Object-Oriented Programming
Software Workshop 1

Assignment 3:
Maps

Set by
Jacqueline Chetty
Ana Stroescu

Early Testing Deadline:
Wednesday 8th December 2021 2pm GMT

Final Submission Deadline:
Monday 13th December 2021 2pm GMT

Use Canvas for the definitive deadlines

This assignment is worth
35 % (**thirty-five** percent)
of the overall course mark

You must use **OpenJDK 11**

Overview

Table of contents

List of figures

1   Introduction

The Colossal Adventure Game is a text-based game that was developed in the 1970s. In the game, the player controls a character through simple text commands to explore a cave rumored to be filled with wealth. Players earn predetermined points for acquiring treasure and escaping the cave alive, with the goal to earn the maximum number of points offered.

You must complete the implementation of a version of this game where a player can type the direction that they would like to go in and the application must respond accordingly. For example, a player can request "I want to go north" and the application must determine whether they can go North — based on direction / location .txt files. If they are able to go North then the application should allow the player to go North, then list the next set of directions (North, North East, West, etc) that are available to the player to navigate. This mapping from one direction to the next, depending on where the player wants to go, continues until the player quits the game, reaches a dead end and cannot return, or locates the pot of gold.

If you want to consider the game as follows: there is a house with a variety of rooms. If you are in the kitchen you may then be able to go north to the dining room or south to the lounge. You may be able to also go west to the bathroom. If a player chooses to go north (from the kitchen) to the dining room they can then go south again and return to the kitchen or perhaps navigate east to a bedroom.

The beauty about the development of such a game in the 70s was the player's ability to type a variety of text, where the application "picks up" only the direction and ignores the other part of the phrase. This was considered to be very sophisticated "in its day."

Descriptions of the overall requirements are given in §4 Coding tasks and §5.1 Code requirements, but specifics of these requirements are generally left to 'TODO' comments in the skeleton code (use IntelliJ's TODO tab). You also need to analyse differences between your version's output against a set of ExpectedOutput files provided in the assignment pack.

2   Important points

Starting your assignment by typing code will go wrong and waste a lot of time. We recommend the following approach:

1. read this document quickly to see the big picture;
2. load the skeleton code into IntelliJ and use these plugins:

    Call Graph (Chentai Kao) to visualise which methods call each other,

    SequenceDiagram (VanStudio) for a different visualisation of methods
      calling each other,

    UMLGenerator (Alessandro Caldonazzi) to visualise class relationships;

3. study those diagrams even though they are of incomplete code and think what extra connections and sequences your completed code will probably need based on the assignment requirements;

4. study and understand the names of methods and constants and decide when and how the author intended them to be used;

5. analyse the relationship between the provided input files and their corresponding expected output, as this helps you decide and design the steps needed to convert input to output;

6. study IntelliJ's TODO tab and relate the list to the assignment requirements;

7. re-read this assignment document in more detail for the requirements and relate the information to the TODO list, the expected outputs (from their respective inputs), and the diagrams;

8. put multiple designs on paper and consider which parts of which designs are more (or less) suitable — be creative here and consider attempting tasks 'backwards' or from an opposing perspective;

9. transform parts of your design into Java code, probably starting with something from the final stages (for example printing some calculated output) and working forwards (for example towards the raw inputs needed to produce that calculated output);

10. use an end-to-end approach and make something work from beginning to end for a simple case before making your code more flexible, more robust, more efficient, or more elegant;

11. re-test your code after every two or three lines you write or change: baby steps are fastest overall.

But first of all, please take a moment to remind yourself of the rules §B Rules and §D Submission instructions as well as the School's and the University's requirements for good academic practice. In particular:

1. you are encouraged to discuss the design of your program but **you must not share actual program code** otherwise you risk plagiarism charges;

2. you are expected to **work out for yourself** the details of what your code must do from analysing this document, the skeleton code, and especially any ExpectedOutput files we give you;

3. should there be any apparent contradictions between this document and either the skeleton code or ExpectedOutput, then **specifications in the skeleton code and output in the ExpectedOutput take precedence**;

4. **your code must compile** as submitted otherwise you will score zero, no matter how trivial a change is needed to make it compile;

5. **your code must run on our system** — whether it runs on yours does not count.

