

# Informatics 1: Object Oriented Programming

## Assignment - Part III

The University of Edinburgh

Volker Seeker (volker.seeker@ed.ac.uk)

## 1 Introduction

For the INF1B course you are expected to complete an assignment which consists in three parts. Once all three parts are completed by the end of the semester, your final course mark will be determined based on your submissions.

This is the third part of the assignment (PART III). Its aim is to assess your understanding of the concepts taught in this course. This includes your ability to ...

- ... write functionally correct code using a test driven approach and specific instructions.
- ... understand and extend code written by someone else.
- ... write code of high quality with a focus on documentation, readability and structure.
- ... correctly use Object Oriented programming techniques and library classes in the Java language.

### 1.1 Administrative Information

#### Deadline

The assignment is due at **16:00 on 3 April 2020**.


#### Assessment and Feedback

PART III of the assignment is marked for credit after the submission deadline. You can review the course's specific marking criteria in Appendix B. Make sure you demonstrate a good understanding of the individual marking criteria listed there.

Like for PART I, the CODEGRADE auto test system is used and you can **and should!** submit your code as often as possible for continuous feedback on your progress. Remember though that the auto tests only show your progress on the *Correctness and Robustness* aspect. Although important, it is not the only one you need to pay attention to.

From PART II you should now have a better understanding of how you should document and structure your code as well as make use of Java language features and library classes to improve its overall quality.

The marking is done by human markers who follow the marking scheme as specified in Appendix B. They will make use of the auto test results in a first instance but if in doubt will also look at your code directly.


 You will receive a report document with individual feedback on your submission once the marking is done. This will include overall results and a brief description of all executed tests. You will, however, not receive a sample solution or access to the advanced test sources for this part of the assignment.

## Coding

For every coding task in the assignment, you are not only expected to write functionally correct code but also maintain high code quality in terms of readability, robustness, documentation, structure and use of the Java language.

For this part of the assignment you are free to use most aspects of the Java language and API library with a few exceptions:

- You are allowed to implement classes in addition to the required ones, however, you must not change the provided classes unless you are specifically told to do so, like for `LibraryFileLoader.java`, `BookEntry.java` and `CommandFactory.java`.
- Do not use any third party libraries, only the Java API classes.
- Do not use any functional language constructs such as lambdas or streams (that was last semester).
- Keep all classes in the default package, i.e. no package. This will allow the automatic tests to run through smoothly.

 With this amount of files, it is definitely a good idea to subdivide them into individual packages but we have to prioritise a smooth marking process for a large amount of students over best software engineering practices here. So please keep them all in the default package. **Please attend the introductory lecture for this assignment for a suggestion on how to best setup your IntelliJ project.**

## Submission

The submission should happen via the CODEGRADE infrastructure. You can find more information on how to use it and what to submit in Appendix A.

Before the deadline, you can **and should!!** submit as often as possible for continuous feedback on your progress. Your latest submission will then always be replaced with the newest version. All auto tests will run for your final submission after the deadline.

Deadline extensions are possible for PART III of the assignment, if you have a very good reason to ask for one. The School of Informatics has a [policy on coursework deadlines](http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests)<sup>1</sup>, which applies across all taught courses. Please make sure you fully understand it before you apply for an extension.

---

<sup>1</sup><http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests>

## Support

Since PART III is marked for credit, you should **strictly** adhere to the *Good Scholarly Practice* (see below) when implementing your solutions. Do not share your code with other students and come up with all solutions on your own.

Nevertheless, you will have the possibility to get support should you need it. At any time, you are welcome to discuss the assignment with any course instructor. You can do that during teaching sessions or via the course's [Piazza forum](#)<sup>2</sup>.

When posting on Piazza, please put all questions into the corresponding assignment folder. Feel free to discuss general techniques amongst each other unless you would reveal an answer. If in doubt about revealing an answer, please ask the instructors privately.

Lastly, as before please make good use of any internet or literature resource you consider helpful for solving the tasks (that, of course, includes all the course material released so far).

## Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

<http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

---

<sup>2</sup><https://piazza.com/ed.ac.uk/spring2020/inf1b>

## 2 Content

This section will give you an overview over the content of the actual tasks. You will build a library for a book data set taken from the [goodreads website](https://www.goodreads.com/)<sup>3</sup>. Next to functionality of loading books into the library, you will implement management features such as displaying, removing or searching for individual book entries. Listing 1 shows an example of the final software.

```
1 Enter a library command or type HELP for command overview.
2 > ADD books.csv
3 5 new book entries added.
4 > LIST
5 5 books in library:
6 The Changeling
7 The Vampire Companion
8 Harry Potter and the Goblet of Fire (Harry Potter #4)
9 Gunslinger and Nine Other Action-Packed Stories of the Wild West
10 Animal Farm
11 > REMOVE TITLE Animal Farm
12 Animal Farm: removed successfully.
13 > EXIT
```

Listing 1: Book Library and Management Functionality


### 2.1 Template Material

**Skeleton Classes** To get you started with the implementation, some part of the library has already been implemented for you. Specifically, all of the user interaction via the command prompt already exists as well as parts of the infrastructure for loading book data and executing commands. Your job is to fill in some missing links and implement most of the management commands.

**JUnit Tests** Similar to lab exercises, you are provided with a set of JUnit tests which will help you check the functional correctness of your implementations. We highly recommend that you follow a regression testing approach by testing your code vigorously after each new change.

Be aware, that the provided tests do not offer complete functional coverage of all possible inputs to your program. The automatic tester will use the provided tests as well as additional advanced tests for checking corner cases and specific scenarios you should think of on your own. For that purpose, feel free to extend the provided tests.

**Book Library Files** You are given a set of library data files with sample entries for books. One of them is very long with over ten thousand entries. They have been cleaned up and cover all cases you can expect as data input to your program.

 You should take some time to familiarise yourself with the functionality and source code that is already there. You will certainly find that helpful for solving the tasks given below.

### 2.2 Book Library Usage

As described above, some functionality is already included:

---

<sup>3</sup><https://www.goodreads.com/>

- The program can be started by executing the main function in the Main class. It does not require command line arguments.
- You can control the library by putting in specific commands as soon as you see the command prompt appearing in the console indicated by the > sign.
- A command is always an upper case keyword followed by zero or more command arguments, depending on its specification.
- The HELP and EXIT commands are already fully functional. You can try them out and look at their implementation for details on how they work<sup>4</sup>.

### 3 Tasks

The following section contains all tasks you should complete for the assignment.

Some parts are marked as **ADVANCED** which means they are usually more difficult to solve but not necessarily required for a passing solution. You might find that some self study of Java language features and API classes is required to tackle them.

#### Task 1 - Implement a BookEntry class

◀ Task

Your very first job is to implement a class which encapsulates all the data required for a single book entry within the database. A class dummy of the BookEntry class already exists and you should add your own implementations there. The BookEntry class is meant to be immutable, so **it should not be allowed to manipulate any data** an instance of this class contains after it was created.

Complete the following steps for your implementation of the BookEntry class:

**Instance Fields** A book entry should contain the following data: the title of the book, the author or authors of the book, a rating between 0 and 5, an ISBN and the number of pages. You should create five immutable and private instance fields to represent that: a String **title**, a String array **authors**, a float **rating**, a String **ISBN** and an int **pages**.

**Constructor** The BookEntry class should have a single constructor which takes five parameters and initialises the corresponding fields. The order and type of constructor parameters should be the same as specified above. **You should also make sure that the constructor checks the legality of provided arguments in the following way: None of the object parameters must be null, the rating must be between 0 and 5 and the given pages must not be negative. If any of those specifications are violated, you should throw a corresponding exception as specified in the INF1B Coding Conventions.**

**Getters** Write individual getters for all five instance fields.

**ToString** Override the toString method of the Object superclass for your BookEntry. The format of the generated String should be the following:

```
1 <title>
2 by <author 1>, <author 2>
3 Rating: <rating>
4 ISBN: <ISBN>
5 <pages> pages
```

<sup>4</sup>The implementation for library commands follows a variant of the **Command programming pattern**<sup>5</sup>

for example:

```
1 Selected Writings
2 by Thomas Aquinas, Ralph McInerny
3 Rating: 3.90
4 ISBN: 140436324
5 841 pages
```

You can always use the plural word for pages even if there should be only one. Make sure the rating is formatted with two decimal places.

