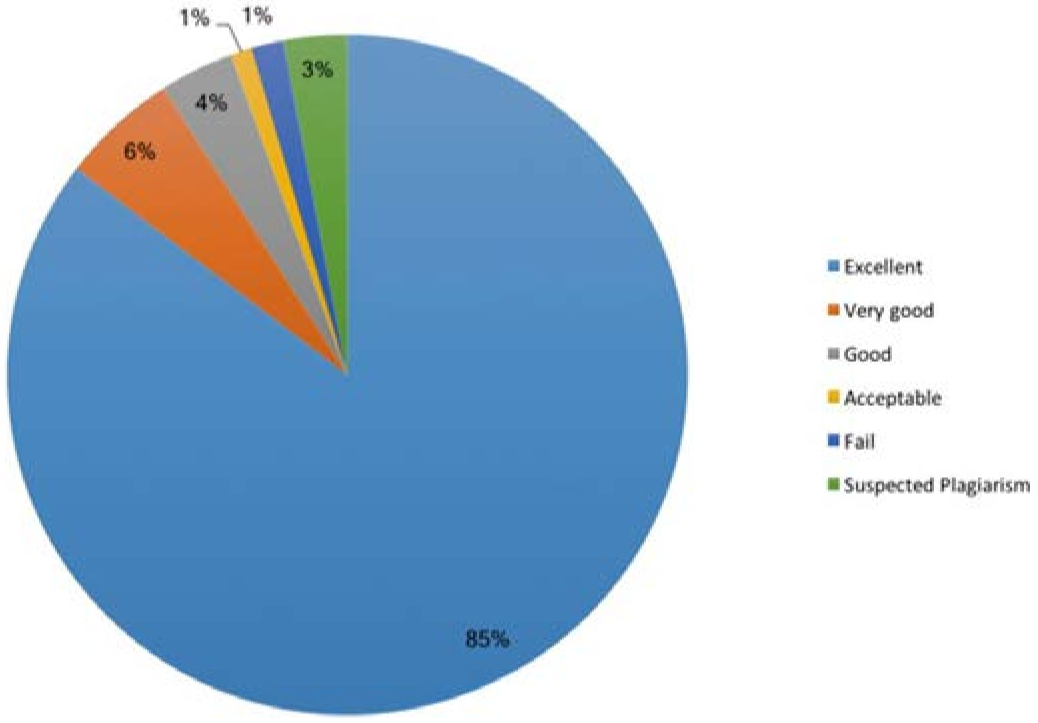
Coursework commentary 2017–2018

CO2220 Graphical object-oriented and internet programming in Java

Coursework assignment 1

General remarks

On the whole this coursework assignment was attempted well, with the majority of students gaining excellent marks, as the chart below shows:



Model answers

The following files are provided with this commentary:

Part A: CirclesGUI_Answer.java

Part B:

The following files were as given out with the assignment:

- *Dictionary.java*
- *DictionaryFileManager.java*
- *DictionaryWord.java*
- *ThesaurusFileManager.java*
- *ThesaurusWord.java*
- *Word.java*
- *dictionary.txt*
- *thesaurus.txt*

The following files contain answers to the assignment:

- *DictionaryEditor.java (with a constructor)*
- *DictionaryUI.java (with four completed methods)*
- *Thesaurus.java*
- *ThesaurusEditor.java*
- *ThesaurusUI.java*

Comments on specific questions

Part A: *CirclesGUI*

Students were given the *CirclesGUI* class, a class that used an inner class to place a `JPanel` onto a `JFrame`. A red circle was drawn on the `JPanel`, and when the user clicked on the `JPanel` the red circle would be redrawn, with coordinates taken from the point where the user had clicked. The class implemented the `MouseListener` interface, and the `MouseClicked(MouseEvent)` method was used to find the coordinates of each user click on the `JPanel`.

