University of London
Computing and Information Systems/Creative Computing
CO2220 Graphical object-oriented and internet programming in Java
Coursework assignment 2 2018–19

Contents

Introduction

This is coursework assignment 2 (of two coursework assignments total) for 2018–19. Part A looks at a more sophisticated GUI than the one seen in coursework assignment 1, as well as: saving our data using text files and by serializing objects; static utility methods and classes; parsing text files to objects; overriding `Object's` *equals()* method; and the `Comparator` interface. Part B asks that you demonstrate an understanding of simple network programming, `Exceptions`, *toString()* methods and `String` formatting.

IMPORTANT NOTE: Please use at least **Java 8** to compile and test your programs. Later versions of Java are also fine.

Electronic files you should have

Part A

Java files

- *DatedMatchResult.java*
- *DatedMatchResultParser.java*
- *LeagueStanding.java*
- *LeagueStandingComparator.java*
- *LeagueTableCreator.java*
- *LeagueTableCreatorGui.java*
- *LeagueTableFileWriter.java*
- *LeagueTableFormatter.java*
- *SerializationUtil.java*

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*
- *2017-18-PLResults.ser*

Reading

Please note that the reading given for part A and part B is carefully chosen to reflect the tasks you are asked to achieve. Every year students miss marks for their submissions that they very likely would have gained had they paid attention to the recommended reading.

What you should hand in: very important

At the end of each section there is a list of files to be handed in – **please note the hand-in requirements supersede the generic University of London instructions**. Please make sure that you give in **electronic versions** of your Java files since you cannot gain any marks without handing in the .**java** files asked for. Class files are **not** needed, and any student giving in only a class file will not receive any marks for that part of the coursework assignment, **so please be careful about what you upload as you could fail if you submit incorrectly**.

- Please only hand in the Java files asked for, and not any additional files.
- Please put your name and student number as a comment at the top of each Java file that you hand in.

There is one mark allocated for handing in uncompressed files – that is, students who hand in zipped or .tar files or any other form of compressed files can only score 99/100 marks.

There is one mark allocated for handing in just the Java files asked for without putting them in a directory; students who upload their files in a directories can only achieve 99/100 marks.

There are two marks for (1) not changing the names of the classes given to you, and (2) for naming any classes that you have to write (in this assignment *MatchResultsClient.java*) ***exactly*** as you have been asked to name them.

Anything added to the names given means that your files are incorrectly named, for example the following count as wrong names:

- *JSmith_ MatchResultsClient.java*
- *cwk2- MatchResultsClient.java*
- *JSmith-220-assignment2- MatchResultsClient.java*
- *MatchResultsClient .java (note space before .java)*

Using a file name that differs from the class name leads to a compilation error caused by a file name / class name mismatch.

The examiners will compile and run your Java programs. For this reason, programs that will not compile will not receive any marks.

You are asked to give your classes certain names, please follow these instructions carefully. If your file does not compile ***for any reason*** (except file name / class name mismatch) you will receive no marks for that part of the assignment. In particular, files that contain Java classes that cannot be compiled because they are the wrong type (*e.g.* PDFs) will not be given any marks.

Part A

The *LeagueTableCreatorGui* class constructs a football league table and displays the result on the `JTextArea` of the GUI. The league can be constructed from text files or serialized files. Serialized files must contain an `ArrayList` of *DatedMatchResult* objects, and text files must be laid out in a particular way for successful parsing to a *DatedMatchResult* object. The files you have been given will construct the English Premier League table for 2017-18; naturally other league tables are possible with different data.

The *2017-18 Premier League Results.txt* file has all the games played in the 2017-18 season, and the serialized file, *2017-18 Premier League Results.ser,* contains all the 2017-18 results as an `ArrayList` of *DatedMatchResult* objects.

The GUI also has a `JScrollPane` displaying a clickable list of dates. Clicking on a date gives the partial league results up to and including that date. This could allow a user, for example, to trace their team's progress throughout the football season.

League tables can be constructed either in one go, or by adding in extra results to a partial league table. In the second case the GUI's *addNewResults(List<DatedMatchResult>)* method is used to make sure that the same match result is not added twice.