**Equals and hashCode** In order to allow state based comparisons of pairs of `BookEntry`s, you now need to override the `equals` and `hashCode` method of the `Object` superclass. The `equals` method should do some initial instance checking first and then compare all five instance fields with the given parameter. Only when all of them are equal, you should return `true`. The `hashCode` method needs to generate a hash code from all five instance fields.

## Task 2 - Add books to the library


◀ Task

Now that a `BookEntry` class exists, you can populate your database with data from a given csv file. To do that, you will implement the `ADD` command and fill in the missing bits in the `LibraryFileLoader`.

### Command Workflow

Each command in this application has a corresponding type entry in the `CommandType` enum and extends the class `LibraryCommand`. The life cycle of handling user input to execute a command has multiple steps:

- In the `LibraryBrowser`, the user input is read from the command prompt and passed to an instance of the `CommandInterpreter` to create a corresponding command and execute it.
- The `CommandInterpreter` identifies the leading command keyword, maps it to the corresponding `CommandType` and splits it from the preceding command arguments.
- Both type and argument `String` are passed to the `CommandFactory` which will call the constructor of the corresponding specific implementation.
- The constructor of each specific command class passes both type and argument to its super class `LibraryCommand` (see `HelpCmd` or `ExitCmd` for an example), which in turn calls the `parseArgument` method.
- The default implementation of `parseArgument` expects a blank argument which is sufficient for `Help` and `Exit`. Your own commands need to override this method in the specific command subclass to adapt for individual argument parsing.
- After the constructors have been executed and the argument `String` was parsed as part of that process, the newly created command instance is passed to the `CommandInterpreter` together with the internal book library (`LibraryData`). There, its `execute` method is called.
- The `execute` method is abstract in `LibraryCommand`, so each specific subclass needs to override it with corresponding functionality.

 Note that `parseArgument` should only parse the actual argument and remember necessary parameters in instance fields. The actual execution should happen in the `execute` method.

## The ADD Command

This command should allow the user to add additional books to the library from a book data csv file. The command syntax is as follows:

```
1 ADD path/to/book/data.csv
```

Listing 2: ADD command syntax

The ADD command is followed by a command argument which is the path to a book data csv file. This path could include subfolders or only a file name. Consider the following examples:

```
1 Enter a library command or type HELP for command overview.
2 > ADD books01.csv
3 24 new book entries added.
4 > ADD ../data/books04.csv
5 3 new book entries added.
6 > ADD booksDoesNotExist.csv
7 ERROR: Reading file content failed: java.nio.file.NoSuchFileException: booksDoesNotExist.csv
8 ERROR: Loading book data failed for file: booksDoesNotExist.csv
9 > ADD incorrectFormat.txt
10 ERROR: Invalid argument for ADD command: incorrectFormat.txt
11 ERROR: Given command input is invalid: ADD incorrectFormat.txt
```

Listing 3: Sample usage of the ADD command

Most of the file loading and adding books to the `LibraryData` class has already been implemented for you. Complete the following steps for your implementation of the `AddCmd` class:

- Create a new class `AddCmd` which extends the `LibraryCommand` class. To achieve compilation, you need to provide a dummy `execute` method for now.
- Provide a constructor which has a single `String` parameter for the argument input. The constructor needs to call `LibraryCommand`'s constructor with the required arguments.
- **Override** `LibraryCommand`'s `parseArguments` method. In this method you should parse the given `String` argument and remember it for later use in the `execute` method. You can do this by creating an instance field in the `AddCmd` class and assigning your parsed book data path to it.

If the argument is a valid path that indicates a file name which ends with `.csv`, you should return `true`. If not, you should return `false`. Validity here is independent of whether the file exists or not. Remember, argument parsing is not concerned with executing the actual command. That means, pretty much everything is allowed as long as the file ending is correct.

- Now implement the `execute` method. Make sure the given `LibraryData` is not null and throw the necessary exception if so. The actual loading of the book data should be a simple call of the given `LibraryData` instance's `loadData` method with the path you have remembered in your `parseArgument` method.

## Parsing Book Data

The loadData method will create a LibraryFileLoader which loads the book data from file, saves each line in the list fileContent and calls the parseFileContent method. The parseFileContent method already exists as a dummy function. Its job is to **iterate through each entry of the fileContent instance field and create a BookEntry for each of them. All BookEntries you create, should be added to an ArrayList** which is returned as a result of the method.

