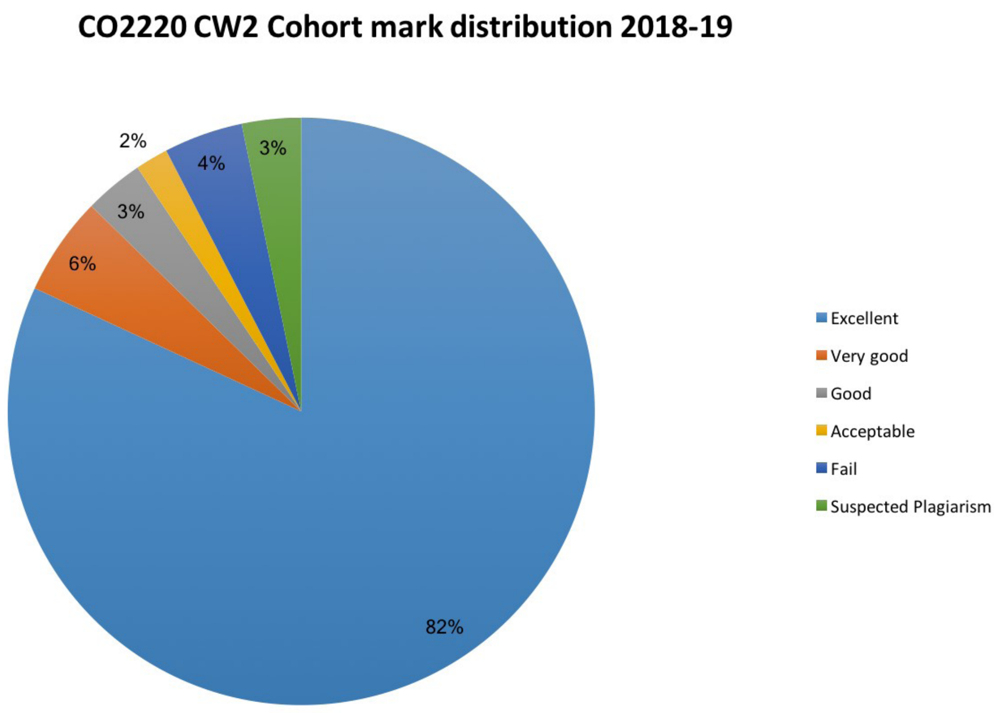
Coursework commentary 2018–2019

CO2220 Graphical object-oriented and internet programming in Java

Coursework assignment 2

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

See 2018–2019 cohort mark distribution for CO2220 CW2 below:



Model answers

The following files and model answers are provided with this report:

Part A

Java files

- *DatedMatchResult.java (with answer to Q4)*
- *DatedMatchResultParser.java (with answer to Q1)*
- *LeagueStanding.java*
- *LeagueStandingComparator.java (with answer to Q5)*
- *LeagueTableCreator.java*
- *LeagueTableCreatorGui.java (with part of the answer to Q2)*
- *LeagueTableFileWriter.java (with answer to Q3)*
- *LeagueTableFormatter.java*
- *SerializationUtil.java (with part of the answer to Q2)*

Note: *There was a mistake with the DatedMatchResultParser class given to students. The statement on line 39 of the DatedMatchResultParser class would fail on any machine that did not have its locale set to an English-speaking country.*

This meant that when dates had English words in them, 'Sunday' or 'May' for example, and the locale was not set to an English-speaking country then there would be an error. This has been corrected in the DatedMatchResultParser class given with this assignment, in that line 39 now sets the `Locale`*, and this import statement has been added:* `import java.util.Locale;`

Text files

- *2017-18 Premier League Results.txt*
- *2017-18 Premier League Results partial-1.txt*
- *2017-18 Premier League Results partial-2.txt*

Serialized files

- *2017-18 Premier League Results.ser*
- *2017-18 Premier League Results partial-1.ser*
- *2017-18 Premier League Results partial-2.ser*

Part B

- *ThreadedMatchResultsServer.java*
- *DatedMatchResultV2.java (with answer to Q2)*
- *2017-18-PLResults.ser*
- *MatchResultsClient.java (model answer to Q1)*

Comments on specific questions

Part A

In Part A students were given a system that would display football league results on a GUI. The GUI had a clickable list of dates, if a student clicked on a date then the GUI would display the results of the English premier league for 2017-18, up to and including the date clicked on.

