Coursework commentaries 2016–17

CO2220 Graphical object-oriented and internet programming in Java – Coursework assignments 1 and 2

Coursework assignment 1

General remarks

Please note the following files are provided with this commentary:

Part a

RectanglesGUI.java

Part b

The following files were given out with the coursework assignment:

- *BaseQuiz.java*
- *FreeQuiz.java*
- *TrueFalseQuiz.java*
- *Question.java*
- *FreeQuizQuestion.java*
- *MultipleChoiceQuestion.java*
- *GenericQuizClasses.pdf*

The following files contain solutions to the coursework assignment:

- *MultipleChoiceQuiz.java*
- *TrueFalseQuestion.java*
- *TrueFalseQuiz.java*

Comments on specific questions

Part a: RectanglesGUI

This question was done well on the whole, with the majority gaining full credit.

Question 1: A loop to draw the rectangles

Students were given the RectanglesGUI class, which when run drew 25 rectangles on a drawPanel variable in a `JFrame`. Each rectangle was drawn with two dedicated statements, one to place the rectangle and one to set its `Color` variable. This meant that the *paintComponent (Graphics)* method had more than 50 individual statements. Students were asked to reduce the number of statements by using a loop of their choosing. The question noted that the model answer used nested *for* loops and reduced the number of statements needed to five. This was meant to be a guideline, but many students took it as an instruction to use nested for loops and have no more than five statements in total to draw the rectangles. Most students achieved this, with a minority using a different loop structure. Provided that the number of statements was reduced by successfully repeating some in a loop, full credit was given.

Question 2: distribute the rectangles evenly across the *drawPanel*

Students were asked to write statements for drawing the rectangles to: (1) completely fill the `drawPanel`; and (2) be scaleable – if the user were to change the size of the `JFrame` then the rectangles would grow or shrink yet still 25 squares would completely fill the *drawPanel*.

This was done well on the whole, with a very few distinct errors. One such was drawing too many rectangles, most of which would not be displayed on the *drawPanel*, since it did not have space to show them all. Many students wrote nested for loops, and rather than using five to limit the number of rectangles drawn by the inner and outer loops, they used the width or the height of the *drawPanel* (i.e. `getWidth()` or `getHeight()`) to limit the number of iterations of the loop, or occasionally, a far too large number, such as 100. These students did not realise their mistake, as the drawPanel could not display all the rectangles. Other mistakes seen more than once were that some of the rectangles were larger than others, arising from logic errors in the loops; and that the rectangles would scale properly one way (horizontally or vertically); but not the other (vertically or horizontally) where students used *getWidth()* to limit both the width and the height; or used *getHeight()* to limit both dimensions.

Question 3: Add a JButton to the SOUTH region using *BorderLayout*

All students attempting this question received full credit.

Question 4: The RandomColorListener class

This was well done on the whole; the only mistake, seen rarely, was not writing an inner class, and either directly implementing the `ActionListener` interface, or adding an anonymous `ActionListener` class in the *go()* method. If a student made either mistake in Question 4, they would make the same mistake in Question 6 and therefore lost credit for both.

Question 5: Using the *RandomColorListener* class to change some of the rectangles to a random color when the user clicks the button in the SOUTH region

This was done well on the whole, but was the single question with the most errors. Some students could not get the logic of the class right, such that both `Color` variables changed at once; or one set of rectangles would change randomly, while the other set would always be blue. In addition, some students had trouble with their display, and should have tried adding a `repaint()` statement to the class; namely, `drawPanel.repaint();` and similarly for the ResetListener class.

Question 6: The *ResetListener* class

This was attempted well, despite having the lowest average mark. A few students lost marks by not writing an inner class, but many more made no attempt at the question at all.

Question 7: Add the *ResetListener to a JButton* in the NORTH and make some changes to the functioning of the class

Most students gained full credit for this question, with a very small number losing marks for such things as adding the button, but not to the NORTH region as asked; and, more seriously, not being able to add the inner class *ResetListener* to the button, since the `ActionListener` interface was directly implemented.