Overriding `Object`'s equals() method

Note every Java class inherits from `Object`, hence they inherit its public methods, one of which is the *equals()* method. The API notes that "this method returns true if and only if [both objects] refer to the same object" – see:
https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

This means that the method returns true only if two objects are pointing to the same place in memory. Hence, two objects may be in the same state (*i.e*. their instance variables hold or point to the same values), but if they are held in different places in memory they are not the same.

However, you can use equals to compare `Strings`, and to compare `LocalDates` to see if their fields are pointing at or holding the same data, because both classes override *equals().* – see https://stackoverflow.com/questions/2265503/why-do-i-need-to-override-the-equals-and-hashcode-methods-in-java for more information.

The `JFileChooser` class

The *LeagueTableCreatorGui* class uses the `Swing` class `JFileChooser` for selecting files to open and to save to. The class is itself a GUI used for navigating the local file system. More information can be found at the following links:

- https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html
- https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

Why you should use at least Java 8 to compile and test your work

The classes you have been given for this assignment use the diamond operator, the `Files` and `LocalDate` classes that are not available in earlier versions of Java.

The diamond operator (from Java 7)

Some of the classes you have been given for part A use **the diamond operator**, meaning that the type of an `ArrayList` can be inferred by the JVM. For example this declaration in the *buildDateList()* method of the *LeagueTableCreatorGui* class:

```
List<LocalDate> dates = new ArrayList<>();
```

From the left side of the declaration, the JVM infers that the `ArrayList` is of type `LocalDate`.

The `Files` class (from Java 7 and 8)

`Files`, introduced in Java 7, is a class with static methods that operate on files and related data structures, it includes the method ***readAllLines(Path, Charset)***. This method reads all lines from a file, recognises standard line endings, and closes the file after use.

The *parse(File*) method in the *DatedMatchResultParser* class uses the Java 8 `Files` method ***Files.readAllLines(Path)*** that takes only a `Path` parameter. Bytes from the file are decoded into characters using the UTF-8 `Charset`.

For more information on the `Files` class, see the API:
https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

The `LocalDate` class (from Java 8)

We are using the `LocalDate` class in preference to the `Calendar` or `Date` classes. `LocalDate` is simpler to use than the `Calendar` class. Note also that some of the constructors and methods of the `Date` class are now deprecated.

The `LocalDate` class overrides *equals()* from the `Object` class, so that comparing two `LocalDate` variables with *equals()* gives the result we expect – they are equal if they both point at the same date; dates do not have to be in the same place in memory. `LocalDate` is a final class so it cannot be extended, and has both instance and static methods. Static methods include *now(),* for example:

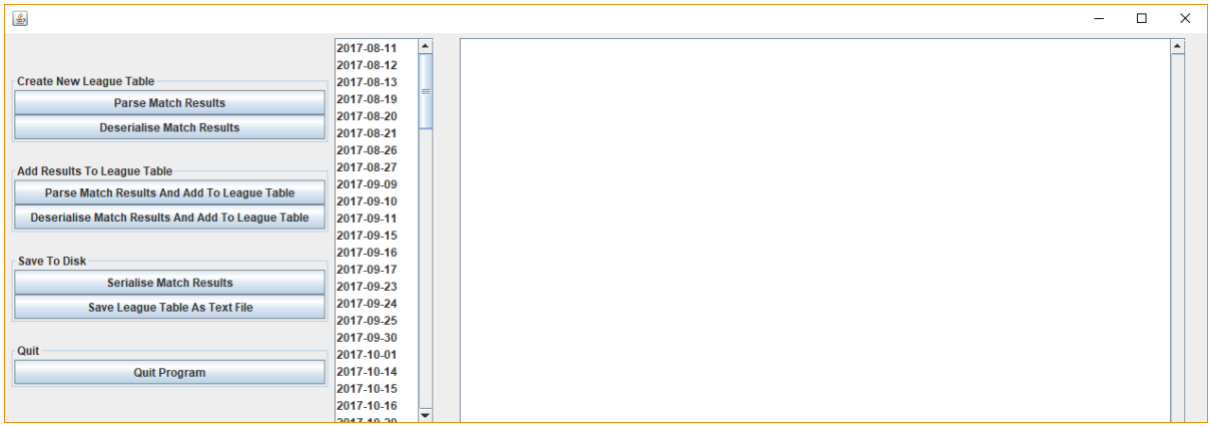
```
LocalDate date = LocalDate.now();
```

will give a new `LocalDate` object initialised with today's date.

