Coursework commentary 2018–2019

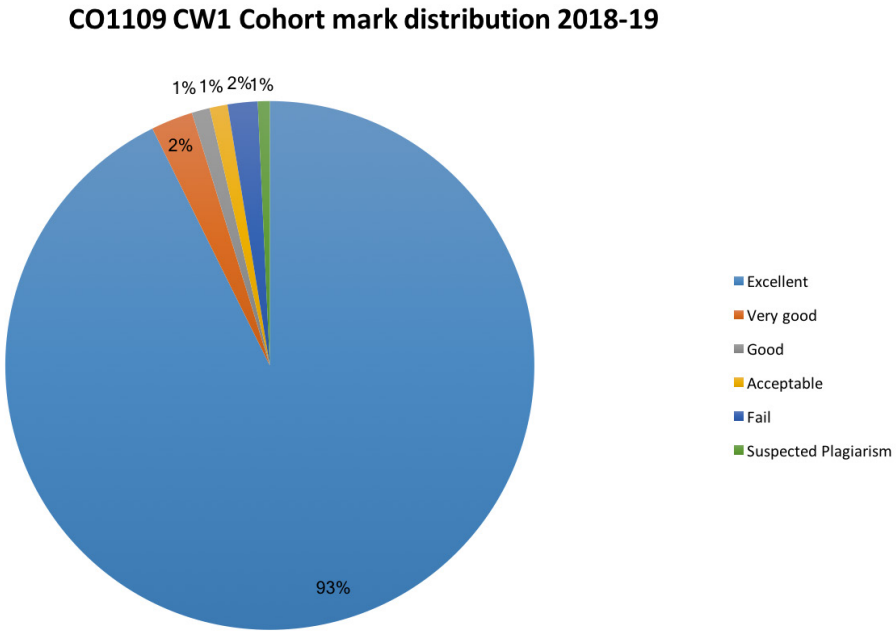
CO1109 Introduction to Java and object-oriented programming

Coursework assignment 1

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:



Comments on specific questions

Model answers

Please note that the following Java files are provided with this commentary:

- *GamesArcade.java (model answer)*
- *NumberGuessingGame.java (needed for the GamesArcade class to work)*
- *WordScramble.java (model answer)*

The assignment

Students were given two files: *GamesArcade.java* and *WordScramble.java*. The *WordScramble* class could be run through the *GamesArcade*, which displayed

the following menu when compiled and run:

```
GAMES ARCADE
Choose one of the following options:
1. Number Guessing Game
2. Word Scramble
3. Quit
Enter your choice:
```

Choosing Option 1 would result in the message 'I don't know how to do that'. Choosing Option 2 would run the *WordScramble* class.

Question 1

Question 1 asked students to play the Word Scramble game and write a comment at the top of the file, describing any improvements they would make to the game play or the code.

There were many suggestions for improving the game play. Good answers included:

- Give the player an introduction to the game to explain the rules before the first game in the session.
- Prevent the user from being given a repeated word in the same session.
- Increase the number of words to choose from, perhaps with a dictionary file.
- Make sure that an invalid input is rejected and does not count as a guess (e.g. when the user accidentally enters non-alphabetic characters, or enters a guess that is too short or long to be the actual word).
- Strip out any whitespace entered by the player before evaluating the guess so that if a player accidentally enters spaces their guess is not automatically rejected because of the spaces.
- Give the player more guesses as two is not enough.
- State the number of guesses the player has, and count them down each time the user guesses.
- Implement playing the game through a GUI.
- Allow the player the option of giving up instead of playing on until their guesses are used up.
- If the player finds the word difficult, allow them the option of either re-scrambling the word, or having a new word.
- Tell the player if they have entered a duplicate guess, and do not count that as an attempt.
- Show players their previous guesses.

The above answers all focused on the game as it was, and tweaked the game play to be more enjoyable. Other answers focused on expanding the minimal game play to make the playing experience more interesting and exciting. Such suggestions included:

- Have a countdown timer, such that the player loses if they take too long to guess.
- Have difficulty levels based on the length of the word.
- Have categories of words for the player to choose from (sports, medical, nature, etc.).
- Assign words to categories and give the category of the word as a hint.
- Have hints that are based on synonyms of the word.