Conclusion

Most students attempting Part (a) achieved most or all of the marks. A very small number lost all credit by handing in a *RectanglesGUI* class that had numerous errors and would not compile. It is better to hand in a class that compiles and answers some of the questions than to hand in a file that attempts all questions but does not compile. It is also possible to lose marks by writing classes that are unnecessarily complicated – developers always strive to keep their classes as simple and readable as possible. Over-complicated code can make it hard to trace logic errors, and in fact, more than one student produced a class that did not work entirely as it should, and found themselves unable to locate the error in their much too complicated code.

Finally, the examiners very much appreciated the very small minority of students who wrote meaningful comments in their submissions, such as noting faults in the RectanglesGUI class that they had not been able to correct. Please remember that such comments can provide some evidence of understanding.

Part b: Quizzes

Question 1: The *MultipleChoiceQuiz* class

Students were given the complete *FreeQuiz* class, and asked to write a *MultipleChoiceQuiz* class. Like *FreeQuiz*, *MultipleChoiceQuiz* would inherit from *BaseQuiz*, and be the same as *FreeQuiz* in many respects, except that the class would use the *MultipleChoiceQuestion* class for the questions displayed to the user in the quiz.

Mistakes were made in showing the possible answers to the user; these could not be copied from the *FreeQuiz* class, which did not display possible answers, only questions. (The most common errors are in bold.)

- The multiple choice question is presented to the user without its possible answers.
- **Each question is missing one of its possible answers.**
- **Questions are presented to the user without numbering the possible answers, and without instructions as to how to answer, so the display to the user looks like the *FreeQuiz* class.**
- The user is told the number of answers is one more than they actually are; for example, if there are six answers the user is told there are seven.
- The user is told there is one fewer answer than there actually is.

The FreeQuiz class accepted any `String` as legitimate input, while the *MultipleChoiceQuiz* class needed to accept only the numbers from 1 to the total number of possible answers, which varied with each question. Many mistakes were made with restricting and detecting valid input including the following (those in bold are the most common):

- **If a `String` that cannot be parsed to an int is entered, class stops with an exception.**
- **Accepts any `int` as legitimate input, if the number is out of range, the user is told their answer is wrong.**
- Accepts all input as legitimate; the user is told their answer is wrong both for the wrong answer and for invalid input.
- Accepts all input as legitimate and correct, so that whatever the user enters, they will always score 100%.
- For every question accepts the numbers 0-9 inclusive, since the length of the input String is checked, and any with length greater than 1 are rejected.

- The top of the range of legitimate answers is rejected as invalid; for example, if there are 6 questions, then 6 is rejected, the user is told their input is not valid.
- **All invalid input is detected but the error message, copied from the *FreeQuiz* class, says that the user's entry "cannot not be blank".**
- **If the user enters an out of range integer, they are told that their answer "cannot be blank".** A `String` that could not be parsed to an `int` received an appropriate error message (for example, 'You must enter a number between 1 and x').

Students found this question quite challenging, and it had a low average score of 4.81; the question was marked out of 7.

Question 2: The *TrueFalseQuestion* class

Students were asked to write the *TrueFalseQuestion* class. This could be done by following the model given by the *FreeQuizQuestion*, but changing the type of the answer field to `boolean`. Most students found this question straightforward, and it received a good average mark of 5.53 (the question was marked out of 6).

Only two significant mistakes were seen, both related to the constructor. The first was not initialising the `boolean` answer variable in the constructor; that is:

```
public TrueFalseQuestion(String question, boolean answer) {
      super(question);
}
```

Since the `boolean` *answer* variable is not initialised, it takes the default value (for a `boolean`, *false*), hence the answer is always false.