When the user clicked on the `JPanel`, the coordinates of the point where the user had clicked became the top left point from which the circle was redrawn. Students were asked to change the *CirclesGUI* class such that the circle was redrawn with its centre at the point where the user had just clicked. After this, students were asked to add two buttons to the `JPanel`. The first button could be clicked repeatedly to resize the circle with a new, randomly generated, diameter. The second button, if clicked, would redraw the existing circle with a randomly generated colour. Students were asked to use inner classes to implement the `ActionListener` interface in order to listen to the buttons. Students were asked to call the first inner class *DiameterListener* and the second *RandomColourListener.* Finally, the user could still click anywhere on the `JPanel` to have the circle redrawn in a new position, and the redrawn circle should have the same diameter and colour as the one then on the `JPanel`.

This question was done well on the whole, with a majority gaining full credit. Some students lost credit by using anonymous classes, rather than inner classes, to listen to the buttons. A few students lost marks for not limiting the size of the randomly generated diameter by the width of the `JPanel`, as asked. Some used the width of the `JFrame`, and others used a constant, usually 600. The use of 600 is understandable, as it is the width that the *go()* method sets the `JFrame` to. Since the width of the `JFrame` may differ from that of the `JPanel`, and since the circle is drawn on the `JPanel`, using the width of the `JFrame` was not ideal.

When asked to implement the random colour, most students took the hint given in the reading for the assignment, and made a random colour by using the `Color` constructor that took three `ints`, one each for the red, green and blue components of the `Color`. Randomness was achieved by generating three random `ints` for this constructor, limited in size to 255 as colour component values must be between 0 and 255 inclusive. A few students implemented a limited colour palette, by including a few set `Color` values in their inner classes, and choosing randomly between one of these values. In most cases where students did this there were not more than 10 set `Colors` for the class to choose between, compared to the more than 16 million colours possible by using the `Color` constructor with random values for the colour components.

Another error seen more than once was when students wrote correct code, but failed to include an instruction to redraw. For example, the diameter of the circle was successfully given a new random value in the *DiameterListener* class, but since there was no `drawPanel.repaint();` statement, the circle was not redrawn. If the user dragged a part of the `JFrame` to resize it, this would cause the `JPanel` to be redrawn and the user would then see the redrawn circle. Similar mistakes were made with the random colour. Both of these errors could have been found if the students concerned had paid more attention to testing their work.

While the majority of students answered this question well, the best answers seen included something not asked for in the question, enforcing that the diameter had a minimum value, often 10, so that the circle could always be seen.

Part B: Dictionary and thesaurus classes

Students were given some Java classes that were intended to work together to allow a user to edit, add and delete items in a dictionary.

The *DictionaryWord* class had three `String` fields: the word; a definition of the word; and a usage example for the word. The *Dictionary* class made an object consisting of an `ArrayList` of *DictionaryWords*. The *DictionaryEditor* class extended the *Dictionary* class, and added methods to edit the *Dictionary* object. The *DictionaryUI* class was a user interface, that made a *Dictionary* object, and was intended to allow the user to view and edit the *Dictionary* object. This class made use of the methods in the static utility class *DictionaryFileManager*.

A text file with four potential *DictionaryWords* was also given to students.

Question 1

Students were asked to write a constructor for the *DictionaryEditor* class; the class as given to students would not compile because it did not have a constructor. Of course, the JVM will insert the default no-argument constructor into any class without a constructor. The problem with this is that an implicit call is made to the parent class constructor by the default constructor, and if the parent class does not have a no-argument constructor, then the call will fail and the compiler will flag this as an error.

The first step to answering this question was to consider what instance variables the *DictionaryEditor* class had, that the constructor should initialise. Since the class had no instance variables, the only possible work for the constructor was calling the superclass constructor. Since the *DictionaryEditor* class extended the *Dictionary* class, the next step was to look at the *Dictionary* class, and see what constructors it had. In fact, the *Dictionary* class had only one constructor, which took a `List` of *DictionaryWords* as its only parameter. Hence a correct constructor was:

```
public DictionaryEditor(List<DictionaryWord> words){
   super(words);
      //this.words = words; //not needed, but see later
}
```

While many students answered this question correctly, an error often made by students was adding a `List<DictionaryWord>` instance variable to the *DictionaryEditor* class. These students clearly thought that as the call to the parent class constructor took a `List<DictionaryWord>` variable, the *DictionaryEditor* class would need to have a parameter of the same type. In fact, it is a misunderstanding of constructors and inheritance to think this, and could cause the *DictionaryEditor* object to be `null`.