Question 1

Students were asked to write four methods invoked by the *parseLine()* method in the *DatedMatchResultParser* class. The methods were invoked to help parse a `String` into a *DatedMatchResult* object. This question was challenging, and some students could not write a sensible answer at all. Of those who did, a minority had errors because they failed to allow for teams that have more than one word in their name, or because they mistook the order in which elements of the split `String` would appear in the `Array` made by the `String.split()` method.

The method that invoked the four methods that students were asked to write is below:

```
private void parseLine(String line, LocalDate date) {
   //line looks like e.g.: 'Crystal Palace 2-0 West Brom'
   String[] resultSplit = line.split("-");
   //resultSplit[0] will look like "<team> <score>",
                [1] will look like "<score> <team>"

   String[] homeResult = resultSplit[0].split(" ");
   String homeTeam = parseHomeTeam(homeResult);
   int homeGoals = parseHomeScore(homeResult);

   String[] awayResult = resultSplit[1].split(" ");
   String awayTeam = parseAwayTeam(awayResult);
   int awayGoals = parseAwayScore(awayResult);
   DatedMatchResult matchResult = new DatedMatchResult(date,
           homeTeam, homeGoals, awayTeam, awayGoals);
   results.add(matchResult);
}
```

Putting together the comments, and their own understanding of the `String.split()` method, students should have understood that the *resultSplit* Array held the team names and scores in two entries. The first entry in the *resultSplit* `Array` was a `String` in the order of home team / home score, while the second entry in the `Array` was a `String` with the order away score / away team. This meant that the *parseHomeScore()* method had to parse the ***final*** entry in the *homeResult* `Array` to an `int`, while the *parseAwayScore()* method had to parse the ***first*** item in the *awayResult* `Array` to an `int`. Some students ignored the different order of the home and away teams and scores, and attempted to parse the final entry in the `Array` in both cases. This would cause an exception as the *parseAwayScore()* method attempted to parse a `String` that was part or all of a team name to an int.

Despite the examples given in the comments with the *parseLine()* method making it clear that some teams had two parts to their name, some students failed to recognise that both the *homeResult* and the *awayResult* `Arrays`, would contain two entries if the team had one word in its name, and three entries if the team had a two-word name. These students assumed that the *awayResult* and *homeResult* `Arrays` would always have two entries only. Their methods attempted to directly access `Array` elements with the index numbers 0 and 1. This would mean that the *parseHomeScore()* method would throw a `NumberFormatException` as sometimes `homeResult[1]` would be part of a team name and incapable of being parsed to an `int`.

In a good answer, both the *parseHomeTeam()* and the *awayHomeTeam()* methods had to account for the case when the team had a one-word name, and when it had two. This could be done by checking the length of the *homeResult* and the *awayResult* `Arrays`, as most students realised. A minority recognised that the *homeResult* and the *awayResult* `Arrays` could contain two or three elements, but did not attempt to include the second part of the name with the first, resulting in such things as a team called *Man* who received all of both Man City and Man United's results, and team *West* who received all of West Brom and West Ham's results.

Question 2

Question 2 asked students to write a serialization method in the *SerializationUtil* class, to give the class an 'appropriate' constructor, and to write an inner class in the *LeagueTableCreatorGui* class. The inner class would listen to a `JButton`, and, if pressed, would ask the user for the name of a file to save serialized results to, and then invoke the serialization method in the *SerializationUtil* class.

Asking students to write a constructor for the *SerializationUtil* class caused some controversy, with some students writing comments pointing out that a static utility class did not need a constructor. Students had been given reading for Part A, which included pages 275–278 of *Head First Java.* If all students had done this reading, they would have understood from page 276, that writing a private constructor would prevent the inappropriate instantiation of the class. Therefore, an appropriate constructor for the *SerializationUtil* class would be private, and empty:

```
private SerializationUtil(){}
```

Many students did not even attempt to write a constructor, many wrote an empty public constructor, some felt the need to have their public constructor take some actions, for some this meant writing unnecessarily complicated public constructors. Only a minority correctly wrote a private constructor.