The second mistake (seen several times) was that the student copied the *MultipleChoiceQuestion* class, with the single significant change that `private int answer`; was changed to private `boolean answer;`. This meant that the *TrueFalseQuestion* class had a superfluous field (that is, a superfluous instance variable) `private List<String> possibleAnswers;`, and a constructor that was expecting a `String` *question*, and `int` *correctAnswer*, followed by any number of `String` possible answers. This is clearly not appropriate for the *TrueFalseQuestion* class, whose constructor needed to initialise a `String` question and a `boolean` answer, as most students understood.

Question 3: The *TrueFalseQuiz* class

Question 3 was generally answered well, with an average mark of 5.79 (as Question 1, it was marked out of 7). Several mistakes were seen; those relating to copying the form of the *FreeQuiz* class without reflecting on what was appropriate to the new *TrueFalseQuiz* class follow (with the most common in bold):

- **The user is not told how to answer the questions.**
- **Any non-empty String is accepted as legitimate input.**
- The user gets an error message if entering anything except true or false (good), but the message says that the answer "cannot be blank" (bad).

Evaluating true/false answers from the user

In order to evaluate the users' answer in the *TrueFalseQuiz* class, the *isValidAnswer(Question, String)* needed to decide if the user had entered a valid answer, either true or false (or "yes" and "no" if the student decided to also accept these as valid input).

A typical answer tested the input `String` against both *true* and *false* as literal strings, usually without trimming the input with *String.trim()* (this method removes any leading or trailing white space), and often with other logical errors, such as in the following:

```
Protected boolean isValidAnswer(Question question,String
answer) {
   boolean validAns = true;
   if (answer == "true") validAns = true;
   else if (answer == "false") validAns =true;
   return validAns;
}
```

The above method will always return true, a logical error. To correct this, the *validAns* variable should be initialised to false. In addition, the input `String` should be trimmed and made lower case before the comparisons with the literal Strings are made, otherwise answers such as "`true`" or "`True`" will register as invalid.

In the *isCorrectAnswer(Question, String)* some students chose to parse the answer to a `boolean`, using one of two methods from the `Boolean` class, either *parseBoolean(String)* or *valueOf(String)*. In both methods, anything that can be parsed to true, ignoring case, will return *true*; anything else returns *false*. However, both statements below return false because of the space after the final 'e' of "true".

```
Boolean.valueOf("true ");
Boolean.parseBoolean("true ");
```

Since it is quite possible that the user will add a space after their input, this means that to use these methods successfully in every case, the input must be trimmed. Very few students trimmed the input `String`.

Other problems with the logic of the *isCorrectAnswer(Question, String)* method:

- Correct answers are counted as incorrect and vice versa.
- Whatever the user inputs is considered to be true.
- The method returns the actual answer to the question, not the answer given by the user. The consequence is that if the actual answer is false then the user's actual answer is counted as wrong, whatever the user inputs. If the actual answer is true, the user's response, whatever it is, counts as the right answer.

Question 4: Generics

This question was in two parts; the first part was answered very badly. Students were asked to comment on the *Question*, *FreeQuizQuestion*, *BaseQuiz* and *FreeQuiz* and classes rewritten to use generics, as follows:

GenericQuestion	vs	*Question*
GenericFreeQuizQuestion	vs	*FreeQuizQuestion*
GenericBaseQuiz	vs	*BaseQuiz*
GenericFreeQuiz	vs	*FreeQuiz*

The first part of the question referred to the classes that had generic in their name as the 'Generic classes', and asked students to comment on how they were different from the classes that they were based on. Too often students gave complicated answers about generics that **did not relate at all** to the classes listed above.