If no content has been loaded before the parseFileContent method was called (**that means the fileContent field will be null**), **you should return an empty list and print the following error message: ERROR: No content loaded before parsing.**

A single line in a book data file is a comma separated list of data values with the following format:

```
1 <title>,<Author1-Author2-Author3>,<Rating>,<ISBN>,<PAGES>
```

Listing 4: Format of individual lines in book entry data files

Authors can have single or multiple entries which are again separated by hyphens. Consider the following examples and look at the given data files for more:

```
1 The Changeling,Zilpha Keatley Snyder,4.17,595321801,228
2 Journey of the Sparrows,Fran Leeper Buss-Daisy Cubias,3.67,142302090,160
```

Listing 5: Sample entries of book data files

You can make the following assumptions to simplify parsing of file content:

- The first line of each file is a column header and not actual data.
- Within individual files no duplicate book entry exists.
- Book entry lines are always well-formed as specified in Listing 4. So you do not need to handle potential formatting errors.
- There are no empty lines in book entry data files.

## Task 3 - Display the contents of the library

◀ Task

This command should allow the user to display all books currently loaded into the library in different formats. The command syntax is as follows:

```
1 LIST [short|long]
```

Listing 6: LIST command syntax

The LIST command is followed by a command argument which is either **long** or **short** for a long and extensive displaying of books or a brief listing of only titles. It should also be possible to give no argument at all which would automatically use a **short** display method by default. Consider the following examples:



```

1 Enter a library command or type HELP for command overview.
2 > ADD books04.csv
3 3 new book entries added.
4 > LIST short
5 3 books in library:
6 The Jew of Malta
7 Selected Writings
8 The Fixer
9 > LIST long
10 3 books in library:
11 The Jew of Malta
12 by Christopher Marlowe, H. Havelock Ellis
13 Rating: 3.61
14 ISBN: 486431843
15 80 pages
16
17 Selected Writings
18 by Thomas Aquinas, Ralph McInerny
19 Rating: 3.90
20 ISBN: 140436324
21 841 pages
22
23 The Fixer
24 by Bernard Malamud, Jonathan Safran Foer
25 Rating: 3.95
26 ISBN: 374529388
27 335 pages
28
29 > LIST
30 3 books in library:
31 The Jew of Malta
32 Selected Writings
33 The Fixer

```

Listing 7: Sample usage of the LIST command

Complete the following steps for your implementation of the `ListCmd` class:

- Create a new class `ListCmd` which extends the `LibraryCommand` class. To achieve compilation, you need to provide a dummy `execute` method for now.
- Provide a constructor which has a single `String` parameter for the argument input. The constructor needs to call `LibraryCommand`'s constructor with the required arguments.
- Override `LibraryCommand`'s `parseArguments` method. In this method you should parse the given `String` argument and again remember it for later use in the `execute` method. The given argument could be either **short, long or entirely blank**. Return `true` if you could parse a valid argument successfully and `false` otherwise.
- Now implement the `execute` method. **Make sure the given `LibraryData` is not null** and throw the necessary exception if so. You can get the list of currently loaded books from the `LibraryData` parameter of the `execute` method. Iterate through and print each book entry to the console.

If the user selected **short** print (or default by giving a blank argument), you should only print each **book's title**. If the user selected **long** print, you should print **all information in the format specified for `BookEntries toString` method**. Each **long** entry should be followed by an empty line.

Before you print individual entries in the required format, you should always start with a header displaying the message: <book count> books in library:

If the library contains no books and the LIST command is called, you should print the following message: The library has no book entries.

## Task 4 - Search for specific books

◀ Task

This command should allow the user to search for specific books within the library and display them. The command syntax is as follows:

```
1 SEARCH <value>
```

Listing 8: SEARCH command syntax

The SEARCH command is followed by a command argument which is a specific search value. Search values have to be single words only. Upper and lower cases in search words should be irrelevant. The search only considers book titles. Consider the following examples:

```
1 Enter a library command or type HELP for command overview.
2 > ADD books01.csv
3 24 new book entries added.
4 > SEARCH Fire
5 Harry Potter and the Goblet of Fire (Harry Potter #4)
6 > SEARCH Harry
7 Harry Potter and the Goblet of Fire (Harry Potter #4)
8 Harry Potter and the Chamber of Secrets (Harry Potter #2)
9 Harry Potter and the Philosophers Stone (Harry Potter #1)
10 > SEARCH firE
11 Harry Potter and the Goblet of Fire (Harry Potter #4)
12 > SEARCH notinlibrary
13 No hits found for search term: notinlibrary
```

