment 1.

| | |
|---|---|
| Total marks for questions | [94 marks] |
| Mark for giving in uncompressed files | [2 marks] |
| Mark for giving in standalone files; namely, files **not** enclosed in a directory | [2 marks] |
| Mark for giving classes the correct names | [2 marks] |
| **Total marks for coursework assignment 2** | **[100 marks]** |

**[END OF COURSEWORK ASSIGNMENT 2]**