3   Marking scheme

Question	Description	Marks
1	Mapping	25
2	Location	25
3	LocationMap	25
4	FileLogger	20
5	ConsoleLogger	5
	Total	100

The marks listed are for submissions which correctly do everything required and which produce output exactly matching the expected output. It is not feasible to specify the exact break-down of marks within each item in the table and we keep some of the tests secret until after the deadline to hamper attempts to game the system.

Generally speaking, if your submissions meets all the requirements in §4 Coding tasks and §5.1 Code requirements in the **must**, **must NOT** and **Functionality** lists, then it will probably obtain a reasonable pass mark — deductions for late penalties and rule-breaking notwithstanding.

Adding functionality not specified in the assignment can lose you marks — especially if it prints anything not required by this assignment — and might be interpreted as an attempt to cheat if it looks like you used code originally written for a different program.

4   Coding tasks

This section details the coding tasks. You do not have to attempt the tasks in the order listed. Nor is it necessary to finish one task before starting another. Cycling between tasks is expected and indeed encouraged.

Each subsection below says what a particular part of the code **must** or **must not** or **should** or **should not** do. The difference matters. Functionality defined by **must (not)** are **core requirements**: the minimum a client paying you for

this software would accept. Successfully meeting only these requirements will probably limit you to a reasonable overall mark, penalties notwithstanding. Higher marks can be unlocked by fulfilling the lists of **should (not)**.

Breaking requirements marked with a dagger[†] could result in a zero score for the entire assignment. Breaking any requirement marked with a double dagger[‡] will definitely result in a zero score for the entire assignment.

4.1 Overall functional requirements

Additional to the requirements below, if your program is a mixture of 'advanced' code with basic bits that are badly programmed, we reserve the right to question you on the 'advanced' parts and deduct marks — possibly all of them — if we are not satisfied by your ability to explain them. This is to discourage you from just copying or adapting code simply for the sake of scoring marks: it is far, far more important to your academic development that you understand what you are doing, even if your code is currently clumsy. If you do not learn to understand programming at this stage, you will struggle even more with subsequent modules.

Overall, your program **must**:
1. [†] only ever use **one** `Scanner` to read from the keyboard;
2. follow the requirements (positive and negative) set out in this document;
3. account for any official changes to the assignment after the assignment has been released.

Overall, your program **must NOT**:
1. [‡] `import` any other libraries apart from the ones that are allowed;
2. [‡] change any signatures of classes, class fields, or methods (see Item 3, page 16, and subsequent explanation, page 17);
3. [‡] remove any required methods;
4. [‡] remove any classes;
5. [‡] add any new classes;
6. [‡] add any new methods;
7. [†] crash when given well-formed correct input;
8. [†] print any extra output beyond what is officially specified;
9. [†] add any extra functionality beyond what is officially specified;
10. [†] create any new `Scanner`s.

Overall, your program **should**:
1. [†] only `import` items that are explicitly included in the skeleton;
2. [†] work correctly if the **content** of items in `Defaults.java` are changed;
3. exhibit consistent and predictable behaviour;

4. tolerate some mildly inappropriate input such as extraneous white space or an `int` outside the required range;

5. be efficient in terms of execution speed[1] mainly by not performing unnecessary steps or calculations, though this is less important than the program working correctly.

With respect to Item 1, a definitive list of allowed `import`s will be maintained on the Canvas assignment page. The **only** permitted way to seek adjustments to this list is to email the shared mailbox for the course with a compelling justification for your request. Emailing someone directly or asking them in person is not allowed because there are serious implications to the viability of the assignment if `import` restrictions are changed, hence these must be considered by multiple people involved with different aspects of the assignment. Please expect in advance that your request is likely to be denied.

Overall, your program **should NOT**:

1. ‡ crash when given badly formatted input of the **correct** type, such as multiple spaces between `int`s when there should be only a single space;

2. ‡ crash if input is of the correct type but inappropriate content, such as an `int` out of range or a `String` containing leading or trailing white space.

> We will test all your classes individually and then substitute them in some, probably most, tests with our own versions. This is to try to avoid penalising you multiple times for the same error. But it means your code must work within the boundaries set by the assignment and not have any additional, unwarranted functionality. This also means you absolutely must not add or remove any methods or class fields to these classes, nor modify the signatures of any provided methods and fields. If your program cannot function without all but one of the classes substituted at any one time, then you will probably score zero overall.

---

[1] We test this by limiting the amount of time your code program is allowed to run for.