If students gave the `List<DictionaryWord>` instance variable in the *DictionaryEditor* class the same name as the constructor's parameter (*words* in the above constructor), then it was possible that the *Dictionary* object would be made with a `null List`. This would be because the *DictionaryEditor*'s `List<DictionaryWord>` instance variable was not initialised in the constructor and so would be set to the default value for an object, `null`. With the same name as the constructor's parameter, the *words* variable would be found by the compiler and used in the parent class constructor. Hence the *DictionaryEditor* object would also be `null`. Some students who had made this mistake, avoided problems by adding `this.words = words;` as the second statement in their constructor. This would mean that the call to the super class constructor would make an object with a `null List`, but after this the `null List` would be overwritten with the parameter `List`.

Question 2

Students were told that of the six methods called by the menu in the *DictionaryUI* class, four had not been completed: *addWord(); editWord(); removeWord()* and *quit(),* Students were asked to complete the methods, reading the comments written above each method to tell them what the method should do.

There were two common errors seen with answers to these question. Firstly, in dealing with 'yes/no' input from the user, and secondly, in making each task that each method had to carry out independent of any other task the method was required to do.

Yes/no user input: Both the *editWord()* and *quit()* methods needed to ask questions of the user that had 'yes' or 'no' answers. Some students included 'yes/no' questions in the other two methods, but this really was not necessary. There were two mistakes. One was in repeating code to evaluate the user's answer in every place that the user was asked for yes/no input. This breaks the software developers' maxim DRY: Don't Repeat Yourself. The best answers wrote a method to evaluate the user's yes/no input. The model answer has the following method:

```
private boolean userSaysYes(){
   String answer = getUserInput();
   if (answer.trim().toLowerCase().startsWith("y")) return true;
   return false;
}
```

The name of the above method is designed to make the program more readable, as user input could be solicited with a question, followed by an `if` statement to carry out any actions should the user respond affirmatively to the question:

```
   //ask user a question
   if(userSaysYes())//do some stuff
```

The second mistake seen with yes/no user input was in rejecting user input that ought to be accepted. Often only lower case input was accepted; hence if the user entered 'Y' this would either be rejected, or treated as a negative answer (because if the answer was not 'y' then the code assumed it was negative). Many students wrote code that would only accept a `char`, or only accept a `String` with a single character, when it would make sense to accept the `Strings` 'yes' and 'no' since the character that the answer started with could easily be found and evaluated, as it is in the *userSaysYes()* method above, using the `String.startsWith()` method.

The editWord() and quit() methods do not treat the tasks they are asked to perform as independent of each other: In the *editWord()* method, the user needed to be asked for the word to edit, and then asked if they wanted to edit the definition, and following this asked if they wished to edit the usage example. The method should not link these tasks; the user should be able to edit one or the other, or both, or even neither. Some students enforced that both fields had to be edited, while others enforced that if the user said no to editing the definition, they were not allowed to edit the usage example. In others users could edit the definition or the usage example, but not both. These mistakes were often made because the student seemed to think that an `if` statement had to be paired with an `else` statement, which is not the case. For example, the user might be asked if they wanted to update the definition, and if they responded positively then the `if` statement would update the definition, and if they said no, the `else` statement would update the usage example, meaning that unless the user said no to editing the definition, they would not be able to edit the usage example.

In some cases students correctly implemented their methods by using an `if/else` statement with an empty `else` part. The *editWord()* method in the model answer uses two `if` statements, one for the definition and one for the usage example. This means that users can say yes to editing both fields, or can edit one but not the other, or can even say no to editing both.

Similar issues were seen in the *quit()* method, which was supposed to allow the user to save (or not save) changes to the dictionary in the text file, and after this to print a goodbye message, close the `Scanner`, and quit the user interaction loop. In some methods the user could save changes to the dictionary, or quit the program, but not both. In other cases unless the user chose to save the dictionary, the method would not quit the program. Again, this was often because of yoking the two actions together in an `if/else` statement, when they should have been independent of each other.