When writing the inner class *SerialiseButtonAction* in the *LeagueTableCreatorGui*, a large minority showed no understanding of serialization, by copying and renaming the *DeserialiseButtonAction* inner class, and making trivial, incorrect or no changes. Some of these students copied the *deserialize()* method from the *SerializationUtil* class, and renamed it *serialize(),* once again making trivial, incorrect or no changes, usually with the result that the method would deserialize. Hence these students did not manage to serialize the *matchResults* `ArrayList`, since their *SerializeButtonAction* inner class would attempt to deserialize.

Some students who wrote a correct *SerialiseButtonAction* inner class, attempted to write a serialize() method by copying the *deserialize()* method in the *SerializationUtil* class, and renaming it *serialize()*. These students then changed every occurrence of `ObjectInputStream` to `ObjectOutputStream`, and also changed `FileInputStream` to `FileOutputStream`. The problem with this was that while they now had correct syntax for serializing, they had not given their method an ArrayList parameter, and had not deleted the first line of the *deserialize()* method, which is: `ArrayList<DatedMatchResult> clubs = null;`

This meant that many students wrote a *serialize()* method that made and serialized a `null ArrayList`. In some cases, this `null ArrayList` was then saved as the *matchResults* `ArrayList`, i.e. the `ArrayList` that was displayed on the GUI in the *LeagueTableCreatorGui* class, causing the class to end with a `NullPointerException`.

In some cases, students did not appreciate that once the button listened to by the *SerialiseButtonAction* inner class was pressed, it was the *matchResults* `List` that should be serialized. These students attempted to make new `Lists` by parsing the contents of a file, with the name given by the user. Since the user would think that they were giving a file name to save the serialized list into, this was unlikely to work, as the file either did not exist, or did but was unlikely to be capable of being parsed into *DatedMatchResult* objects.

Question 3

Question 3 asked students to write a method in the *LeagueTableFileWriter* class to save the league table displayed on the GUI as a text file. This was written correctly by the vast majority of students. A few miscellaneous errors were seen, with the only common errors being firsly, those students who forgot to close the file after writing to it, and hence the text file that their method made was empty; and secondly, a few students who used an `ObjectOutputStream` chained to a `FileOutputStream` for saving, which meant that the 'text' file contained some non-text data.

Question 4

In Question 4, students were told that in order to be able to add new results to the league, while excluding any duplicate results, the *DatedMatchResult* class would need to override *equals()* so that the comparison worked when two *DatedMatchResult* objects were compared that were in different places in memory. Overriding *equals()* meant, for reasons not gone into, that `Object's` *hashCode()* method would also need to be overridden, however, this had already been done in the *DatedMatchResult* class. In fact, a start had been made on the *equals()* method, but it was not finished. Students were asked to complete the method so that the buttons to add extra results work to the GUI worked as they should; that is duplicate results were rejected.

While some students wrote a correct *equals()* method, a significant minority did not attempt this question at all. Other students made a serious attempt at the question, but the resulting method did not quite work as it should. Finally a small group of students made trivial changes to the *equals()* method that in no way addressed the question. However, this was quite a challenging question. The model answer is as follows:

```
public boolean equals(Object o) {
1. if (this == o) return true;
2. if (o == null || getClass() != o.getClass()) return false;
3. DatedMatchResult that = (DatedMatchResult) o;
4. return homeScore == that.homeScore &&
   awayScore == that.awayScore &&
   homeTeam.equals(that.homeTeam) &&
   awayTeam.equals(that.awayTeam) &&
   date.equals(that.date);
}
```

The statement numbered 1 means if the object being compared to the parameter points at the same place as the parameter, return `true`. The statement numbered 2 says that if the parameter is `null`, or the class of the parameter is different to the object it is being compared to, return `false`. Statement 3 casts the object parameter to a *DatedMatchResult* object called *that*. Finally, statement 4 says that if each of the fields of the *DatedMatchResult* object and the corresponding field in *that* consist of the same data, then the objects are equal so return `true`.