Listing 9: Sample usage of the SEARCH command

Complete the following steps for your implementation of the SearchCmd class:

- Create a class as for previous commands and provide the necessary constructor.
- Override LibraryCommand's parseArguments method, parse the given argument and remember what you need for later. Return true if the argument is valid and false otherwise. Remember, search values have to be single words (no white space in between) and must not be entirely blank.
- Implement the execute method and perform appropriate error checking for parameters. Search through the titles of all books within the library and check if any of the titles match your search parameter. Print the titles of all books that contain the search value to the command line in the order you found them.

If you did not find any entries that matched the given search value, you should print the message:  
No hits found for search term: <value>

## Task 5 - ADVANCED Remove books from the library

◀ Task

This command should allow the user to remove specific books from the library. The command syntax is as follows:

```
1 REMOVE TITLE|AUTHOR <value>
```

Listing 10: REMOVE command syntax

The REMOVE command is followed by a two part command argument. The first part indicates which parameter should be considered for removal. This can either be **AUTHOR** or **TITLE**. The second parameter indicates a value which is either a full title or full author name. Consider the following examples:

```
1 Enter a library command or type HELP for command overview.
2 > ADD books01.csv
3 24 new book entries added.
4 > REMOVE TITLE Animal Farm
5 Animal Farm: removed successfully.
6 > REMOVE AUTHOR J.K. Rowling
7 3 books removed for author: J.K. Rowling
8 > REMOVE TITLE not in library
9 not in library: not found.
10 > REMOVE AUTHOR not an author
11 0 books removed for author: not an author
```

Listing 11: Sample usage of the REMOVE command

Complete the following steps for your implementation of the RemoveCmd class:

- Create a class as for previous commands and provide the necessary constructor.
- Override LibraryCommand's parseArguments method, parse the given argument and remember what you need for later. Return true if the argument is valid and false otherwise. Remember, a valid argument is either **AUTHOR** or **TITLE** followed by a string value. Unlike for the SEARCH command, this value can contain white spaces. It must not be entirely blank though.
- Implement the execute method and perform appropriate error checking for parameters. Evaluate if you are removing by title or author, go through all book entries in the library and remove any matches with the given remove value you find.

If you are removing titles, you can assume that each title appears only once within the library. Print a confirmation afterwards: <remove value>: removed successfully.

If you are removing authors, you should count how many titles you removed for this author and print it as part of the confirming message: <count> books removed for author: <remove value>

If you did not find any entries that matched a given title or author, you should print the messages: <remove value>: not found. and 0 books removed for author: <remove value>, respectively.

## Task 6 - ADVANCED Group books by title or author

◀ Task

This command is a variant of the LIST command and should allow the user to display book entries in specific groups. The command syntax is as follows:

```
1 GROUP TITLE|AUTHOR
```

Listing 12: GROUP command syntax

The GROUP command is followed by a command argument which specifies either **TITLE** or **AUTHOR**. The former groups all library entries by the starting letter of the title while the latter groups all entries by full author names. Consider the following examples:

```

1 Enter a library command or type HELP for command overview.
2 > ADD books05.csv
3 8 new book entries added.
4 > GROUP TITLE
5 Grouped data by TITLE
6 ## E
7     eBay for Dummies
8 ## H
9     Harry Potter and the Goblet of Fire (Harry Potter #4)
10    Harry Potter and the Chamber of Secrets (Harry Potter #2)
11    How the Mind Works
12 ## J
13    Journey of the Sparrows
14 ## O
15    On Duties
16 ## P
17    Paths to God: Living the Bhagavad Gita
18 ## [0-9]
19    1984
20 > GROUP AUTHOR
21 Grouped data by AUTHOR
22 ## Daisy Cubias
23    Journey of the Sparrows
24 ## E.M. Atkins
25    On Duties
26 ## Erich Fromm
27    1984
28 ## Fran Leeper Buss
29    Journey of the Sparrows
30 ## George Orwell
31    1984
32 ## J.K. Rowling
33    Harry Potter and the Goblet of Fire (Harry Potter #4)
34    Harry Potter and the Chamber of Secrets (Harry Potter #2)
35 ## Marcus Tullius Cicero
36    On Duties
37 ## Marsha Collier
38    eBay for Dummies
39 ## Miriam T. Griffin
40    On Duties
41 ## Ram Dass
42    Paths to God: Living the Bhagavad Gita
43 ## Richard Alpert
44    Paths to God: Living the Bhagavad Gita
45 ## Steven Pinker
46    How the Mind Works
47 ## Walter Cronkite
48    1984

