Coursework commentary 2018–2019

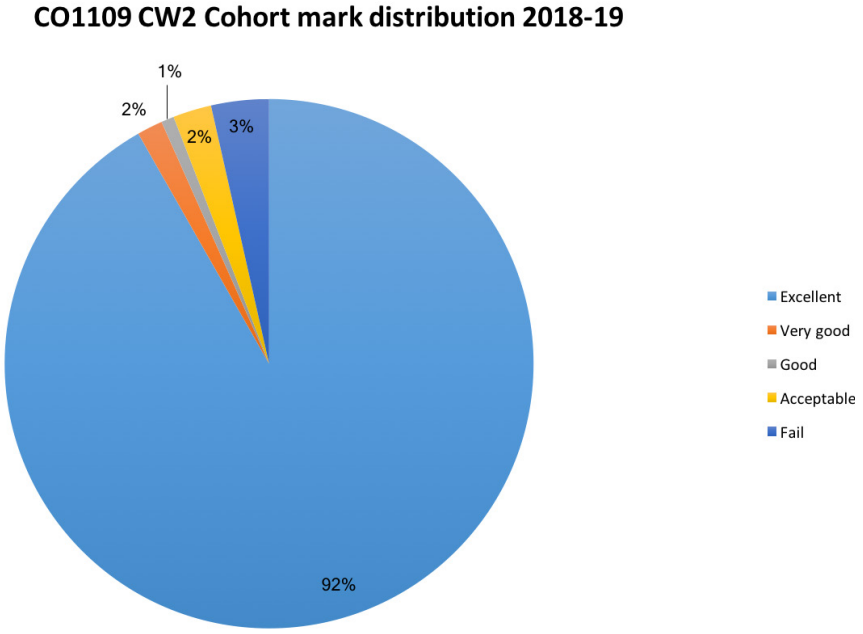
CO1109 Introduction to Java and object-oriented programming

Coursework assignment 2

General remarks

On the whole, this coursework assignment was attempted very well, with the majority of students gaining excellent marks.

See cohort mark distribution for 2018–2019 below:



Model answers

Please note that the following model answers are given with this commentary:

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

The following files are also given so that students can run and test the model answers:

Java files

- *Category.java*
- *QuestionParser.java*

- *TimeFormatter.java*
- *TimeKeeper.java*

Quiz files

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

The assignment

Students were given Java and text files that together constituted a system for running a quiz with questions based on previous CO1109 examination questions. The system would prepare the quiz by randomly choosing questions from the text files, and after running the quiz give feedback to the user. The system did not work because of a missing *Question* class and some unfinished methods; students were asked to write the missing class, and to make changes to the *Revisionator* and *RevisionatorUserInterface* classes.

Students were asked to submit the following files:

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

Comments on specific questions

Question 1

There *The Category, Revisionator, QuestionParser,* and *RevisionatorUserInterface* classes as given to students did not compile because they could not find the *Question* class. Students were asked to write the *Question* class. One way to approach this question would be to write a *Question* class that would compile, such as:

```
public class Question {}
```

Hence the compiler would be able to find the *Question* class, and its error messages would give more information about the *Question* class. For example, the following compilation error from the *RevisionatorUserInterface* class:

```
D:\109\RevisionatorUserInterface.java:116: error: cannot
find symbol
        output.println(question.getQuestion());
                               ^
  symbol:   method getQuestion()
  location: variable question of type Question
```

This means that the giving the *Question* class a method called *getQuestion()* would resolve the above compilation error.

The following error from compiling the *Revisionator* class:

```
.\QuestionParser.java:45: error: constructor Question in
class Question cannot be applied to given types;
        return new Question(question.toString(),
answerSplit[1]);
               ^
```

```
  required: no arguments
  found: String,String
  reason: actual and formal argument lists differ in
length
```

This means that the compiler expects the *Question* class to have a constructor with two `String` arguments. Adding a constructor to the *Question* class with two `String` parameters will resolve this compilation error from the *Revisionator* class. The above error states `required: no arguments` because when a class has no constructor written, the compiler adds a no-argument constructor so that an object of the class can be made. Hence the empty *Question* class will have a no-argument constructor as far as the compiler is concerned.

