Object-Oriented Programming
Software Workshop 1

Assignment 1:
Student Marks Capture System

Set by
Jacqueline Chetty and Brian Mitchell

Early Testing Deadline:
Monday 18th October 2021 2pm BST

Final Submission Deadline:
Friday 22nd October 2021 2pm BST

This assignment is worth
0 %
of the overall course mark

You must use **OpenJDK 11**

Overview

Table of contents

List of figures

1   Introduction

You are tasked with completing a menu-driven text-based application that can:
1. generate a student ID based on a student's full name
2. capture (record) marks for multiple students
3. draw a simple bar chart of a student's highest and lowest marks
4. print a table of student IDs and corresponding average (mean) mark.

   Please read this assignment document carefully and completely. The submission guidelines and the rules will be broadly similar across assignments but you must still check for specific requirements. Also consider the School's and University's requirements for good academic practice.

2   Marking scheme

Question	Description	Marks
1	printf templates	10
2	displayMenu	5
3	main	25
4	generateStudId	15
5	captureMarks	15
6	printBarChart	20
7	reportPerStud	10
	Total	100

   The marks are awarded for functionality: partial functionality scores partial marks. This assignment is a practice so its marks do not count towards your course grade but the practice definitely will. Your submission will be tested and marked by computer, see §4 Non-functional requirements, §B Rules, and §C Submission instructions. Those sections explain how to score zero. Once your functionality marks are calculated, there can be deductions, for example:
1. late or improper submission or poor programming style;
2. breaking the specific requirements for this assignment, §4 Non-functional requirements, by letter or spirit;
3. breaking the general assignment rules, §B Rules, by letter or spirit.

3 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. In Software Engineering these are known as **Functional Requirements**. There are also Non-functional Requirements, see §4. One way to assess a program's quality is by measuring compliance with its Functional Requirements.

Overall, your program **must**:

1. compile, even if the functionality is incomplete;
2. follow the requirements set out in this section;
3. only ever use one `Scanner`;
4. account for any official changes to the assignment after the assignment has been released.

Overall, your program **must NOT**:

1. change the class signature `public class StudentMarking`;
2. change any of the method signatures;
3. add any methods;
4. remove any methods;
5. add any classes;
6. remove any classes;
7. `import` anything else, especially third-party libraries;
8. create any new `Scanner`s;
9. add any extra functionality beyond what is officially specified.

Overall, your program **should**:

1. exhibit consistent and predictable behaviour;
2. tolerate some slightly inappropriate input such as an `int` being outside the expected range;
3. be efficient in terms of execution speed, mainly by not performing unnecessary steps or calculations, though this is less important than the program working correctly.

Overall, your program **should NOT**:

1. crash when given input of the correct type...
2. ...even if that input is an `int` out of range.

In the subsections below, notes like this means the information is vital to the correct completion of the assignment but requires you to do further investigation to find what you need.

In the subsections below, notes like this give extra information that is useful but not vital to the completion of the assignment.

3.1   printf templates

Near the top of `StudentMarking` are a series of templates for use in `printf` statements elsewhere in the program. Each is defined **`public final`** `String` followed by an **identifier** then a **template** for a `printf` statement. For example:

```
public final String NOT_FOUND_TEMPLATE =
        "No student id of %s exists";
```

Note the `%s` to print a `String` variable specified in the appropriate `printf` statement which uses this template. This `String` will be printed using its natural width (because no width is specified) and in its original case. Also pay attention to the use of `SCREAMING_SNAKE_CASE` for naming constants in Java. You must know this convention and follow it at all times when programming in Java.

Each template **must**:
1. format the required variables appropriately, including making a `String` uppercase where required;
2. be a `String` constant (**`final`** `String`);
3. be a well-formed `printf` template;
4. be part of a well-formed Java statement declaring a `String` constant;
5. round numbers with decimals to 2 decimal places even if they have only 1 decimal place initially.

Each template **should**:
1. use `%n` instead of `\n` to produce a newline in the output to make it properly 'platform independent;'
2. use `SCREAMING_SNAKE_CASE` for its identifier.
3. use spaces or zeros for padding where appropriate.

Each template **should NOT**:
1. use multiple contiguous spaces within the template itself.

In order to calculate the correct widths for the various % symbols used in the templates, you must investigate the files of 'Expected-Output' supplied with the assignment, see §6 Testing your code.