Something that escaped most students was a motivation for using generics in the above classes. A good answer to the first part of the question would have said:

> Using generics lets you declare a variable that can hold different types, which can be bounded or unbounded. The GenericQuestion class declared a type T that was unbounded:
>
> ```
> public class GenericQuestion<T> {
>    private String question;
>    private T answer;
> ```
>
> thus allowing extending classes to choose any type for their answer variable. Generics were used in the *GenericQuestion* class because the answer field could be of various types in child classes, for example a `boolean` (*TrueFalseQuestio*n), `String` (*FreeQuizQuestion*) or an `int` (*MultipleChoiceQuestion*). In the original, non-generic, classes the implementation of the answer field was left to the child classes. Using generics the answer field could be included in the *GenericQuestion* class, with the type left to implementing classes. This allows the developer to set out more of the expected interface and design in the parent class.
>
> In comparison, the *GenericBaseQuiz* class declaration:
>
> ```
> public abstract class GenericBaseQuiz<T extends
> GenericQuestion<?>>
> ```
>
> meant that the type *T* in *GenericBaseQuiz* was bounded, in that it had to be either *GenericQuestion* or one of its child classes. The abstract method in the *BaseQuiz* class
>
> ```
> protected abstract boolean isCorrectAnswer(Question
> question, String answer);
> ```
>
> could be changed in the *GenericBaseQuiz* class to leave the extending classes to choose an appropriate *question* variable, as follows:
>
> ```
> protected abstract boolean isCorrectAnswer(T question,
> String answer);
> ```
>
> This eliminated the need for casting the question variable to one of its child types since the method as implemented in, for example, *GenericFreeQuiz* could now give the appropriate variable as a parameter to the method:
>
> ```
> protected boolean isCorrectAnswer(GenericFreeQuizQuestion
> question, String answer){
>    return question.getAnswer().equalsIgnoreCase(answer.
>    trim());
> }
> ```

An answer covering the above points would have gained full credit; an answer covering some of the above points, partial credit.

Much more understanding was shown in answering the second part of the question. Students were asked to explain why the *getRandomQuestions(ArrayList, int)* method was static in *BaseQuiz*, while the equivalent *getRandomQuestions(List, int)* was an instance method in the *GenericBaseQuiz* class.

An example of a good answer would have said:

> The type parameter *T* is only resolved to a concrete type when an instance of an extending class is made, meaning the parameter is defined at an object level, not a class level. *T* therefore is non-static, and cannot be referenced in a static method. In the *BaseQuiz* class the *getRandomQuestions(ArrayList, int)* method could be declared as static since its information was defined at the class, not the object, level.

Conclusion

Students found Part b quite challenging, with specific reference to motivation for generics.

Coursework assignment 2

Please note the following files are provided with this commentary:

General remarks

Part a

The following files were given out with the coursework assignment:

- *TextQuestion.java*
- *testquestions.txt*

The following file contains a solution to the coursework assignment:

- *TextQuestionUtils.java*

Part b

The following files were given out with the coursework assignment:

- *Client.java*
- *ClientGUI.java*
- *CourseworkComparator.java*
- *History.java*
- *ServerUtils.java*
- *StudentFileReader.java*
- *StudentServer.java*
- *StudentServerEngine.java*
- *TotalComparator.java*
- *Marks.txt*

The following files contain solutions to the assignment

- Student.java
- ExamComparator.java
- GradeComparator.java

Comments on specific questions

Part a

Question 1: Serializing

Students were asked to complete a method that would serialize a `List` of *TextQuestion* objects to a file, and would handle any I/O errors. This was done well on the whole with few errors seen, but among them was throwing the `IOException` instead of handling it, despite the explicit instruction; another was handling `Exception` instead of `IOException`.

The method was of type `boolean`. No explanation was given for the type, but many students understood that it would be appropriate to return *true* if the serialization succeeded, and false if it did not. This meant returning *false* from the catch block, and *true*, either at the end of the try block, or at the end of the method, after the catch block. A large minority did not understand this, and returned *true* only, clearly only adding the return statement to make their method compile, and giving no thought to what the value returned might mean.

Oddly, a minority of students had no return statement at all, which caused a compilation error. It was hard to see how these students could have tested their work.