This was quite a challenging question and it was answered well by most students. When mistakes were seen, usually the *Question* class compiled, but it did not do everything that it needed to do, and this was often because of issues with the constructor.

One mistake made by a very small minority was to give the *Question* class static variables, which meant that every instance of the class had the same values. Effectively, every time a new instance of the class was made, the new values given to the static variables overwrote all of the values in the other *Question* objects, because only one copy of a static variable exists for every instance of the class.

Some *Question* classes had no constructor, but did have two `String` instance variables, and this meant that the instance variables would be initialised to `null`. Others correctly had a two-argument constructor, but the constructor was empty meaning that both fields of all *Question* objects would again be initialised to `null`. Other mistakes with the fields included assigning the question `String` to the `String` variable used to hold the question, but then overwriting this with the answer `String`. Hence the question field had the answer, and the answer field was null. *Question* classes whose constructor made an object with one or more null fields would cause the system to throw a **`NullPointerException`** when the quiz was run. Again, this mistake is easy to find with testing.

One other mistake seen was confusion in assigning values to the fields, such that one field received the value that the other field should have had; for example, the question `String` contained the answer and the answer `String` contained the question. This would not cause any exceptions, but would cause some confusion when running the quiz, so it was surprising that this mistake was not spotted in testing and corrected.

Question 2

Students were asked to handle the unreported `IOException` that was preventing the *Revisionator* class from compiling. The compilation error was caused by the *loadQuizQuestions()* method, which in turn was calling the *parse(String)* method from the *QuestionParser* class. The *parse(String)* method, which was reading in from text files using the `Files` class, threw rather than handled the potential `IOException`. This meant that any method invoking it had to either handle the exception, or throw it in its turn.

A good answer to this question would have given some meaningful information from the catch block about why the exception had been thrown. In the model answer, the catch block tells the user which file was being read when the error happened.

One issue with many answers, was including the statement `String quizFile = quizFiles.get(i);` inside the try block. This was not a

mistake as such, but it was unnecessary and showed a lack of understanding that it was the following statement, `questions = parser.parse(quizFile)`; that could potentially trigger the exception.

Other errors included methods that threw the exception as well as handling it; and students who did not give any feedback on the exception from the catch block, either because the catch block was empty, or because the message that was printed was not at all informative (e.g. 'an error has occurred').

Since the *loadQuizQuestions()* method was called from the constructor of the *Revisionator* class, it was possible to handle the exception in the constructor rather than in the method itself, and a minority of students took this approach successfully.

Question 3

Students were asked to complete two methods in the *RevisionatorUserInterface* class that were empty: *getValidNumberOfQuestions()* and *runQuiz()*.

The *getValidNumberOfQuestions()* method

The comment in the *RevisionatorUserInterface* class notes that the *getValidNumberOfQuestions()* method should:

- tell the user the maximum number of questions available
- ask the user how many questions they want in their quiz
- loop for valid input such that if the user entered a number that was too low or too high the user would be prompted again to enter a valid input.

The comment did not specify what 'too high' or 'too low' meant, but it should have been clear to students that the minimum number of questions was one, and the maximum was the number of potential questions from the categories the user had chosen.

Most students recognised that the first thing their method should do was to invoke the *getNumberOfQuestionsInCategories()* method, which was in the class as given to students, in order to find the total number of possible questions from the categories chosen.

Secondly, the method should tell the user the maximum number of questions possible, and ask them to enter the number of questions wanted. Then the method needed to loop until the number entered by the user was valid, and finally return a valid number.

Boolean condition errors

Most of the errors seen with this method were with the logic of the `boolean` condition guard in the loop for valid user entry. The loop in the model answer has a while loop with this `boolean` condition:

```
while (number <= 0 || number > totalNumberOfQuestions)
```

This means that the loop will only be entered if the user enters a number that is zero or less, or the user enters a number that is bigger than the possible number of questions. Once entered the loop will continue until the user enters a number that is both greater than zero, **and** less than or equal to the total number of possible questions.