> It is good practice to define constants for things whose values do not change providing the value is known when the program is being written. One excellent reason for this is it avoids 'magic numbers' and 'magic Strings' because you use a name not a value. Whereas `finalCost = totalPrice * ((100-10)/100);` is possibly confusing `finalCost = totalPrice * TEN_PERCENT_DISCOUNT;` is clear. You should also use a constant if there is a real possibility that the value might be needed more than once in the program — even if it is only used once now, the program might be extended in future. Because that might be done by someone else, constants need to have obvious names and be seen easily. Thus it is also good practice to define constants at or near the top of a class or method as appropriate. To help you see what constants (and other things) are available in a class, open the `Structure` tab in IntelliJ on the side of the lower left corner of the IntelliJ window. You can also show and hide `Structure` by pressing `Alt` `7` .

3.2 `displayMenu`

`public void displayMenu()`

must:

1. use `printf` with the appropriate `String` constant that has been provided near the top of the `StudentMarking` class to display the main menu.

It **must NOT**:

1. do anything else.

It **should**:

1. be one line long.

It **should NOT**:

1. create any local variables.

Note the exact correlation between the functionality and the method name displayMenu. Each method in a well-designed program has one clearly defined purpose. This purpose must be reflected in the method's name which typically starts with a verb. Often the method will need multiple steps but in this case it is a single step. Methods comprising only a single statement are common in well-written programs. This is good design because it means that method is simple for someone else to understand, is more likely to be reusable by other methods, is easily tested for reliability, and may be more resilient to changes made to other parts of the code now or later.

3.3 main

```
public static void main(String[] args)
```

must:

1. instantiate an instance of the StudentMarking class in the variable sm;
2. make appropriate use of the sm variable;
3. use the Scanner sc that has been provided in the skeleton code;
4. reject any inappropriate input to the main menu;
5. process appropriate input correctly;
6. perform any necessary preparations before calling the method from the sm variable corresponding to the menu options the user has chosen;
7. perform any necessary actions after the called method has exited;
8. handle calls to generateStudId appropriately:
    8.1. count the number of student IDs returned by generateStudId;
    8.2. store the result returned by generateStudId;
9. handle calls to captureMarks appropriately:
    9.1. prompt the user using the String provided to enter a student ID or a special code to cancel;
    9.2. read the user input and act appropriately;
    9.3. find the student ID in the appropriate array;
    9.4. only call captureMarks if the student ID has been found…
    9.5. …otherwise printf using the provided template about a student ID not being found;
    9.6. store the result returned by captureMarks;

It **must NOT**:

1. create another Scanner;
2. crash if an **int** outside the permitted range is entered;
3. instantiate any other instances of the StudentMarking class;
4. reinstantiate the sm variable.

It **should**:

1. assume that all input to the main menu will be an `int` (not necessarily within an appropriate range) followed by end-of-line;
2. provide some form of loop that repeatedly displays the menu and reads input from the keyboard;
3. use an appropriate and clear way to switch program execution between different methods;
4. make appropriate use of the local constants.

3.4 `generateStudId`

`public` String generateStudId(Scanner sc)

must:

1. use the `Scanner sc` from the method parameters;
2. use the program code provided to prompt the user to enter either a name or a special code to return to the main menu;
3. immediately return to the main menu if the user enters the special code instead of a name;
4. generate a student ID according to this specification in this order:
   4.1. the uppercase initial of the given name;[1]
   4.2. the uppercase initial of the family name;[2]
   4.3. the length of the family name, zero-padded to 2 digits;
   4.4. the middle letter of the given name in its original case;
   4.5. the middle letter of the family name in its original case.
5. print the generated student ID using the appropriate `String` constant template defined near the top of the class;
6. return the generated student ID when the method exits.

It **must NOT**:

1. create another `Scanner`;
2. directly save the generated ID anywhere outside the method (but see Item 6 from the previous list).

It **should**:

1. use appropriate local variables with names meaningful to someone else;
2. assume the user will enter both parts of a name (separated by a single space) on one line;
3. assume it does not matter if two different names generate the same ID.

---

[1] A 'given name' is also known as 'first name' or 'Christian name.'

[2] A 'family name' is also known as a 'surname.'

> You will need to declare local variables appropriately within the method. You will need to examine the 'ExpectedOutput' files provided with the assignment to understand what is meant by 'the middle letter of' a name.