- Let the player choose whether or not to have hints in order to adjust the difficulty level.
- Let the player earn points.
- Add a timer so that the more time the player takes to guess the word, the lower their score.
- Factor into the scoring the number of guesses made, time taken, difficulty level, whether hints were given and how many.
- Let the player enter their name in order to keep a game board of high scores.
- Keep a permanent record of player names and high scores so that players can judge themselves against their friends.
- Let the player use their points for such things as buying an extra go when they have run out of guesses.

Students also made suggestions for improving the code, for example by using class variables in order to decrease parameter passing and renaming some methods and variables for readability. Other students made good suggestions that would involve using classes and methods not covered by the CO1109 curriculum, such as using the `shuffle(List)` method from the `Collections` class to randomise the order of `chars` in the String after first placing the chars into an `ArrayList<Character>`.

The best answers included at least one of the top five numbered comments above. Good answers also noted that the game should accept all anagrams of words, noting that the word *scramble*, one of the seven words that the game could randomly pick, was also an anagram of *clambers*; however, clambers was not allowed as an answer.

It was good to see some students engaging with the readability rules outlined in the coursework assignment. However, a few suggested shortening method names and including a comment to give any further explanation of the purpose of the method, which directly contradicts the advice given with the assignment. This advice was taken from Robert C Martin's book *Clean Code: A Handbook of Agile Software Craftsmanship*. (published 2008 by Prentice Hall, [ISBN 9780132350884]). Martin dislikes comments on the grounds that they

> 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.

are rarely or never updated, but the accompanying code often is, so they become out-of-date and misleading. He writes:

One understandable error made by a few students was to comment that the variable NUMBER_OF_LIVES was inconsistently named, as it did not use the accepted convention for variable naming in Java. This is to start your variable name with a lower case letter, and, if the variable name has more than one word (e.g. chocolate cake) then use lower case except for the first letter of any word after the first. Hence chocolate cake as a variable name would be *chocolateCake*.

The complete declaration of the variable is as follows:

```
private static final int NUMBER_OF_LIVES = 2;
```

The subject guide does not explain that variables declared using the key word *final* are constants, meaning that they cannot be re-assigned once initialised. By convention constant (*final*) values are written by Java developers in upper case.

Some students commented that the game should have a loop so that the player could play again, rather than the game ending after one go and the player having to run it again. In fact, this would not be an issue as the game is intended to be run through the *GamesArcade*, which offers the player the choice of playing again once the game ends.

Those students playing through the *GamesArcade* noted that the game should be able to recover from user input error. This was because when playing through the *GamesArcade*, if the player entered a menu choice that was not an `int`, then the class ended with a `NumberFormatException`. The *WordScramble* class itself only accepts `Strings` and cannot be broken by user input. It was noted in the assignment instructions that incorrect user input could cause the *GamesArcade* class to end with an exception, and that this should be ignored as the assignment was not covering validation of user input.

Question 2

Question 2 asked students to write a new class called NumberGuessingGame, following some very specific rules, and change the menu of the GamesArcade class such that if the user chose menu Option 1 the NumberGuessingGame would run. In their new class, students were asked to apply rules for readability that were given with the assignment. Students were asked to submit their new class – NumberGuessingGame.java, together with their revised GamesArcade.java and with WordScramble.java – after testing and adding their comments.

Students were asked to implement a guessing game, where the computer picked a number at random from a range of numbers, and the player had to guess the number. The range of numbers should have been at least 0-5,000, but students could choose a bigger range if they wished. Obviously, the player could guess the number fairly quickly if they remembered the binary search algorithm, but this would require the user to be told what the range of numbers were, since applying the algorithm would mean first guessing the middle number in the range, and proceeding from there depending on whether the game fed back that the guess was too high or too low.

a. Students were asked to make sure that their class had the following four methods:

```
askUserToGuessRandomNumber();
readAndReturnUserGuessAtRandomNumber();
getAndReturnFeedbackMessageForUser();
showFeedbackMessageToUser();
```

Any parameters in the methods listed above were omitted, so it was up to the student to decide what parameters each method should have, if any.