The following sections describe each class that forms part of the Colossal Game application:

4.2 `Mapping`

The `Mapping` class creates a structure based on directions, such as north, south, east, west, etc. The primary function is to determine where the player is, determine exits and allow the player to move to another direction.

Functionality:

1. The skeleton code provides a constant that represents the initial location
   at the start of the game - DO NOT change it;
2. Creates a static `LocationMap` object;
3. Creates a vocabulary `HashMap<>`;
4. Creates suitable objects for the `FileLogger` and `ConsoleLogger`;
5. Provide suitable directions within the constructor;
6. Create a `Scanner`;
7. Within a loop:
   - 7.1 finds a player's location;
   - 7.2 determines whether any exits exist;
   - 7.3 maps the next chosen direction or print a suitable message that a
     player cannot proceed in that direction;
8. Creates method `main()`.

4.3 `Location`

The `Location` class provides a data structure for the location id and the description. For example, `LocationId` and description are linked as seen in
`locations.txt`, where `LocationId` is a number such as 0, and description lists where the player is located.

Functionality:

1. A constructor that accepts parameters and creates a `HashMap` if the player is not exiting;
2. A method that accepts the direction and the location of where the player wants to go to, called in `LocationMap`;
3. The appropriate getters for `LocationId` and description;
4. A method that returns a `HashMap` of location ⇒ direction.

4.4 `LocationMap`

The `LocationMap` class is a class that behaves like a map by implementing `Map<>`.

Functionality:

1. Creates an object that represents each location;
2. The skeleton code provides two constants that represent the `.txt` files for
   locations and directions - DO NOT change them;
3. Create a static block that (within a try-with-resources / catch block) reads
   from the `locations.txt` so that a user can move from one location to
   another;
4. Also within the static block have a try-with-resources / catch block that
   reads from `directions.txt` so that a player can move from one location to
   another;
5. Implements the necessary methods required by Map (remember to
   change the return values to the appropriate ones used by the class.

4.5 ConsoleLogger

The `ConsoleLogger` class implements Logger and writes the output to the
console.

Functionality:

1. Creates a method that accepts a message parameter and prints the mes-
   sage to the console.

4.6 FileLogger

In this assignment, the output is written to a `.txt` file, in addition to the console.
The `FileLogger` class implements `Logger` and writes the output to a file.

Unlike the previous assignments where you made use of IntelliJ's "save
console output to file", here the messages must be printed to both console and
log file using `FileLogger` and `ConsoleLogger` that implement `Logger`. The
contents of the output `.txt` file and the console MUST be the same.

Functionality:

1. A static block that deletes the existing log file before recreating it;
2. The skeleton code provides a constant that represents the name of the
   `.txt` file to write your output to - DO NOT change it;
3. Creates a method that passes a message in as a parameter and appends it
   to the log file;
4. Makes use of a try-with-resources/catch/finally block to write a message to
   the file.

4.7 Logger

The `Logger` is an interface that is given to you.

Functionality:
1. This interface consists of one method header, namely `public void` log(
   `String message);`
2. This interface is implemented within the `FileLogger` and `ConsoleLogger`
   classes.

5   Non-functional requirements

5.1   Code requirements

Your code **should** follow the Java capitalisation conventions for naming vari-
ables, constants, methods, and classes. As a **minimum** you **should** use IntelliJ's
Problems tab to fix **all** occurrences of the following:
1. Variable initializer is redundant
2. Variable is assigned but never accessed
3. Value assigned to a variable is never used

> This does **not** mean you have to fix **all** the problems in IntelliJ's
> Problems tab: **only** the ones listed here. But do consider the others.

5.2   Submission requirements

The submission part of your assignment is every bit as important as the coding
part. Your submission `zip` file **must**:
1. be a `zip` file of your own IntelliJ project;
2. be created by IntelliJ;
3. be renamed according to the requirements in §D.2 How to submit.

Your submission `zip` file **must not**:
1. be bigger than 1 MB (one megabyte) in size when compressed;
2. have contents bigger than 1 MB (one megabyte) in size when uncom-
   pressed;
3. contain any one file bigger than 500 kb (five hundred kilobytes);
4. contain more than 100 files in total.

Failure to comply with the submission requirements can result in penalties.

6   Assignment IntelliJ project

You are given an IntelliJ project containing skeleton code: this a framework that
lacks functionality. You must use IntelliJ to expand this skeleton to complete the

assignment according to the instructions in §4 Coding tasks while following the
rules set out in §B Rules and §D Submission instructions.