Other common errors: The other common errors were not validating user input as appropriate; *quit()* methods that did not in fact quit the program or close the `Scanner` object as requested in the assignment instructions; using `System.exit()` inappropriately; and various different errors that caused the program to end with an exception under certain test conditions.

Validating user input: When the user was asked for a word to edit, the method invoked should check that the word was, or was not, in the dictionary as appropriate. If the user had made a mistake, then an error message should tell them this, and the method should either end, or the user should be asked for valid input. In the model answer, methods end if given invalid input, after printing an error message to standard output.

- ***addWord():*** the method should check that the word given was not in the dictionary, thus preventing users from adding a repeated word.
- ***editWord():*** the method should check whether the word entered by the user was in the dictionary. Attempting to edit a word that was not in the dictionary could cause a NullPointerException.
- ***removeWord():*** The method should check that the word entered by the user was in the dictionary. If the user entered a word that was not in the dictionary, then the method should not attempt to delete the word.

The quit() method: The *quit()* method was supposed to allow the user to save (or not save) changes to the dictionary in the text file, and after this to print a goodbye message, close the `Scanner`, and quit the user interaction loop. A very common error here was that the method did not in fact quit the program or close the `Scanner`. Some students did not seem to understand how to enforce the end of the program, which would end once the `boolean` variable *finished* became `true`. This was because the main user interaction loop in the *start()* method would continue as long as the *finished* variable was `false`. Hence the method should make the *finished* variable `true` in order to quit. Some students got around their lack of understanding by using `System.exit()` to end the program.

Students were expected to close the `Scanner` object by adding the statement `scanner.close();` to their *quit()* methods. This was not essential, but the question asked for the `Scanner` to be closed, as it is good practice to close system resources when they are no longer used.

Testing: A minority of students used `System.exit()` to end their *addWord(), editWord()* and/or *removeWord()* methods, but this had the effect of ending the entire program. Other *DictionaryUI* classes tested ended with an exception of one sort or another. These errors suggest that some students spent little or no time testing their work and tracing errors.

Question 3

Students had been given the files·*ThesaurusFileManager.java; ThesaurusWord.java* and *thesaurus.txt* with the assignment, and were now asked to write the classes: *Thesaurus; ThesaurusEditor;* and *ThesaurusUI*. Once the three additional thesaurus classes had been written, then all five classes and the text file should work together in the *ThesaurusUI* class to allow the user to edit a thesaurus, similarly to how the *DictionaryUI* class allowed the user to edit a dictionary.

In many ways answering this question involved much copying and pasting, as the three classes students were asked to write were very similar to their corresponding dictionary classes. There was an important exception, and this was that the *ThesaurusWord* class had one `String` field, and two `List` fields (the *DictionaryWord* class had three `String` fields). A *ThesaurusWord* object consists of a `String` word, a list of synonyms (words which mean the same thing as the thesaurus word) and a list of antonyms (words which mean the opposite). Since a word can have more than one synonym, and more than one antonym, a `List` variable was used for these fields.

Most errors seen in this question were errors that were repeated from question 2, plus problems with *addWord()* and *editWord()* methods linked to adding or updating the lists of synonyms and antonynms. Various mistakes were seen such that the lists were not edited appropriately. The most common error was that the methods did not parse the user's input to a list of synonyms or antonyms, but instead the user's input was added to a `List` as a `String`, and this `List` variable with its single entry was then used to make or amend a *ThesaurusWord*.

Question 4

Students were asked how the use of generic types could have simplified writing the various dictionary and thesaurus classes. A good answer would have explained that using generics lets you declare a variable that can hold different types, which can be bounded or unbounded. It would have gone on to state that the type variable would need to be bounded to a class common to both the *Dictionary* and *Thesaurus* classes. As *DictionaryWord* and *ThesaurusWord* both inherit from *Word* this class is a good candidate. Since there is a great deal of duplicate code in the classes, it would be possible to have one set of generic classes that contained the duplicate code, which the specific *Dictionary* and *Thesaurus* classes would then inherit from. So, for example, *DictionaryUI* and *ThesaurusUI* would have *AbstractUI<T extends Word>* as the superclass, with *DictionaryUI* extending *AbstractUI<DictionaryWord>* and *ThesaurusUI* extending *AbstractUI<ThesaurusWord>*.