`Scanner` errors

The most common error seen in answers to this question was caused by an issue with the `Scanner` class's *nextInt()* method. Possibly students with this error had only tested their class by running it directly, rather than running it through the *GamesArcade* and hence had missed that running the class through the *GamesArcade* meant that the game ended with an exception after the player's first go. This was usually because the method *readAndReturnUserGuessAtRandomNumber()* was using the `Scanner` class's *nextInt()* to read the user's guess, and this leaves a hanging new line, or rather it leaves everything after the int it has read to the end of the line, and this is usually just the newline character. At the end of the game, the *GamesArcade* reads the new line (taking it as user entry for a menu item) and tries to parse it to an `int`, causing the exception. Running the game directly means that the new line character is never read in, and so the

exception cannot be thrown. The solution to this is 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).

Another error seen was closing `Scanner` object before the game had finished using it. A minority of students closed the `Scanner` object in one of the methods, either *askUserToGuessRandomNumber()* or *readAndReturnUserGuessAtRandomNumber()*. This would mean that an exception would be thrown (`IllegalStateException`) the next time the method was run. Testing should find this error even when the game is only played once, since the exception would be thrown after the user's second guess, making the game unplayable.

Methods did not do what their names suggested they should

Combining the advice given in the coursework assignment on readability, with the names of the four methods to be written, should have allowed students to understand what it was that the methods should do: they should do what their name suggests and nothing else. A minority of students wrote methods that did not perform the tasks that their names suggested they did. The errors seen were as follows:

- *readAndReturnUserGuessAtRandomNumber()* method did not return the player's guess.
- *getAndReturnFeedbackMessageForUser()* method displayed feedback to the player; it did not return it.
- the *showFeedbackMessageToUser()* should display all feedback messages to the user, but was only used to show the player the winning message, so it was not doing all that it should.

Usually, students whose methods did not do what their name suggested had their methods call other methods to complete their tasks. So, for example, if the *readAndReturnUserGuessAtRandomNumber()* method did not return the player's guess, then instead it would call the *getAndReturnFeedbackMessageForUser()* method, with the player's guess given as a parameter, in order to provide feedback on the guess. This made the logic of their class harder to read and decipher, as it made decoding the actions of the user interaction loop a lot harder. The model answer uses the four methods in a game-playing loop that is concise and readable as follows:

```
int guess;
String message;
do{
    askUserToGuessRandomNumber(output);
    noOfGuesses++;
    guess = readAndReturnUserGuessAtRandomNumber(inp
ut);
    message=getAndReturnFeedbackMessageForUser(numberToGu
ess, guess);
    showFeedbackMessageToUser(output, message);
} while (userHasNotGuessedCorrectly(numberToGuess,
guess));
```

Well done to those students who wrote similar concise and readable user interaction loops.

Methods that did more than their name suggested

It was important that the four methods did what their names suggested, and no more. The rules given for readability with the coursework assignment explained that:

> 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.

Hence methods that did more (or less) than their name suggested made the entire class harder to read. By far the most common error here was having one or other of the four methods running the game play loop, either in addition to the work they should be doing, or instead of it with that work done in one of the other methods. These students lost marks for part 2(a), but also for part 2(b), which asked for a separate method that looped until the user guessed the number.

A minority took the instruction to write the methods literally, in that they wrote all four methods, but some or all were not invoked in the class. Usually, these students did not invoke one of their four methods, instead including the work that the method should have done in one of the other methods. These students lost marks twice, once for not invoking one of their methods, and again because one of their methods was doing more work than its name suggested.

Methods not written at all

Those students who lost all of the marks for this question by not writing the four methods at all made by far the worst error. Either these students wrote the game play entirely in the main method, exactly as the coursework instructions had told them not to do; or alternatively, they wrote a method that carried out all the game play, and was called by the main method.

b. Students were asked to write a method that looped until the user guessed the number.

The majority of students wrote good, readable methods using the four methods specified by part 2(a). Some errors were seen, including game playing loops that did not end when the player guessed correctly, which meant that they did not end at all.

Note that the game was specified in Volume 1 of the CO1109 subject guide, Section 8.9.14, and students were asked to read and implement this guessing game play. Some errors were seen, indicating that a minority had not read these instructions very carefully; the most common of these was not telling the player the number of guesses they had made at the end of the game. Another less common error was ending the game-playing loop after a fixed number of guesses, instead of letting the loop continue until the player correctly guessed the number.