6.1   Obtaining the assignment project

Go to Canvas, then go to the Java course, then to Assignment 3.[2] There is a link
to a `.zip` file. Download and unpack this file to its own folder and move the
unpacked directory (folder) somewhere sensible. This unpacked folder contains
an IntelliJ project.

6.2   Structure of the assignment project

The IntelliJ project has a src folder which contains the Java source code. The
project also contains a series of text files (which have `.txt` extensions). Some
files contain the phrase `TestInput` and these have corresponding `Expected-
Output` files. This means that using the input sequence specified in a partic-
ular `TestInput` file should generate the **exact** output in the correspondingly
numbered `ExpectedOutput` file: see §7 Testing your code for how to make use of
this.

6.3   Loading the assignment project

Remember when you load this project into IntelliJ to open the directory itself
rather than one of the files inside it. You are required to work in IntelliJ be-
cause you must submit your completed IntelliJ project. You are required to use
IntelliJ's built-in Problems tab to fix specific problems with your code, see §5
Non-functional requirements.

7   Testing your code

An essential part of becoming a skilled programmer is learning to test your own
code and to test it frequently. Generally you should only write a couple of lines
of code before testing again. This may seem slow initially but is actually fast in
the long run. To help you test your code, the project assignment is shipped with
input files paired with expected output files.

 To make use of this you must first run the main code once. It does not
matter whether or not this is successful: it is purely to auto-generate a run
configuration. You now need to edit the run configuration:

1. Run ❯ Edit Configurations…
2. choose the Modify options drop-down (the fastest way is with the short-cut
   key, Alt m on Windows and Linux)

---

[2] Direct link: https://canvas.bham.ac.uk/courses/56084/assignments/330198

3. set the option for Redirect input

4. set the option for Save console output to file to make sure that the output log
   file StudentFileOutput.txt has the same content as the console. Ig-
   nore any differences in line separators LF or CRLF.

5. outside the drop-down but still in the Edit Configurations dialogue, click the
   folder icon at the right-hand end of the Redirect input from: line and choose a
   suitable TestInput file

6. click the folder icon at the right-hand end of the Save console output to file: and
   choose StudentConsoleOutput.txt or another file if you wish, but the
   file must have already been created

7. choose OK to save the changes

Steps 2–5 can be seen in Figure 1, page 11.

Now when you run your program, IntelliJ will automatically use whichever
test file you chose to provide the input. You can make your own test file or
files, and either change files to substitute the input source or edit one or more
files to alter the input itself. It is advisable to start testing with simple input
before expanding to test a fuller range. As you add functionality, sometimes you
should go back to an older test file to ensure that things that used to work still
do. This is known as **Regression testing.**