Question 5

Question 5 asked students to amend the *LeagueStandingComparator* class. The class compared points to find teams' standing in the league. If teams were equal on points, then the *LeagueStandingComparator* class would compare goal difference. If teams were equal on points and goal difference, then they were considered equal. Students were asked to add another comparison following on from the first two, teams that were equal on points and goal difference should have the number of goals scored by each team compared. The team with the higher number of goals would be ahead of the other team in the league. However, if both teams had scored the same number of goals then they were equal.

Note that the *LeagueTableCreator* class given to students would create a `List` of *LeagueStanding* objects from the *matchResults* variable, an `ArrayList` of *DatedMatchResult* objects. It was the *LeagueStanding* class that held the statistics about a team that allowed their ranking in the league to be determined. The *goalsFor* field in the *LeagueStanding* class held the number of goals scored by a team up to a certain date.

Students needed to change the *compare()* method in the *LeagueStandingComparator* class, firstly by removing that the goal difference comparison could return a value indicating that the two objects were equal. The goal difference comparison should have been changed to return values indicating only that one score was greater or lesser than the other. Returning whether or not two objects were equal should then be left to the final comparison, the second change needed, adding a comparison of the *goalsFor* field.

In the final version of the class, the *points* field should be the first field of the *LeagueStanding* objects to be compared by the *compare()* method in the

LeagueStandingComparator. If the *points* field of both objects were different then a ranking had been achieved. In this case the method would return a value indicating that one team was ranked ahead of the other and the method would end. If they were equal then no return would be made and the comparison of the *goalDifference* fields would be next. Again, if these fields were unequal then the *compare()* method in the *LeagueStandingComparator* would return a value indicating which team was ranked ahead of the other, and method would end. If the *goalDifference* fields had the same values, then the final comparison would be done. The *compare()* method would compare the number of goals scored by each team (*goalsFor*), returning values indicating that one team was ranked ahead of the other, or that they were equal.

Most students successfully made the changes, however a number implemented changes that did not compare the *goalsFor* field of the *LeagueStanding* class, and so had no hope of achieving the desired outcome.

Question 6

Students were asked to write a long comment in the *LeagueTableCreatorGui* class, that described how the GUI would give the results of the league up to a particular date, when the user clicked on that date. A large minority of students (about 15 per cent of submissions) made no attempt at this question. Of those that did, most students had a reasonable attempt at this question, although some comments were rather vague, and too short to include all the steps taken, even described at a high level.

A good answer might have said the following:

> The *createDatesScrollPane()* method in the *LeagueTableCreatorGui* class makes a `JScrollPane` displaying a `JList`, in this instance a clickable list of objects of type `LocalDate`. An inner class implementation of the `ListSelectionListener` interface listens to the `JList` and when a date is clicked by the user it invokes the inner class *DateListSelectionListener* which calls the *constructLeagueTable()* method passing it the date which was clicked.
>
> In the first place, when the *LeagueTableCreatorGui* class is run, the GUI is constructed and displayed to the user, but it contains no data. The user will click one of the two buttons that can make a league table. Both buttons are being listened to by inner classes that will create a new league table by first making an `ArrayList` of *DatedMatchResults* objects from a text or serialized file. This `ArrayList` is then used to construct a `List` of dates on which games were played, and this `List` is displayed on the GUI in the `JScrollPane`. Finally, the inner class will call the *LeagueTableCreatorGui's constructLeagueTable()* method, in this first instance with the current date as its parameter. The method will make an *LeagueTableCreator* object. The *LeagueTableCreator* object's *calculateLeagueTable()* method is then invoked, with the date as its parameter, and it will use the `ArrayList` of *DatedMatchResults* objects to make and return a `List` of *LeagueStanding* objects. Each *LeagueStanding* object in the `List` will have statistics about the games played by a particular team (including points and goal difference), up to and including the date given to the *constructLeagueTable()* method.
>
> Finally, the *constructLeagueTable()* method formats the `List` into a `String`, using the *format()* method of the *LeagueTableFormatter* class. This will take the `List` of *LeagueStanding* objects and format the data in each object into a row for display in the GUI. In order to display the data in nicely aligned columns, the team with the longest name is found, so that enough space can be given to team names without disturbing the alignment into

columns of the rest of the data. The *constructLeagueTable()* method puts the formatted `String` onto the `JTextArea`. Now that the GUI has data, the user can view the league table for certain dates by clicking on one of the dates in the `JScrollPane`. In order to make a new dated league table to display, the *DateListSelectionListener* inner class will invoke the *constructLeagueTable()* method, passing it the date which was clicked.