Many students wrote loops that accepted 0 as a valid number of questions, or accepted negative numbers, or would accept any positive `int`, including values that were bigger than the total number of questions available. These mistakes were sometimes caused by students specifying either a minimum or a maximum number that the loop would accept, but not both. The examiners saw loops such as:

- `while (number <= 0) //method will accept any positive int`
- `while (number > totalNumberOfQuestions)`
  `//method will accept zero and negative numbers`
- `while (number < 0 || number > totalNumberOfQuestions)`
  `//method will accept zero`
- `while (number < 0 || number >= totalNumberOfQuestions)`
  `//method will accept zero and will not accept the total possible number of questions as a valid entry`

Note that the effect of accepting a number that is zero or negative would be running a quiz with zero questions; the effect of accepting a number that was too big would be that the system would enter an infinite loop when trying to compile a quiz with more questions in it than those available. This is because the *getQuizFromCategories()* method in the *Revisionator* class picks questions randomly from the categories available, and has a `for` loop that has no final expression (see section 8.4 of Volume 1 of the CO1109 subject guide for the definition of *final expression*). The `for` loop continues until the iterative variable *i* is equal to the number of questions chosen by the user. Instead of being updated in the final expression, the iterative variable *i* is updated in an `if` statement that checks to see if the question just chosen is already in the question list. If it is then *i* is not updated and the loop continues. If the method is trying to compile a quiz that has more questions than are available in the categories, eventually it is going to find that every new question it chooses is already in the list, hence *i* will not be incremented and the loop cannot end.

Another error seen more than once was writing the following `boolean` condition:

```
while (number <= 0 && number > totalNumberOfQuestions)
```

The above condition means that the `while` loop will only be entered if the user enters a number that is less than zero (that is, a negative number) and greater than the total possible number of questions. Since the total possible number of questions is likely to be a positive number greater than zero, it is impossible for this condition to be fulfilled, since it effectively means *if the number entered is both negative* ***and*** *positive*. This means that the loop will never be entered and the method will effectively accept any `int` value.

Other more idiosyncratic loop errors were seen, such that the method would accept any value, or the user only got one more attempt at entering a valid number, if their first number was invalid. Other idiosyncratic errors meant that if the loop for valid entry was entered, it was never left. With a few test runs, students could easily have found all of the various errors with loops seen by the examiners.

Messages to the user

Some students made mistakes with their messages to the user in their *getValidNumberOfQuestions()* method, such that the user would have to guess what valid input was, or that additional input from them was needed, as follows:

- Some methods only accepted numbers greater than 1 or 2, but did not

tell the user this at the start. If a student decided that 3 was the minimum number of questions for a quiz, that is not unreasonable, but users should be told this before they enter their first number, not either (1) after their first entry is rejected or (2) not at all.

- Some methods did not tell the user the total number of questions available from their chosen categories.
- Some methods looped for valid input without asking the user to enter another value.
- Some methods set the maximum number of questions as an arbitrary value (e.g. 15, 23) rather than calculating it from the total number of questions in the categories chosen by the user.

Other errors

Another reason for losing marks for the *getValidNumberOfQuestions()* method was that the method did all that it was supposed to do, but also did work that it was not supposed to do; for example, doing something that the *quiz()* or *runQuiz()* method should do.

Finally, a small minority wrote their *getValidNumberOfQuestions()* method to iterate recursively, rather than with a `while` loop. Using recursion (where a method calls itself) for iteration is a bad idea unless you are absolutely sure that the recursion will not go so deep that your system runs out of memory and the class you are running ends with a `StackOverflowError`. Even without a stack overflow, recursion can make heavy use of the memory, possibly much more so than a method written to use a `for`, `while` or `do/while` loop.

The *runQuiz()* method

The *runQuiz()* method should run the quiz and keep a record of correct answers. The method should time the quiz using the *TimeKeeper* class. Once the quiz has ended, the method should display to the user their total number of correct answers, the time they took and the percentage of correct answers with an encouraging message.