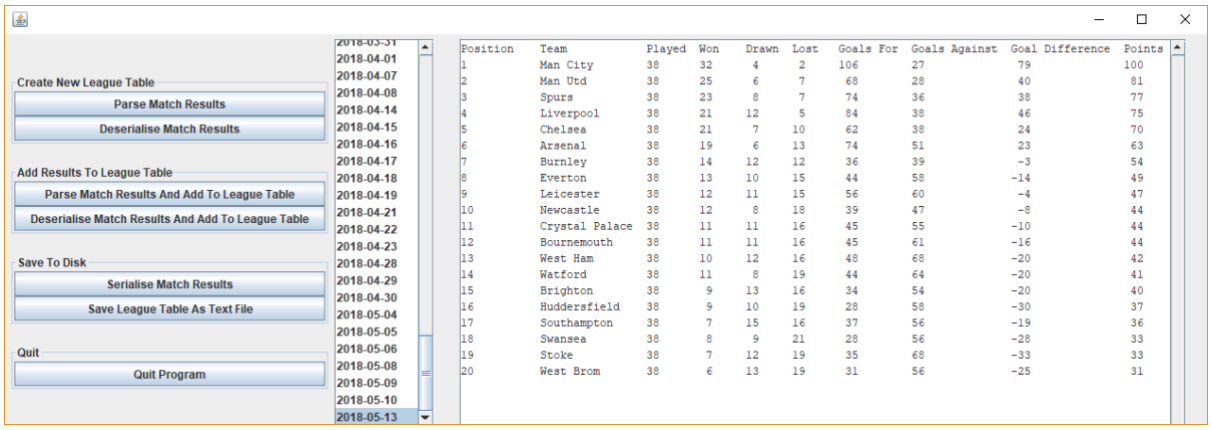
`LocalDate` has a *toString()* method so its output can be formatted as a `String`.

The LeagueTableCreatorGui class

Compile and run the *LeagueTableCreatorGui* class. Choose one of the buttons under the label `Create New League Table`. If you choose the `Parse Match Results` button and then the *2017-18 Premier League Results.txt* file you will see that the GUI displays nothing on the `JTextArea`, as in the screenshot below.



If you click the `Deserialize Match Results` button and choose the serialized file, *2017-18 Premier League Results.ser*, then you will see that the GUI displays the final rankings of the English Premier League for the 2017-18 football season, as in the screenshot below.



The questions

Please complete the following tasks:

1. When you compile and run the GUI, and choose the `Parse Match Results` button and then the *2017-18 Premier League Results.txt* you will find that the `JTextArea` is empty. This is because the *DatedMatchResultParser* class is not parsing the results given to it to a *DatedMatchResult* objects successfully. You will also see the shell displaying information about a `NullPointerException` that has been thrown by the *updatePlayedStats(DatedMatchResult)* method in the *LeagueTableCreator* class. `NullPointerException` is an unchecked sub-class of `Exception` and hence it has not been handled with try/catch. The exception will not be thrown once the *parseLine(String, LocalDate)* method works as it should.

   The following methods called by the *parseLine(String, LocalDate)* method in the *DatedMatchResultParser* class return dummy values in order that the GUI will compile. Complete the methods so that the *parseLine(String, LocalDate)* method works as it should to parse its `String` and `LocalDate` parameters to a new *DatedMatchResult* object.

   - *parseHomeTeam(String[ ])*
   - *parseHomeScore(String[ ])*
   - *parseAwayTeam(String[ ])*
   - *parseAwayScore(String[ ])*

   [12 marks]

2. If the user clicks on the `Serialize Match Result` button, the GUI takes no action. What should happen when the button is pressed is that the `JFileChooser` asks the user for a file name to serialize to, then a serialization method in the *SerializationUtil* class is called, and the `ArrayList` *matchResults* containing *DatedMatchResults* objects will be serialized into the file named by the user.