Question 7

Students were asked to follow advice on readability given with the assignment. There was limited scope to follow the advice, as many variable and method names were given to students. The advice also asked for properly formatted text in Java files, which massively improves readability. The examiners saw no issues with the readability of any submission that made a serious attempt at the assignment.

Part B

Question 1

Students were asked to write a class, *MatchResultsClient*. The class should connect to the *ThreadedMatchResultsServer* and accept a serialized file containing an `ArrayList` of *DatedMatchResultV2* objects. The client should then deserialize and print the `ArrayList` contents to standard output using an enhanced `for` loop. In addition, students were told to write a catch block for each of the possible subclasses of `Exception` that the class could throw, with each catch block giving some information about the exception thrown.

A minority of students found this question very challenging. Some gave in a client class that looked as if it had been written to interact with a server in a different way. A small number of students wrote classes that extended the *ThreadedMatchResultsServer* class, which was not what was needed at all, while others wrote a *MatchResultsClient* that would not compile.

Most errors were more minor, with 60 per cent of students gaining full marks. However, 40 per cent of submissions contained at least one error in Question 1. The examiners also made an error, in not asking students to insert a main method into their class to allow for testing. Some students wrote a comment noting that it was not clear whether writing a main was a requirement or not. In fact, the examiners added a main to any class that did not have one, in order to test it, with no deduction of marks.

Some students misunderstood the instruction to output the contents of the `ArrayList` in a `for` loop, and put the connection into a `for` loop, so the connection would be made, the file saved, deserialized and the contents printed, an arbitrary number of times.

Some students wanted the name of the server to be entered on the command line, although this was not asked for in the assignment instructions. In addition, these students had not written a default alternative, hence should the user fail to give a valid entry the class would end. Others asked for the name of the server and the port number to be entered from the command line, similarly no entry or invalid entry meant the class would end.

A few students had problems with deserializing. Some attempted to read and print the file directly, using a `BufferedReader` chained to an `InputStreamReader` which resulted in some odd-looking output. For some reason a small number of students made changes to the *DatedMatchResultV2* class, such as adding a new variable, which caused the *readObject()* method of the `ObjectInputStream` class to throw an `InvalidClassException`.

Most students successfully connected and deserialized, and a number of these then had problems writing an enhanced `for` loop. A few students did not use the enhanced `for` loop at all, while others did but wrote tremendously complicated, and quite unnecessary code, to format the output of the `ArrayList`. These students did not seem to appreciate that using a simple enhanced `for` loop, for example:

```
for (DatedMatchResultV2 mr : results) System.out.println(mr);
```

would mean that the JVM would look for a *toString()* method to print the *DatedMatchResultV2* object, and, finding one in the class would use that. Nothing more complicated was required.

Students were asked to write one catch block for every exception that their class could throw. Perhaps, more accurately, it would have been better to ask students to write one catch block for every *checked* exception that could be thrown. The model answers have 6 catch blocks, but there are others that could have been included. For example, the `ObjectInputStream` constructor can throw four exceptions - see:

https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html#ObjectInputStream(java.io.InputStream)

Only two of these exceptions have been included in the catch block (`StreamCorruptedException` and `IOException`). The other two of these exceptions (`SecurityException` and `NullPointerException`) are subclasses of `java.lang.RuntimeException`, which means they are unchecked exceptions. These are exceptions that should not happen because our code should be robust enough to prevent or exclude them. Many students included unchecked exceptions in their catch blocks, which was probably down to the imprecise wording of the question. No deduction was made.

A minority of students made no attempt at this part of the question, throwing every exception with no catching. Similarly, other students used a single `Exception` catch block for all exceptions, which the assignment instructions explicitly ruled out.