While many students wrote correct methods, a large minority struggled with this part of the question. Mistakes divided into six areas:

1. **Those who had no idea how to approach writing the method.**

   Those students who had no idea how to approach writing the method perhaps had not combined the information given in the question, with that given in the comment written above the empty method in the *RevisionatorUserInterface* class given to students. These students are encouraged to read the method in the model answer.

2. **All quiz results were given the same encouraging message.**

   Most students used the *encouragingMessage()* method to give a different feedback message depending on the quiz results. A small minority did not use the method and instead gave the same message at the end of every quiz, thus losing marks.

3. **Problems caused by using the Scanner class's nextInt() method.**

   A few students used `Scanner.nextInt()` in their *getValidNumberOfQuestions()* method to read in the required number of questions in the quiz. Because the *nextInt()* method leaves a hanging new line, this hanging new line would be read in the next time that the `Scanner` object is invoked. This meant that it was read in as the answer to the first question, preventing the user from answering that question. The solution in every case, would be to invoke the `Scanner` class's *nextLine()* method after the invocation of *nextInt()* in order to eat the new line (or rather the rest of the line after the `int`, including the new line character code).

4. **Unwanted behaviour: the runQuiz() method did things it should not be doing, or repeated work already done in other methods, or did not do things that the method should do.**

   Another reason for losing marks for the *runQuiz()* method was that the method did all that it was supposed to do, but also did work that it was not supposed to do. For example, doing something that the *quiz()* or *getValidNumberOfQuestions()* method should do, such as asking the user for their choice of categories and the number of questions that they wanted.

   Some students added extra behaviour to their methods, which usually had the effect of making the user's experience less enjoyable. For example, asking the user to confirm their already chosen number of questions. This could also lead to an error since the user could enter a higher number of questions, meaning that the method would try to access a list item that did not exist, causing an exception.

5. **Difficulties with using the TimeKeeper class.**

   Most students successfully used the *TimeKeeper* class to time the quiz. However, a minority did not understand how to use the class, so the user would receive feedback such as the quiz took 0 minutes and 0 seconds, or that they took a very long time, for example 25962339 minutes and 38 seconds.

6. **Minor idiosyncratic errors.**

   Minor idiosyncratic errors included asking the same question twice, or preventing the user from answering *true* or *false* as appropriate.

Question 4

Students were asked to write the *quiz()* method, the method called when the user chooses Option 1 from the menu. The method should determine the question categories and the number of questions wanted in the quiz from the user, then make a list of questions from the chosen category or categories, with the number of questions chosen by the user. Finally, it should print a message to the user confirming their selection, run the quiz and give feedback to the user at the end. Since all of this work was done in other methods in the class, most students understood that the *quiz()* method only needed to call other methods in the class.

Mistakes in answering this question came either from students who did not understand how to approach writing the method, or from students whose methods included unwanted behaviour, or methods that did not include wanted behaviour. For example, in one case the *quiz()* method asked the user to confirm the number of questions chosen, with the wrong answer cancelling the quiz. This behaviour did not improve the user experience. Other methods did not include behaviour that the method should have, such as running the quiz by invoking *runQuiz()* or making a list of questions for the quiz.

Question 5

This was the worst attempted question from both coursework assignments. Students were asked to amend the *RevisionatorUserInterface* class so that if the user entered something that could **not** be parsed to an `int`, the class would enter a loop and keep asking for a valid number, until it received one. With the class as given to students, entering a menu choice that could not be parsed to an int meant that the class would stop with a `NumberFormatException`. Similarly, the response to the request for the number of questions in the *getValidNumberOfQuestions()* method should also be parsed to an int, and the potential `NumberFormatException` handled with try/catch.