   - Write the serialized method in the *SerializationUtil* class.
   - Write the inner class *SerializeButtonAction* and remove the comment marks from the statement given below to test your work.
   - Write an appropriate constructor for the *SerializationUtil* class.

   The statement:

   ```
   serializeButton.addActionListener(new
   SerializeButtonAction());
   ```

   has been commented out. You will need to remove the comment marks from the statement to test your work.

   [18 marks]

3. When you press the `Save League Table As Text File` button, the `JFileChooser` asks for a file name to save to, however no file is made and nothing is saved.

   The *saveLeagueTable(File)* method of the *LeagueTableCreatorGui* class uses the *getText()* method from the `JTextArea` class to take the contents of the `JTextArea`, which includes line breaks, as a `String` and send it to the *write(String, File)* method of the *LeagueTableFileWriter* class. The *write(String, File)* method's body is empty, hence no file is made and nothing is saved. Complete the method, being sure to save the text to a file with the UTF-8 charset.

   Note that the *getText()* method is inherited by the `JTextArea` class from `JTextComponent`. See the API for more details. **[6 marks]**

4. In order to be able to add new results to the league, while excluding any duplicate results, we need to override *equals()* in the *DatedMatchResult* class so that the comparison works when two *DatedMatchResult* objects are compared that are in different places in memory. With the system as it is given to you, you can check that the current methods of excluding duplicate entries from the *matchResults* `ArrayList` do not work, by first constructing the league with the `Deserialize Match Results` button and the serialized file, *2017-18 Premier League Results.ser.* After this, press the `Deserialize Match Results And Add To League Table` button and choose the *2017-18 Premier League Results.ser* file again.

   Overriding *equals()* means we also have to override `Object's` *hashCode()* method, which has been done.

   Complete the *equals()* method in *DatedMatchResult* so that the buttons to add extra results work as they should. **[12 marks]**

5. The *LeagueStandingComparator* class compares points to find teams' standing in the league. If teams are equal on points then it compares goal difference. If teams are equal on points and goal difference then they are equal. Change the comparator so that it implements an additional comparison: if teams are equal on points and goal difference, then the comparator compares the number of goals scored by each team. The team with the higher number is ahead of the other team in the league. However, if both teams have scored the same number of goals then they are equal. **[12 marks]**

6. Describe how the *LeagueTableCreatorGui* class can give the results of the league up to a particular date, when the user clicks on that date. You do not need to describe in detail how each method and class involved in producing this result works: a high level overview will be enough.

   For example, you might want to begin your explanation as follows:

   > 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...

   Give your explanation as a long comment at the bottom of the *LeagueTableCreatorGui* class. A long comment means 2 or 3 paragraphs at most. A paragraph is at most 8 sentences. **[6 marks]**

7. When adding new methods and variables, or renaming existing ones, please remember to make your names as meaningful as possible, following the advice from *Clean Code* given in coursework assignment 1, and reproduced here in Appendix A. **[3 marks]**

Reading for part A

The following chapters and pages of volume 2 of the subject guide, and associated *Head First Java* (HFJ) readings:

- Subject guide chapter 2, sections 2.2, 2.11 and HFJ 275-278 (static utility classes)
- http://www.javapractices.com/topic/TopicAction.do?Id=40 (private constructors)
- Subject guide chapter 3, sections 3.2, 3.3, 3.5, 3.6 and HFJ 319-326, and 329-333 (handling exceptions)
- Subject guide chapter 5, sections 5.3 and HFJ 447, 452-454 and 458-9 (files, parsing with *String.split(),* streams including *BufferedReader* and *BufferedWriter*)
- Subject guide chapter 6, sections 6.2, 6.3, 6.4 and HFJ 429-438; 441-3 (serialization)
- https://stackoverflow.com/questions/2265503/why-do-i-need-to-override-the-equals-and-hashcode-methods-in-java (overriding *equals()*)

Deliverables for part A

Please put your name and student number as a comment at the top of your Java files. Please only hand in the files asked for.