> The system, even when fully completed for your assignment, does not guarantee that student IDs are unique. This does not matter for this assignment, although it would be catastrophically stupid for a real world records system.

3.5 `captureMarks`

`public double captureMarks(Scanner sc, String studId)`   165

must:

1. read three marks in **any** order;
2. only accept marks as `int`;
3. only accept marks that are within the permitted range;
4. correctly identify the highest mark;   170
5. correctly identify the lowest mark;
6. calculate the average (mean) of the three marks;
7. print the appropriate message about the lowest and highest mark;
8. print the appropriate message about the average mark;
9. prompt the user (using the provided code) to decide whether or not to   175
   print a bar chart;
10. call the appropriate method to print the bar chart if and only if the user
    enters either `Y` or `y`;
11. return the student's average (mean) mark.

It **must NOT**:   180

1. assume every integer entered as a mark will be within the permitted
   range;
2. create another `Scanner`;
3. directly save the calculated average anywhere outside the method (but see
   Item 11 from the previous list).   185

It **should**:

1. make use of the constants that set the minimum and maximum allowed
   mark;
2. use appropriate local variables with names meaningful to someone else;
3. use an outer loop to read the three `int`s that are the student's grades;   190

4. use an inner loop to repeatedly read an **`int`** until the most recently entered
   `int` is within the permitted range for marks;
5. assume that only integers will be entered for student marks.

It **should NOT**:

1. calculate the mean score more than once per invocation of the method.   195

> ❗ You will need to inspect the 'ExpectedOutput' files to ascertain the
> correct formats for the two messages to print, Item 7 and Item 8
> from the **must** list of requirements.

3.6 `printBarChart`

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

must:

1. make use of the method's parameters;   200
2. first print an opening message using the provided code;
3. print a two-column **vertical** bar chart using a local **`char`** constant;
4. use the appropriate `printf` template for the highest mark column;
5. use the appropriate `printf` template for the lowest mark column;
6. print the column labels underneath the bar chart using the appropriate   205
   `printf` template.

It **should**:

1. use appropriate local variables with names meaningful to someone else.

It **should NOT**:

1. change the values of **`int`** `high` and **`int`** `low` in the parameters.   210

> ❗ You will need to inspect the 'ExpectedOutput' files to ascertain the
> correct formats for the four messages to print from the **must** list of
> requirements: Item 2, Item 4, Item 5, and Item 6.

> ℹ Having a method change the values of its parameters can be risky
> to your program's smooth running because it can have unintended
> consequences if done unwisely or unsafely. Although sometimes you
> do want to change parameter's values and cause an effect elsewhere
> in the program, generally it is good practice to avoid changing the
> value of a parameter without good reason.

3.7   `reportPerStud`

```java
public void reportPerStud(String[] studList,
                          int count,
                          double[] avgArray)
```

must:
1. make use of the method's parameters;
2. print one record for every student currently in the system
3. print one record per line;
4. number the IDs starting from 1;
5. format the number of the ID currently being printed to a width of three padded with spaces;
6. format the actual ID being printed to a width of five, padded with trailing spaces, even though these are unlikely to show with the current ID format;
7. format the average mark to a total width of five, padded with spaces, and rounded to two decimal places.

It **should**:
1. use a loop to print all student records, one at a time;
2. use appropriate local variables with names meaningful to someone else;
3. use only a single `printf` statement to handle the entire printing requirements of this method.

> **❗** You will need to inspect the 'ExpectedOutput' files to ascertain the correct format for the table row, Item 2 from the **must** list of requirements.

4   Non-functional requirements

4.1   Code requirements

Your code **should** follow the Java capitalisation conventions for naming variables, constants, methods, and classes. If it does not, your grade might suffer deductions, and will **SCORE ZERO IF IT DOES NOT COMPILE**. As a **minimum** you **must** 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.

4.2   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 §C.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 uncompressed;
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.

5   Assignment IntelliJ project

You are given an IntelliJ project containing skeleton code, that is you are given a framework that lacks functionality. You must use IntelliJ to expand this skeleton to complete the assignment according to the instructions in §3 Coding tasks while following the rules set out in §B Rules and §C Submission instructions.

5.1   Obtaining the assignment project

Go to Canvas, then go to the Java course, then to Assignment 1.[3] There is a link to a `.7z` 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.