In the *RevisionatorUserInterface* class given to students, the *run()* method uses *getNumberFromUser()* to read in menu choices. In the model answer the *getValidNumberOfQuestions()* method, as well as the *run()* method, use *getNumberFromUser()* to read in numbers. Since the method *getNumberFromUser()* is always used to read in an `int`, then it is only necessary to handle the possible `NumberFormatException` exception in this method. This gives an answer that is simple and readable as follows:

```
private int getNumberFromUser() {
   while (true) {
        try {
                String text = getTextFromUser();
                return Integer.parseInt(text);
        }
        catch (NumberFormatException e) {
   output.print("Please enter a valid number: ");
     }
   }
}
```

Note that `while(true)` gives an infinite loop, however, once the user's entry can be parsed to an `int`, the method will return the `int`, ending the loop. If the entry cannot be parsed to an `int` then the exception is triggered, control passes to the catch block and the loop continues.

Overcomplicated answers

Many students validated int input both in the *run()* method, and again in the *quiz()* method, rather than it being done just once in the *getNumberFromUser()* method. In some cases, this worked, but was overcomplicated and hard to read.

Some students added validation to their *getValidNumberOfQuestions()* methods in such a complicated way that if the user entered something invalid then the program returned to the main menu, and the user was once more asked for the categories that they wanted, and the number of questions. This was not a good user experience, as well as not being what the assignment asked for.

Some wrote overcomplicated methods that did not handle the exception, but instead tried to avoid parsing anything that was not an `int`. In some cases, this meant that if the user pressed return without entering any text, the method would attempt to parse the entry (pressing return on an empty line) to an `int`, causing an exception. This was because the method only identified non-empty `Strings` as text that could not be parsed to an `int`. Others enforced that the *getNumberFromUser()* method would only parse input equal to "1" or "2", which worked for menu choice, although it clearly does not generalise if the menu choice is expanded, but meant that users could only choose a quiz with one or two questions.

Scanner.nextInt()

Some students used the `Scanner` object to read in an int with *nextInt()*, and handled the possible `InputMismatchException` in a loop that asked for valid input. These students received full credit. However, the majority taking this approach did not handle the possible `InputMismatchException` hence invalid input still ended the class.

Recursion

A minority looped *their getNumberFromUser()* with recursion; again, not a good idea even though it worked (see earlier comments on the *getValidNumberOfQuestions()* method).

Partially correct validation

Some students correctly handled the `NumberFormatException` in their *getValidNumberOfQuestions()* but did not handle the exception in their *getNumberFromUser()*, hence the wrong menu choice would still end the program.

Others handled the `NumberFormatException` appropriately in *getNumberFromUser()*, but separately attempted to parse user entry to an int in their *getValidNumberOfQuestions()* methods with no try/catch blocks. This meant that the class could still stop with a `NumberFormatException`. This was a very common error, and the examiners found it odd that these students did not simply call the *getNumberFromUser()* method in their *getValidNumberOfQuestions()* methods.

Attempted validation with no loop

Some students added exception handling with try/catch to their *getNumberFromUser()* method, but did not implement a loop. Some had their method return 0 if the entry could not be parsed to an `int`, which caused problems since 0 was not a valid menu choice, and was also not a valid number of questions. In other methods, the catch block would ask for a new number, but then the entry would be parsed and the exception thrown. Other students implemented ending the program with `System.exit(0);` if the entry could not be parsed to an `int`.

Question 6

Applying the rules for readability given in Appendix B was done well by most students, with two exceptions. Firstly, there were students whose methods did not do what their name suggested, either because some of their behaviour was delegated to other methods, and/or because the methods carried out behaviour that was not asked for or that should have been in another method. Secondly, where the over-complicated answers to Question 5 were hard to read and understand.

Conclusion

In coursework assignment 2, those students whose methods did not do what they were supposed to, but instead delegated some of the behaviour asked for to other methods should note that this meant losing marks, even when the system worked properly overall, which it often did not.

While both coursework assignments were answered well overall, the number of small errors that could easily have been identified by testing surprised the examiners. As a student, you need to understand that when your work compiles, it does not mean that it works exactly as you expect it to. Take note: even the most experienced developer can be tripped up by errors in their logic.