To help you see how accurate your output is, each TestInput file supplied
with the project has a corresponding ExpectedOutput file. Once the program
has finished, the output will not only be on screen but also saved to the file you
specified. This is useful because IntelliJ has a built-in tool to help you see the
differences between your output and the expected output. Find the appropriate
output file in the Project window in IntelliJ's upper-left corner, right-click it, and
choose Compare With… (or, faster, press Ctrl d). Point the dialogue box to the
corresponding ExpectedOutput file. IntelliJ will now show you a window high-
lighting the differences. At the top of this comparison window you are advised
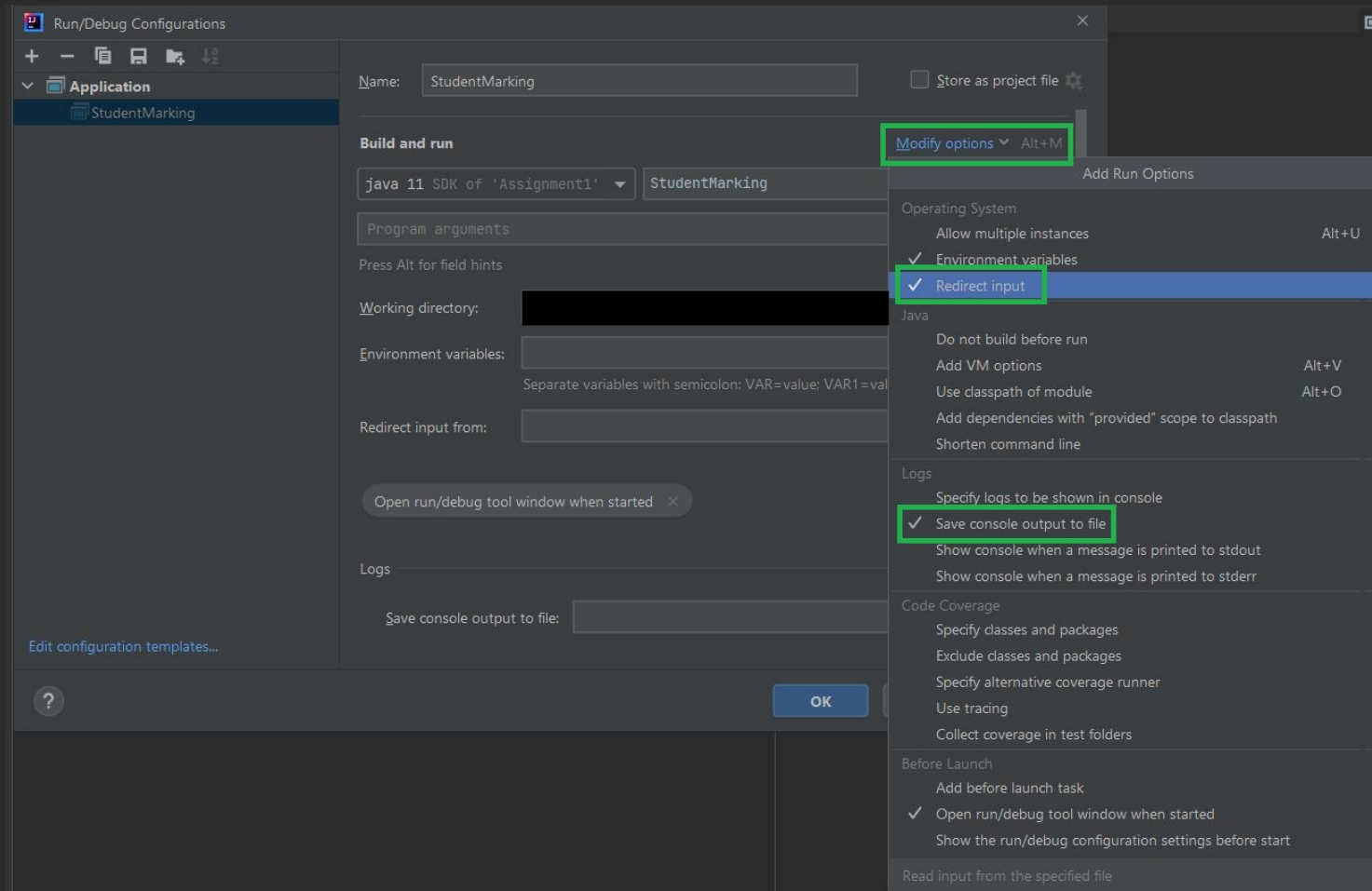to choose the options Side-by-side viewer and Do not ignore and Highlight characters .