Question 2: Deserializing

Students were asked to complete a method of type *List<TextQuestion>*, which would deserialize a `List` of *TextQuestion* objects from a file, and would handle exceptions, including the possible `ClassNotFoundException`, from casting the deserialized object. This question was answered on the whole as well as Question 1, but more individual errors were seen as follows:

Exception handling errors

- The method throws and does not handle exceptions.
- The method both throws and handles the `ClassNotFoundException`.
- The method uses an `Exception` catch block to catch all errors, despite instructions about handling the different possible exceptions.
- The catch blocks print no error messages so the user does not know which exception was triggered.
- The method always throws an `EOFException` due to a `while` loop that continues indefinitely (`while(true)`). The `EOFException` is used to end the infinite loop, then the `List` is returned. However, whether or not the `List` is successfully made, the user gets an error message from the catch block implying that something has gone wrong.

Other major errors

- The method tries to deserialize as if reading from a text file, causing compilation errors.
- The method successfully deserializes to a `List` of *TextQuestion*, but instead of returning this (which would work), the method then tries to copy the contents of the `List` to an empty arraylist using a `for` loop. As the number of iterations of the `for` loop depends on the size of the empty arraylist, the `for` loop does not execute and the method returns the empty arraylist.
- The method tries to read in using a `for` loop, which will iterate a number of times depending on the size of a local `List` initialised to *null*. Accessing the `List` to get the size causes a `NullPointerException` and the program ends.

Question 3: *String.split(String)*

Students were asked to complete a method that parsed a `String` to a *TextQuestion* object. The method should split the input around a '?', which should give an array containing two strings; the first used for the *question* field of the final *TextQuestion* object; and the second used for the *answer* field. The method should work out how to make sure that the question field of the resulting *TextQuestion* object still ended with a question mark, and should remove leading white space from the answer field using *String.trim()*. The method should return null if there was no '?' in the input `String`.

Most common errors

- `String` to be used as the answer variable for the *TextQuestion* object not trimmed.
- Method does not return null if there is no '?' in the input `String`.
- The question is stripped of its concluding '?' and this is not reinstated.
- Uses a method to remove whitespace that in general will not work the way that *String.trim()* does.

Less common errors

- In order to decide whether or not to return null, the method asks if the first array element contains a '?', but the splitting will have removed it, hence the method returns null, every time.
- The method does not try to confirm that the input `String` contains a '?'.
- Incorrect regular expression used for splitting means that there is only one element in the array, giving an `ArrayIndexOutOfBoundsException` when the method tries to access the second array element.
- Removes the '?' with splitting, but adds it back to the answer, rather than the question.
- The method checks that a '?' is in the `String`, and then makes a *TextQuestion* object by adding the input `String` as both the question and the answer.

Most of the common and less common errors listed above could easily be found with basic testing.

Question 4: Saving to a text file

Students were asked to complete a method that would save data from a `List<TextQuestion>` object into a text file. Students were told that the method should handle exceptions, use UTF-8 encoding and `BufferedWriter`, and should save to a file whose name would be given by the user.

One very common issue seen was that students who had removed the '?' from the question *field* of their *TextQuestion* objects in the method to parse took the opportunity to add it back while writing the question field of the *TextQuestion* objects to the file. By far the most common errors seen were: (1) not using UTF-8 encoding; and (2) the method always returning *true* or always returning *false* because the student had not considered what the `boolean` the method returned might be signalling. A trap for the unwary was that the *TextQuestion* class's *toString()* method only outputs the *answer*; a few students relied only on the *toString()* method to write their objects to the file, which meant that their text files only contained the *answer* and not the *question*.

Other errors:

- The method tries to serialize the `List` rather than saving it to a text file.
- The method does not handle exceptions.
- The method both throws and handles exceptions.
- The file name is used as a literal string in the method, despite being one of the parameters.
- The method opens a file for writing, but does nothing else.
- The method does not write each *TextQuestion* on its own line, making parsing TextQuestion objects from the file more challenging.

Conclusion

Many students made a very good attempt at part (a). Quite a number of students could have improved their work with better testing.