Other errors seen were clearly the result of poor testing, or not testing at all. A minority of students clearly need to understand that just because their code compiles, does not mean that it is correct. This would include such errors as:

- telling the user the number they need to guess with each iteration of the loop
- not making the random number successfully so that the number for the user to guess is always zero

- when the game ends, telling the player that the number of guesses they had made was equal to the last number that they guessed.

The worst mistake, which meant students lost all of the marks for this question, was writing the game-playing loop in the main method.

c. Students were told to write any other methods that they needed, remembering the advice about methods given in the rules for readability, which included that methods should do one task only.

   In particular, students were asked to make sure that they wrote a method that, when called, plays the game without the user needing to call any further methods. This method could then be invoked in order to play the game through the *GamesArcade*. Most students took their cue from the *WordScramble* class, and wrote a method called *play()* that invoked the game. Almost all students successfully answered this question, with only a very small minority making mistakes. This included not writing a *play()* method at all for reasons already covered (putting all the work of the game into a single method, or into the main method).

d. Students were asked to format their code in a readable way. Most students gained full marks for this question. A very few students lost marks for formatting that was inconsistent and did not make the structure of their class, and their methods, clear.

e. Students were asked to write a main method in the *NumberGuessingGame* class, and use it to test the class. The main method should have had one statement only in it, calling the *play()* method or equivalent method used to start the game running.

   The main method should have been used to test the class and correct any errors before re-testing. If errors could not be corrected, they should be noted with comments.

   There are three types of errors:

   - Compilation errors.
   - Run-time errors.
   - Logical errors.

   In running our classes, we are testing for any run-time or logical errors. Run-time errors in Java mean that the program stops with an exception. These are sometimes caused by issues that the programmer should have anticipated and allowed for; such as validating user input. An example of a run-time error is that the *GamesArcade* will stop with a `NumberFormatException` if the player enters a menu choice that cannot be parsed to an int (the class was written to fail in this way, since handling exceptions was deferred to coursework assignment 2). Logical errors are those made by the programmer, where the class does not do exactly what it is intended to do.

   Some students lost marks for logical errors that their testing should have uncovered. For example, the examiners' testing found classes that were in a loop such that it was impossible for the player to quit the game. Other tests found run-time errors, such as a `NumberFormatException` thrown because the student wrote a menu into their *NumberGuessingGame* class, and used the `Scanner`'class's *nextInt()* method to read in th' user's menu choice. If the player chose 1 to return to the *GamesArcade* menu, then the program ended with a *NumberFormatException* because the *nextInt()* method leaves a hanging new line, which the *GamesArcade* class will read in as user response and attempt to parse to an int (see also the comments on part 2(a) `Scanner` errors).

   Other errors seen were:

- writing an empty main (an empty main cannot be used for testing)
- writing no main method at all
- writing a main method that was doing all the work of the class (these students also lost marks for parts (a), (b) and (c)).

f. Students were asked to change the *GamesArcade* class such that if the user chose menu option 1 the *NumberGuessingGame* would be run.

Most students who attempted this question received full marks for it. Some students wrote loops into their *NumberGuessingGame* class, so that the game could be played a second and subsequent times when not run through the GamesArcade; sometimes this worked well, and sometimes it had the effect of complicating the running of the *GamesArcade* in a user-unfriendly way. That is, the loop in the *NumberGuessingGame* class had the effect of making the player do more work when trying to play follow-on games through the *GamesArcade*; or did not work properly meaning that the user could not return to the menu in the *GamesArcade* class. Other mistakes were with the `Scanner` class, as documented in comments on part 2(a).

All mistakes that were seen could easily have been uncovered by invoking the game from the menu in the *GamesArcade* class, at most twice in the same session.

g. This question gave marks for applying the readability rules given with the coursework assignment. In writing their *NumberGuessingGame* class, a sizable minority of students wrote comments for each of their methods, explaining their purpose. This was despite the readability rules given with the coursework suggesting that comments should be used sparingly, if at all, and that method names should completely describe their purpose. Other students lost marks for readability by writing methods that did not do what their names suggested they did, or did what their name suggested plus other tasks as well. Some lost marks by obscuring the logic of the game by writing a game-playing loop where a method was called that in turn called other methods, so the logic and structure of the loop could not easily be read from the loop, as it can be in the model answer.