Figure 1 Setting the run configuration to Redirect input from a test file and Save console to file. Note the output file must already exist.
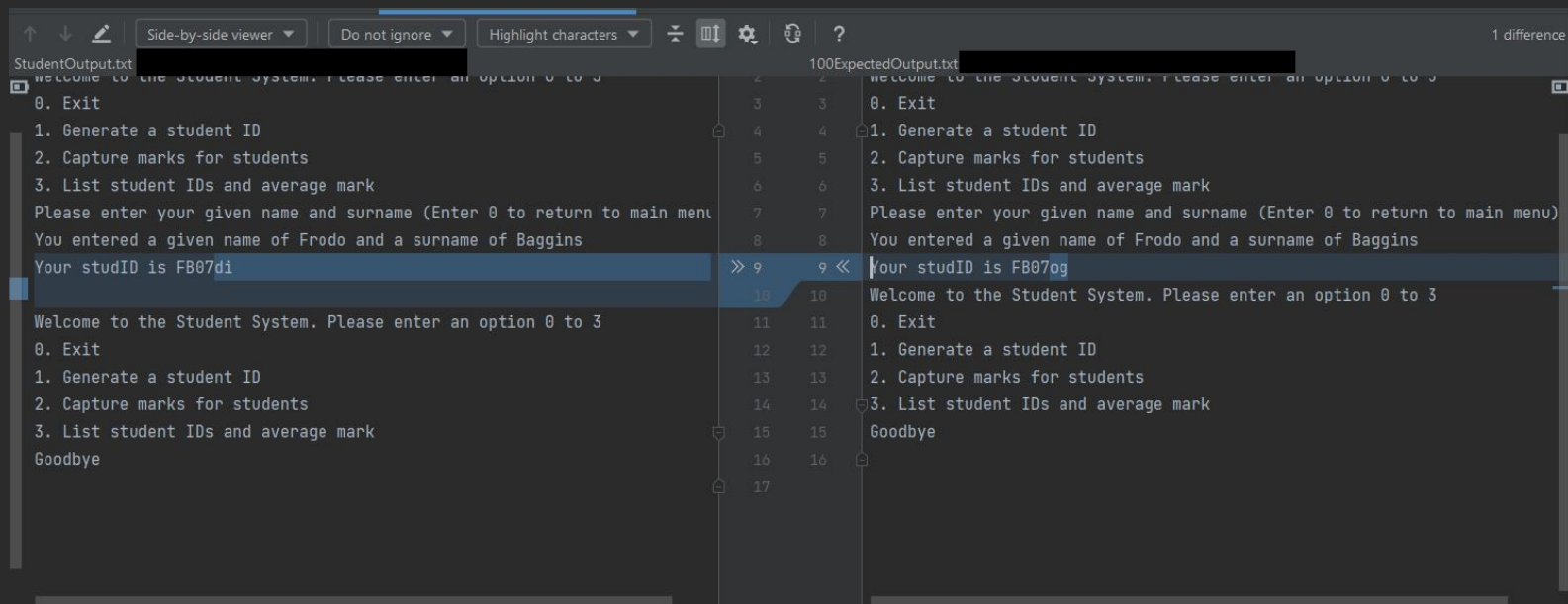
Figure 2 Viewing differences between output: actual output (left) and expected output (right). The image shows two characters are wrong on line 9 followed by an extra blank line in the actual output..

A   General tips

A.1   Before you begin coding

Do not rush to start typing, instead:

1. read **all** of the assignment specification carefully: some of the later tasks might reveal a better way to accomplish earlier tasks;
2. use pencil and paper to draw at least two different designs for each part of the assignment and compare them;
3. assess your different designs for relative advantages and disadvantages: this is future time saved not current time wasted.

A.2   While you are coding

The following tips will increase the speed of your writing the code and the quality of the code you produce:

1. only write one or two lines of code before testing what you have written — baby steps are by far the fastest overall;
2. make use of constants;
3. use IntelliJ's ability to read input from a file to apply rapid and consistent testing — you can write your own test input files;
4. use IntelliJ's ability to save the console output to a file and afterwards use IntelliJ's `Compare With…` (right-click the output file) function to compare your actual output with a file of corresponding expected output;
5. make use of IntelliJ's other tools:
   5.1. `Context actions…` `Alt` `Enter` for suggestions to improve or change the code where the cursor currently is
   5.2. the `Problems` tab `Alt` `6` to see if there are any major problems
   5.3. `Code` `Reformat Code` to pretty print your code
   5.4. `Code` `Inspect Code…` to identify inefficiencies and potential problems
   5.5. `Refactor` `Rename…` `Shift ⇧` `F6` to rename things safely everywhere they are used at once
   5.6. `Run` `Edit Configurations…` to specify files to redirect input from and a file to redirect output to: this greatly facilitates testing your code
   5.7. use the debugger — it is not as complicated as it first appears:
      5.7.1. click immediately to the right of an appropriate line number to set a break point
      5.7.2. use `Debug` instead of `Run`
      5.7.3. use the `Debugger` and `Java Visualizer` tabs to understand the current contents of variables as you step through the code
      5.7.4. use the arrow buttons (or better still the short-cut keys) on the debugger window that step into, step over, and step out of code.

6. make use of 'rubber duck debugging' (yes, rubber duck debugging) — if your code is not working as expected then explain it one line at a time aloud to an inanimate object, such as a rubber duck, and you will usually hear yourself say where your code is wrong: remember program code does **exactly** what you tell it to, so if it is not doing what you want, you are not giving it the correct instructions in the correct order;

7. if something is not working, assume there is more than one thing wrong and that the underlying error is probably at least one line ahead of where the error is causing your program to go wrong.