```

Listing 13: Sample usage of the GROUP command

Complete the following steps for your implementation of the GroupCmd class:

- Create a class as for previous commands and provide the necessary constructor.
- Override LibraryCommand's parseArguments method, parse the given argument and remember what

you need for later. Return true if the argument is valid and false otherwise. Remember, a valid argument is either **AUTHOR** or **TITLE**.

- Implement the `execute` method and perform appropriate error checking for parameters. Evaluate if you are grouping by title or author and print the output for all book entries in the library as required.

You should start the group output with the following header: Grouped data by <group parameter>

Groups for titles should be single upper case letters prefixed with a double hash symbol (`##`). It should not matter for titles if the first letter of the title is upper or lower case when assigning to individual groups, e.g. do not make two groups for lower case 'e' and upper case 'E' they should both go into the same group.

You can assume that titles start with either a letter [A-Za-z] or a digit [0-9]. If a title starts with a digit, you should add it to a single digit group which is called: `## [0-9]`

Author group names should contain the full author name. If there are multiple authors for a single title, make sure you print the title for all of them.

Group names should be ordered lexicographically, while the ordering within groups does not matter.

If the library is empty, you should print the message: `The library has no book entries.`

## Appendix A Submission

We are using an online tool called CODEGRADE to automatically grade your code for correctness and robustness. You can access this tool via the INF1B [Learn page](#)<sup>6</sup> by following the menu:

[Assessment] > [Assignment Part III] > [Assignment Part III - Submission].

This will open the CODEGRADE web interface for you within the Blackboard environment. Here you have the options to review a previous submission and the feedback you received so far, hand in a new submission (which will override the previous one) or see the grading rubrics for this auto test.

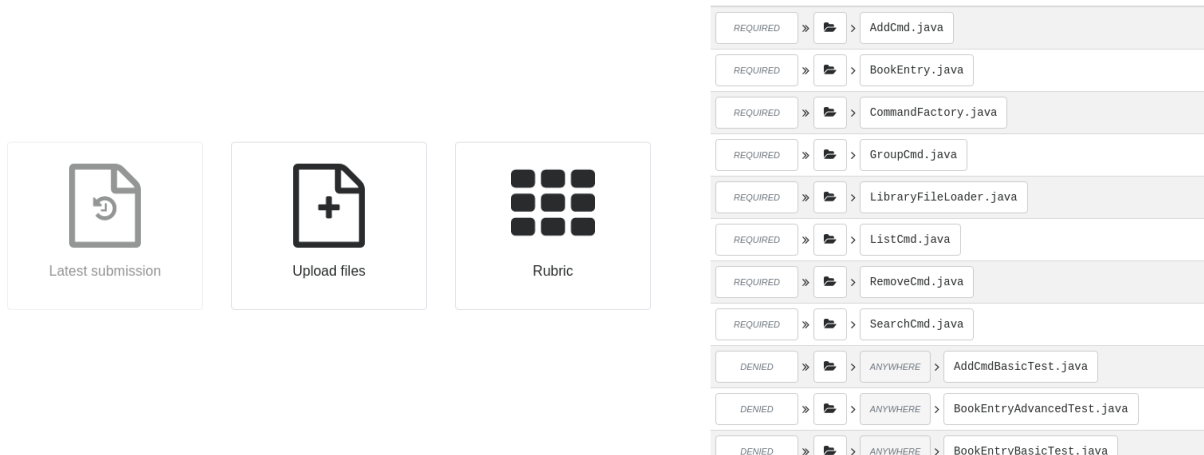


Figure 1: CODEGRADE menu for submission and hand-in instructions.