5.2   Structure of the assignment project

The IntelliJ project has a `src` folder which contains `StudentMarking` the Java source code. The project also contains a series of text files (which have `.txt` extensions). These files start with three digits to group them together. Some files contain the phrase `TestInput` and these have corresponding `ExpectedOutput` files. This means that using the input sequence specified in a particular `Test-Input` file should generate the **exact** output in the correspondingly numbered `ExpectedOutput` file: see §6 Testing your code for how to make use of this.

---

[3] Direct link: https://canvas.bham.ac.uk/courses/56084/assignments/330196

5.3 Loading the assignment project

Remember when you load this project into IntelliJ to open the directory not
one of the files inside it. You are required to work in IntelliJ because 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 §4 Non-functional   280
requirements.

6 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   285
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: 290

  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)
  3. set the option for Redirect input
  4. set the option for Save console output to file   295
  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 StudentOutput.txt or another file if you wish, but the file must   300
     have already been created
  7. choose OK to save the changes

  Steps 2–5 can be seen in Figure 1, page 12.

  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   305
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.**   310

  To help you see how accurate your output is, each TestInput file supplied
with the project has a corresponding ExpectedOutput file. Once the program
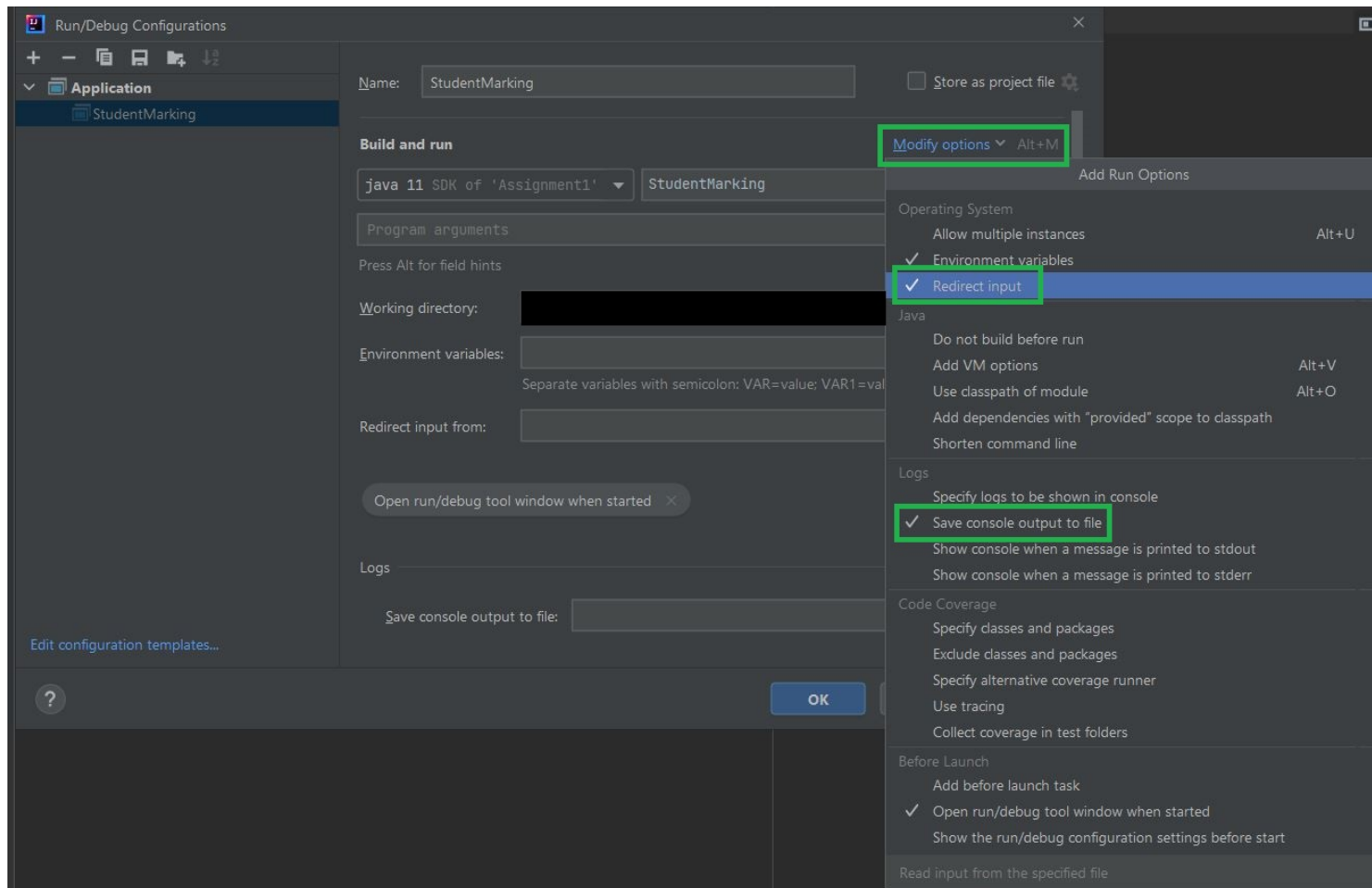has finished, the output will not only be on screen but also saved to the file you