B   Rules

B.1   Critical rules

1. **We only mark the last version you submitted**, even if it is the wrong one.
2. If you do not submit a `zip` file of an `IntelliJ` project, you will score zero.
3. If you submit **code that does not compile** you **will score zero**. Therefore comment out any experimental code or test code and remove from your project any extra files that could interfere with compilation.
4. **Do not print any extra messages**, for example debugging messages, **at all**. **Your code's output must exactly match the expected output** in order to score marks. If you pollute the output with extra messages then you will harm your score, possibly as far as zeroing it.
5. If you know what `StandardErr` is then do not output to it or you will risk scoring zero because our autotester will think your program has generated errors.
6. You must use `OpenJDK 11` — no other version is acceptable.
7. You must use `IntelliJ`.
8. It does not matter to us if your program code runs on your computer. What matters is **your code must run on our computers.** This is not as difficult to achieve as it might sound. But you must ensure that none of your code contains anything specific to your computer. The easiest way to test this is copy the project to a directory that is not in your user area and try running it from there. You could try actually running it on another computer: a Virtual Machine of your own is safest. If you do run it on another computer, do not allow anyone else to copy your code, not even by accidentally leaving a copy on another computer.
9. Similarly if you let someone else use your computer, ensure they cannot copy your assignment.
10. Your code will be checked for potential plagiarism. If your code appears to have too much similarity to one or more other student's code, then you

and they are likely to be given zero for the assignment and potentially sub-jected to a disciplinary hearing, the consequences of which can be severe. Since you cannot prove who originally wrote the code and who copied it, everyone involved is usually given zero. Be careful then about copying code from the internet — copying designs or techniques is fine providing they are high quality. Also be careful about sharing code from the internet or links to the same source of help with other students because that can easily look like copying and it is hard to prove otherwise.

11. You cannot ask a Teaching Assistant or lecturer for specific help with a **current** assignment.
12. You must not discuss the details of your code with other students, nor disclose details of your code to other students.

B.2   Positive suggestions

1. Submit as many times as you wish up to the final deadline: it is useful to submit a preliminary version (complying with the rules) before the final deadline for safety reasons. Only the last submission is marked.
2. Submit a working version, even if unfinished, before the early testing deadline to try to discover whether your code works on our computers and how well your design is doing against a wider range of tests than those supplied with the assignment.
3. Discuss the **general design** of your program with other students.
4. Discuss the **concepts** required for your assignment with a Teaching Assistant. For example: although you are not allowed to ask anyone how you could write a particular `for` loop specific to an assignment, you are allowed — and encouraged — to ask a Teaching Assistant, or another student, to discuss how `for` loops work in general.
5. Discuss the details of your code from a previous assignment, whose dead-line for students with extensions has passed, with a Teaching Assistant. This is an excellent way to help you improve your programming in terms of actual code written and a way to help you improve designing programs.

C   Feedback reports and reporting marking problems

All assignments are autotested and automarked by one system. This ensures consistency for all students. It is not possible for us to preempt all possible legitimate interpretations of all the assignment's requirements. Thus once the assignments have been marked for the first time after the final deadline, you will be given a **preliminary report** which you will have **three days** to inspect and feedback probable or actual errors in the way your assignment has been

marked. We can ignore any requests which come in after three days. Any query you do submit must be evidence-based and specifically and unambiguously indicate where you believe the problem is. Speculative requests just trying to get a higher mark will not be entertained.

We will analyse the findings of genuine concerns and if we agree the auto-tester is wrong or inadequate, we will do our best to fix it or override it, and re-grade every eligible submission. This is a non-trivial task and it can take days. There may be more than one iteration of this. We usually automatically check all assignments which have scored below a certain threshold — which can vary between assignments — to ensure that the low mark is not an autotester error.

You will be told on the assignment Canvas page how to report potential problems with the marking and **you must** comply with those requirements. Although it is tempting to email a particular person, you must not do so because that is not a sustainable or consistent way for us to handle queries. Although you are sending only one query, we are receiving many, often overlapping, queries. We are looking for commonalities in your reports to help identify where the autotester or automarker might need changing.

It is important that the reporting mechanism is used only for reporting genuine or probable errors in the way the assignment has been marked: **it is not a mechanism simply to challenge your mark because you are disappointed.**

Eventually you will be given **a final report with a provisional mark**. All marks for all assignments and all exams for all courses are provisional until the exam board and the external examiners approve them. This happens in summer for undergraduates and autumn for postgraduates.