Part b

Students were given some files that together ran a system that allowed a user to query some student results for a module. The results for individual students were stored in Student objects, with all results stored in an *ArrayList<Student>* object. Queries that the user could run included displaying students sorted by their examination mark, coursework mark, total mark (that is, weighted average of coursework and examination mark); and grade.

However, displaying students sorted by examination mark and by grade had not been fully implemented, since the methods to do both things called on classes that implemented the `Comparator` interface that had not been written.

Question 1: *toString()* for the *Student* class

In Question 1 candidates were asked to write an improved *toString()* method for the *Student* class, using *String.format()*, and this was done well by all candidates. The only issue seen was a few *toString()* methods used `System.out.println()` to output the fields of the Student object, as well as returning them as a `String`. Although this affected the output of the GUI, it was not a serious problem. In general toString() methods should not use `System.out.println().`

After the coursework assignment was published a correction was issued, noting that String.format() could not format the variable-width fonts used on the `JTextArea` precisely. This was particularly the case as the *toString()* method first printed the name, then the rest of the fields, all on one line. The long name variable did not get a fixed amount of space using the usual `Formatter` arguments, leading to a somewhat ragged appearance on the GUI, for example, using `String.format("%-35s %02d %02d %02d %-2s", name, exam, cwk, total, grade);`, the display on the `JTextArea` started with:

```
Acebj Tifjli Ptnbo            -1 96 38 F
Acebmmb Obhxb Sbhbc           08 72 34 F
Acej Binbe Tbffe              34 41 37 F
```

Students were told that they could test their method using the main method of the *StudentFileReader* class. The issue with the GUI was not addressed, as the point of the question was to encourage students to learn how to format output using *String.format()*. The coursework author had overlooked that the `JTextArea` would compromise the formatting due to its use of variable-width fonts.

Despite this, some students are commended for looking for a way to make the GUI output more readable. Solutions included displaying each field clearly labelled on its own line, for example:

```
Name:      Ebs Xbtffn
Exam:      24
Cwk:       96
Total:     53
Grade:     F
```

The most commonly seen solution was to use tabs to space the output into columns as follows:

Acebj Tifjli Ptnbo	Exam: -1	Cwk: 96	Total: 38	Grade: F
Acebmmb Obhxb Sbhbc	Exam: 8	Cwk: 72	Total: 34	Grade: F
Acej Binbe Tbffe	Exam: 34	Cwk: 41	Total: 37	Grade: F

The above used:

```
String.format("%-52s\tExam: %-5d\tCwk: %-5d\tTotal: %-5d\
tGrade: %s", name, exam, cwk, total, grade);
```

Question 2: *ExamComparator*

Students were asked to write an *ExamComparator* class in order to make the *sortByExam()* method work. Almost every student attempting this question achieved full credit.

Question 3: *GradeComparator*

Students were asked to write a GradeComparator class in order to make the sortByGrade() method work. Most students attempting this class achieved full credit, but there were a number of errors seen that meant that the class did not work at all, normally because the student could not work out how to compare two strings.

Comparison errors

- The c*ompare(Student, Student)* method of the class had been written to compare student X's grade to student X's grade (clearly, you should be comparing student X's grade to student Y's).
- The *GradeComparator* does not work because the *compare(Student, Student)* method sends both *Student* objects to `String`, and then compares them. This will do a lexicographical comparison object by object. This means that effectively the ordering is done by whatever field comes first in the *Student* class's *toString()* method.
- The *compare(Student, Student)* method is trying to make an `int` by parsing from the *grade* field of each *Student* object, hence the method throws a `NumberFormatException`. Where the *compare(Student, Student)* method is trying to subtract two strings, the class will not compile.
- The class would not compile because the *compare(Student, Student)* method tries to use a method called *getNum(char)*, which the compiler could not find. If this was changed to the method *getNumericValue(char)* from the `Character` class, then the *GradeComparator* both compiled and worked as intended.

Conclusion

Most students made a very good attempt at Part b.