Please submit an electronic copy of your amended classes as follows:

- *DatedMatchResultParser.java*
- *SerializationUtil.java*
- *LeagueTableCreatorGui.java (with your changes plus your answer to Q6)*
- *LeagueTableFileWriter.java*
- *DatedMatchResult.java*
- *LeagueStandingComparator.java*

Part B

Introduction

You have been given the file *ThreadedMatchResultsServer.java*, together with *DatedMatchResultV2.java* and *2017-18-PLResults.ser*.

The *ThreadedMatchResultsServer* class is threaded, this means that it can connect to multiple clients at the same time. It also means that a particular client can connect and disconnect repeatedly, as the *ThreadedMatchResultsServer* does not die when a client disconnects (which it would do if not threaded).

The *ThreadedMatchResultsServer* sends a serialized `ArrayList` to the client. The `ArrayList` is deserialized from the file *2017-18-PLResults.ser*.

The questions

Please complete the following tasks:

1. Write the class *MatchResultsClient*. The *MatchResultsClient* should do the following:

   - connect to the *ThreadedMatchResultsServer* and accept the serialized `ArrayList` from the server
   - deserialize the `ArrayList`
   - print the contents of the deserialized `ArrayList` to standard output, by printing each `ArrayList` element individually in an enhanced `for` loop.
   - handle exceptions. Your class should not have an `Exception` catch block, but instead should have one catch block for each of the possible sub-classes of `Exception` that the class could throw.
   - Each catch block should give some information about the exception thrown.

   [18 marks]

2. Write a *toString()* method in the *DatedMatchResultsV2* class. Your method should print the fields of the class in the order: *date, homeTeam, awayTeam, homeScore, awayScore*. Your method should output results in columns, such that they are easy for the user to read, as in the example below.

   [6 marks]

3. While writing your class please remember to make your names as meaningful as possible, following the advice from *Clean Code* (see appendix A). **[3 marks]**

***Example of first three lines of output of the deserialized `ArrayList`, using a* toString() *method in* DatedMatchResultV2**

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

Reading for part B

- Subject guide volume 1 sections 6.3 and HFJ page 116 (the enhanced for loop)
- Subject guide volume 2, sections 3.1 – 3.6 and HFJ 319-326, and 329-333 (handling exceptions)
- Subject guide volume 2, sections 7.1 to 7.10 (clients and servers)
- https://www.geeksforgeeks.org/java-string-format-examples/ (using Java String *format()* with examples). Note that the `LocalDate` class has its own *toString()* method and hence can be formatted as a `String`.
- Subject guide volume 2 Chapter 12 (Threads).

Deliverables for part B

1. An electronic copy of your class: *MatchResultsClient.java*
2. An electronic copy of your amended class: *DatedMatchResultV2.java*

Appendix A: advice on readable code from coursework assignment 1

1.0 Readable code

In his book *Clean Code: A Handbook of Agile Software Craftsmanship* (published in 2008 by Prentice Hall, ISBN 978-0132350884). Robert C Martin describes a system for writing readable code.

Mr Martin writes:

> One difference between a smart programmer and a professional programmer is that the professional understands that *clarity is king*. [...] We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

Mr Martin writes that 'making your code readable is as important as making it executable'. He believes that names of variables, methods and classes are a major part of what makes our code readable:

> 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.

Martin dislikes comments, noting that as code is updated comments are rarely updated at the same time, so however helpful a comment at the start, once a class has been in use for a while any comments are likely to be outdated and confusing. He believes that code should be written with names that make the intent clear, such that comments are redundant.

You should note that the programmer has tried to follow the advice given by Martin in writing classes. However, Martin himself notes that names can always be improved, and that we should not be afraid to keep refining our code.

See Appendix B for an example of renaming a simple class to make it more readable.

When answering questions in this assignment, please note the following rules outlined by Martin:

1.1 Formatting

> "You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the layout of your code, and then you should consistently apply those rules. […] It helps to have an automated tool that can apply those formatting rules for you."