Once the final report is issued, no further re-grades will occur and the assignment is close permanently, and no further queries will be considered.

D   Submission instructions

D.1   Before you submit

As a minimum before you submit:

1. ensure your code compiles
2. ensure your code does not print anything it is not supposed to
3. ensure your code has not changed any of the class or method signatures from the skeleton code
4. check the `Problems` tab for categories listed in §5.1 Code requirements
5. check **every** class to ensure it is only `import`ing things explicitly allowed on the official `import` list on the assignment Canvas page
6. reformat your code: `Code` `Reformat Code`
7. ensure your code still compiles (yes, again)

Item 3 means you must not change from the skeleton code the keywords that precede a method or class name, nor change the parameters that a method takes, nor rename the method itself. If you find 'you need to' do one or more of these because the skeleton does not fit your design then it is your design that needs to change. Otherwise you will score zero. So if the skeleton code says:

```
public void printBarChart(String studId, int high, int low)
```

then your submitted code must have the **identical** signature. Changing the parameters in any way (adding or removing some or changing the order):

```
public void printBarChart(int high, int low, String studId)
public void printBarChart(String studId, int high)
```

or changing the keywords or method name:

```
public String printBarChart(String studId, int high, int low)
public void displayBarChart(String studId, int high, int low)
public void String printbarchart(String studId, int high, int low)
```
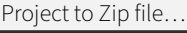
is wrong and will result in a zero score because your code does not compile.

D.2   How to submit

1. Build ⟩ Rebuild Project and fix any compilation errors.
2. File ⟩ Export ⟩ Project to Zip file… and rename the resulting zip file to
   JavaAssign3_Givenname_Familyname_studentIDnumber.zip
   If you officially have only one name then use X for the 'missing' name.
3. Ensure you do not include a previous exported zip file in the latest project export (a zip inside a zip) because this means you cannot guarantee the autotester will run the latest version of your code.
4. Upload the renamed zip file to Canvas for the Java course Assignment 3.
5. Canvas will probably add some extra information to the end of the file-name you have uploaded: do not worry about this.

Once you have uploaded, check you have uploaded the correct version:
6. Download your submission from Canvas and move it somewhere else.
7. Unpack that zip file.
8. Load the unpacked project into IntelliJ.
9. Check that it is definitely the version you intended to submit.
10. Check that the latest zip file does not also contain a zip file of an earlier version of your assignment because if we mark the wrong one, you are stuck with that mark.

D.3   Missing the deadline

If you miss the deadline, you might be able to submit late if the assignment allows late submission. Every assignment has a cut-off date after which no

further submissions are allowed. If Canvas will not allow you to submit, then you have missed the cut-off date and you cannot submit. There is no other way to submit: do not email your assignment to any member of the team because we cannot accept it.

D.4   Early test deadline

If you submit before the testing deadline, **Wednesday 8th December 2021 2pm GMT**, then we will **try** to run your most recent testing submission against a larger test set than is supplied with the skeleton code. You can use the feedback, if there is any, from the early submission to help you improve your assignment and ensure that it runs on our computers. This means you will be much better prepared for submitting for the final submission deadline.

We absolutely cannot currently guarantee this early testing service so do not rely on it. You can — and should — test your own code at any time as explained in §7 Testing your code.

If we do test your early submission, then you will be emailed a personalised report detailing what tests it passed and failed. Any 'score' from the early test submission is purely a guide and does not count towards your assignment because it has not been tested against the full set of tests. You may of course submit again (repeatedly if necessary) after the testing submission deadline, preferably before the final deadline. **Only your last submission counts for your assignment grade**; late submissions are subject to penalties; there is a final cut-off date for late submissions after which we ignore all further submissions.

D.5   Extensions for welfare reasons

Only the Welfare team are allowed to grant extensions, so please do not ask any of your lecturers or Teaching Assistants (for any module) for an extension. You can and should talk to Welfare in confidence about any problems and they do not tell us the reason for granting you an extension.

D.6   Assignment cut-off dates

There is a five-day window after the assignment deadline in which you may submit late with a penalty. After that window, submissions close permanently unless you have a welfare extension. The window for late submission is determined by the late penalty because eventually even an otherwise perfect submission cannot score enough marks to pass. Once submission closes, it closes permanently.