One thing to note is that some students seemed to include catch blocks for exceptions that it did not seem likely their class could throw. One way to determine what exceptions can be thrown is to look in the API. For example, the model answer has these statements in its try block:

```
1. Socket socket = new Socket("localhost", port);
2. ObjectInputStream       ois    =      new
       ObjectInputStream(socket.getInputStream());
3. ArrayList<DatedMatchResultV2>        results      =
       (ArrayList<DatedMatchResultV2>)ois.readObject();
4. ois.close();
5. socket.close();
```

The first statement can throw any exception that the `Socket` constructor that takes a `String` and an `int` can throw - see:

https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html#Socket(java.lang.String,%20int).

The second statement would mean looking in the API for the appropriate `ObjectInputStream` constructor, and at the `getInputStream()` method in the `Socket` class. The third statement would mean looking at the *readObject()* method of the `ObjectInputStream` class, the fourth would mean looking at the *close()* method in the `ObjectInputStream` class, and the fifth would mean checking the *close()* method of the `Socket` class.

A common error was to have several catch blocks, but not to include a catch block for the `ClassNotFoundException` that could be thrown by the *readObject()* method of `ObjectInputStream` - see:

https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html#readObject()

Instead students included an Exception catch block, which, of course, will catch every exception not already caught, as all exceptions are subclasses of Exception. Some students threw some exceptions and caught others, some both threw and caught the same exceptions. Another mistake was made by those students who left some or all of their catch blocks empty, so that no information would be given about the exception.

Question 2

Students were asked to write a *toString()* method in the *DatedMatchResultsV2* class that would print the fields of the class in the order: *date, homeTeam, awayTeam, homeScore, awayScore*. The method should output results in columns, such that they are easy for the user to read. Students were given the example below:

```
2018-05-13         West Ham      Everton       3 1
2018-05-13         Spurs         Leicester     5 4
2018-05-13         Swansea       Stoke         1 2
```

Students were expected to use the `String.format()` method, with formatting codes, to give a fixed amount of space to the date and team names, enough so that each date and team name could fit inside the space given. In the model answer, see below, 15 spaces are given to each date and team name.

```
public String toString(){
   String s = String.format("%-15s %-15s %-15s %01d %01d",
         date, homeTeam, awayTeam, homeScore, awayScore);
   return s;
}
```

About 7 per cent of students made no attempt at this question, while about 9 per cent wrote a *toString()* method, but with no attempt to format the output into columns. Despite this, most students used `String.format()` and format codes in their *toString()* methods, and wrote something that worked well. A few students attempted to format the output but without allowing enough space for team names, so the columns were uneven, or used tabs for the spacing, which again can give uneven columns in the output. A few students made errors with their attempts to use `String.format()`, missing out certain fields, or printing fields in the wrong order.

The coursework assignment instructions noted that "`LocalDate` has a *toString()* method so its output can be formatted as a `String`." Despite this a small number of students made complicated, and unnecessary, attempts to format the date. In the model answer above, the format codes used for the date are `%-15s`, which means give the variable 15 spaces and look for a *toString()* method to return the data.

A number of students did not write a *toString()* method, but instead attempted to format the output in the *MatchResultsClient*, often in quite complicated ways. This was puzzling as writing a *toString()* using `String.format()` and formatting codes was a much simpler approach. Other students simply tried to print the entire `ArrayList`, without using a loop at all. This meant that *toString()* for a `Collection` was invoked, which printed everything on one long line, enclosed with square brackets, with the fields of each entry in the `ArrayList` separated by a comma from the next entry's fields. This was hard to read and was not at all what the assignment asked for.

Some answers given, while correct, demonstrated that many students did not understand that *toString()* methods do not have to be explicitly invoked. While it is correct to explicitly call *toString()*, for example:

```
for (DatedMatchResultV2 mr : results)
    System.out.println(mr.toString());
```

it is not necessary. When `System.out.println()` is called on an object, the JVM will look for a *toString()* method in the inheritance chain. Note that if the class of the object has a *toString()* method then that will be used. If it does not the JVM will look up the inheritance chain until it finds one. Ultimately, if no other *toString()* method is found, the JVM will use the *toString()* method from `Object`, which all classes inherit from. Hence the following is an equivalent statement to the one immediately above:

```
for (DatedMatchResultV2 mr : results) System.out.println(mr);
```

Question 3

Marks for question 3 were awarded for writing properly formatted and readable code. All students who made a serious attempt at the question received these marks.