1.2 Methods

- **Method should do one thing only**. If your method does more than one thing, break it into separate methods.
- **Do not repeat yourself** – if you find yourself writing the same code more than once, put it into a method.
- **Too many arguments (parameters to a method).** "No argument is best, followed by one, two and three. More than three is very questionable and should be avoided with prejudice."

1.3 Comments

> If you do write a comment, make sure it is grammatical, short, does not state the

obvious and is really needed.

1.4 Names

- **Choose descriptive names** "Names in software are 90% of what makes software readable".
- **Unambiguous names** "choose names that make the workings of a function or variable unambiguous".
- **Names should describe side effects**, *e.g.* a method `getOos()` will make an `ObjectOuputStream` if one does not already exist, so should be called `createOrReturnOos()`

1.5 General

- **Obscured intent** – make the code as expressive as possible such that its intention is clear from a first reading.
- **Put conditional statements into a method to make their intention and effect clear**, *e.g.*

  BAD `if (guessedWord.length()< shortestLength)`

  GOOD `if(guessedWordIsTooShort(guessedWord))`

Appendix B

A simple example of renaming methods and variables for greater readability.

Original

```
public class Calculator{

      public static void calc(int x){
            if (x >= 70){
                  System.out.println("grade = A");
                  return;
            }
            if (x >= 60){
                  System.out.println("grade = B");
                  return;
            }
            if (x >= 50){
                  System.out.println("grade = C");
                  return;
            }
            if (x >= 40){
                  System.out.println("grade = D");
                  return;
            }
            if (x<40) System.out.println("grade = F");
      }

      public static void main(String[] args) {
            calc(90);
            calc(53);
            calc(30);
      }
}
```

Renamed

```
public class GradeCalculator {

	public static void calculateAndPrintGrade(int finalMark){
		if (finalMark >= 70){
			System.out.println("grade = A");
			return;
		}
		if (finalMark >= 60){
			System.out.println("grade = B");
			return;
		}
		if (finalMark >= 50){
			System.out.println("grade = C");
			return;
		}
		if (finalMark >= 40){
			System.out.println("grade = D");
			return;
		}
		if (finalMark<40) System.out.println("grade = F");
	}

	public static void main(String[] args) {
		calculateAndPrintGrade(90);
		calculateAndPrintGrade(53);
		calculateAndPrintGrade(30);
	}
}
```

Appendix C: The `List` interface

You will note that in the classes given, `ArrayLists` are of type `List`.

`List` is an interface: http://docs.oracle.com/javase/7/docs/api/java/util/List.html

Explanation below, modified from Effective Java, 2nd edition by Joshua Bloch:

> *You should use interfaces rather than classes as parameter types. More generally, you should favour the use of interfaces rather than classes to refer to objects. If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.*

Get in the habit of typing this:

```
// Good - uses interface as type
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

rather than this:

```
// Bad - uses class as type!
ArrayList<Subscriber> subscribers = new ArrayList<Subscriber>();
```

If you get into the habit of using interfaces as types, your program will be much more flexible. If you decide that you want to switch implementations, all you have to do is change the class name.

For example, the first declaration could be changed to read:
`List<Subscriber> subscribers = new LinkedList<Subscriber>();`
and all of the surrounding code would continue to work. The surrounding code was unaware of the old implementation type, so it would be oblivious to the change.

Marks for CO2220 coursework assignment 2

The marks for each section of coursework assignment 2 are clearly displayed against each question and add up to 96. There are another two marks available for giving in uncompressed files and for giving in files that are not contained in a directory. There are two marks for giving in files with the correct names. This amounts to 100 marks altogether. Another 100 marks were available from coursework assignment 1.

Total marks for part A	[69 marks]
Total marks for part B	[27 marks]
Mark for giving in uncompressed files	[1 mark]
Mark for giving in standalone files; namely, files **not** enclosed in a directory	[1 mark]
Mark for giving in files with the correct names	[2 marks]
Total marks for coursework assignment 2	**[100 marks]**

[END OF COURSEWORK ASSIGNMENT 2]