You are required to hand in the files you are supposed to implement or modify according to the specifications in the tasks above. Those are:

- AddCmd.java
- BookEntry.java
- CommandFactory.java
- GroupCmd.java
- LibraryFileLoader.java
- ListCmd.java
- RemoveCmd.java
- SearchCmd.java


You are also allowed to submit anything else you like. However, you should make sure that java files you intend to be included in the marking process are in the default package and in the root folder.


To make submission a bit easier, all files which were provided to you with the templates and should not be modified are automatically denied by the submission system. When you attempt to submit one of those CODEGRADE can remove them for you automatically and hand in the rest. You can submit individual files or archives such as zip or tar.


---

<sup>6</sup><https://course.inf.ed.ac.uk/inf1b>

You can find a more detailed description on the [CODEGRADE website](#)<sup>7</sup>. We are not using Git or Group submissions for our assignments.

 One of the initial criteria checked for your submission is that all files compile together with the provided basic unit tests. Make sure this step runs through. If you did not manage to solve all exercises, provide required function dummies to enable compilation.

 We recommend that you regularly submit your files for continuous feedback on how well you are doing.

 Keep in mind that this auto grader only looks at one category considered for the final marking process which is *Correctness and Robustness*. Although this is an important part, it is not the only aspect we consider when putting your final course mark together. Have a look at the marking criteria in Appendix B for more details.

## Appendix B Inf1B Marking Criteria

This section contains a description of the assessment scheme used to determine the overall course mark for each student at the end of the semester. This happens after you have submitted the third part of your assignment at the end of week 11. For this process we will consider the feedback you generated for PART II and your code submission for PART III.

We will consider the following criteria:

**Completion** Those with less previous experience may have difficulty completing all of the assignment tasks. It is possible to pass without attempting the more advanced tasks - and a good solution to some of the tasks is much better than a poor solution to all of them.

**Readability & Code Structure** Code is a language for expressing your ideas - both to the computer, and to other humans. Code which is not clear to read is not useful, so this is an essential requirement.

**Correctness & Robustness** Good code will produce “correct” results, for all meaningful input. But, it will also behave reasonably when it receives unexpected input.

**Use of the Java Language** Appropriate use of specific features of the Java language will make the code more readable and robust. This includes, for example: iterators, container classes, enum types, etc. But the structure of the code, including the control flow, and the choice of methods, is equally important.

**Code Review** Working with code provided by other developers is a major part of working with software. The ability to assess a piece of code written by someone else and to provide meaningful feedback on its quality is therefore an essential skill you should learn.

Marks will be assigned according to the [University Common Marking Scheme](#)<sup>8</sup>.

<sup>7</sup><https://docs.codegra.de/guides/use-codegrade-as-a-student.html>

<sup>8</sup><https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme>

The following table shows the requirements for each grade. All of the requirements specified for each grade must normally be satisfied in order to obtain that grade.

**Pass: 40-49%**

- Submit some plausible code for a significant part of assignment PART III, even if it does not work correctly.
- Demonstrate some understanding of the issues involved in your answers to at least one question in the Code Review from PART II.

**Good: 50-59%**

- Submit working code for most parts of assignment PART III, even if there are small bugs or omissions.
- Submit code which is sufficiently well-structured and documented to be comprehensible. This includes an appropriate use of Java features and choice of methods, as well as layout, comments and naming.
- Demonstrate some understanding of the issues involved in your answer to all of the questions in the Code Review from PART II.

**Very Good: 60-69%**

- Submit working code for all parts of assignment PART III, even if this contains small bugs.
- Submit code which is well-structured and documented.
- Demonstrate good understanding of the issues involved in your answers to all of the questions in the Code Review from PART II and provide plausible and actionable solutions for some.

**Excellent: 70-79%**

- Submit and demonstrate working code for all parts of assignment PART III, with no significant bugs.
- Submit code which is well-structured and documented.
- Demonstrate good understanding of the issues involved in your answers to all of the questions in the Code Review from PART II and provide plausible and actionable solutions for all of them.

**Excellent: 80-89%**

- Marks in this range are uncommon. This requires faultless, professional-quality design and implementation, in addition to well-reasoned and presented answers to the Code Review questions.

**Outstanding: 90-100%**

- Marks in this range are exceptionally rare. This requires work “well beyond that expected”.