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.
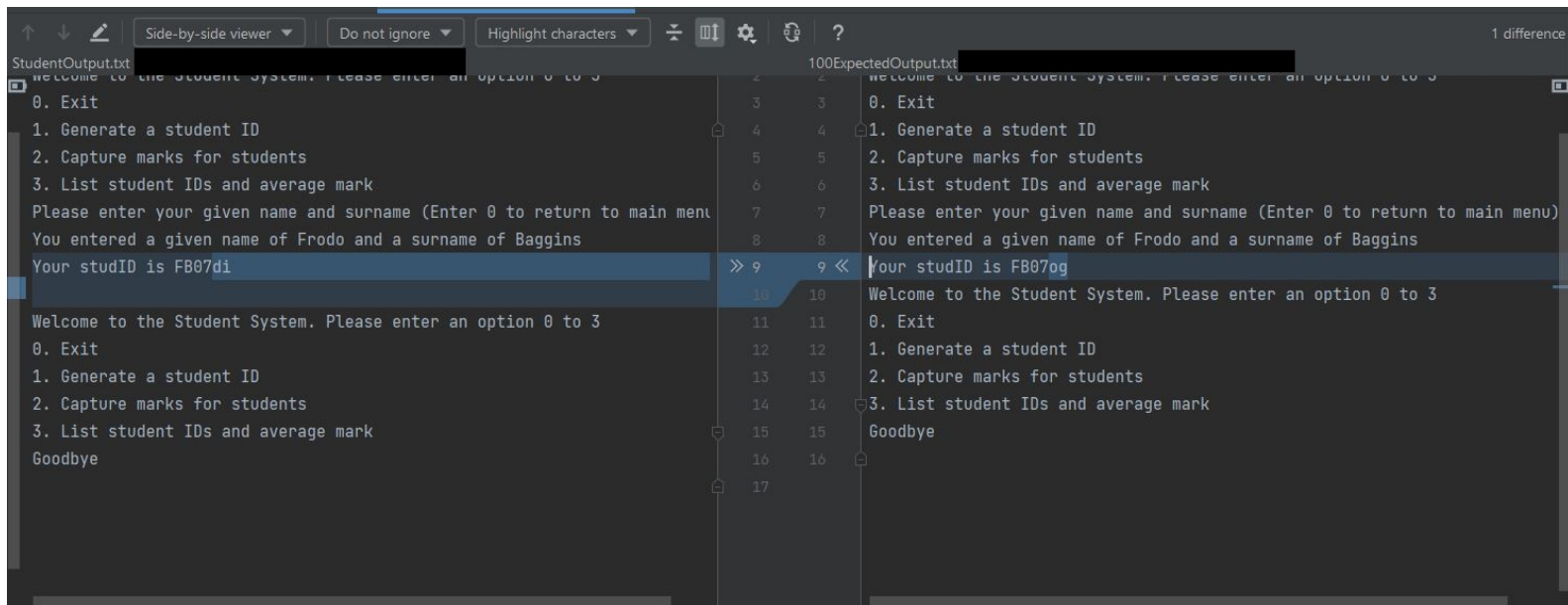
Figure 2 Viewing differences between output: actual output (left) and expected output (right). The image shows two characters are wrong on line 9 followed by an extra blank line in the actual output..

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 highlighting 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.

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.

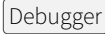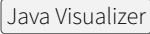### A.2 While you are coding

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: start by copying one of the test input files provided with the assignment;
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 quickly and reliably

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

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 subjected 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.

Here are some positive suggestions for things you are encouraged to do:

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 deadline 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 Submission instructions

C.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 the specific types of problems listed in §4.1 Code requirements
5. reformat your code: `Code` 〉 `Reformat Code`
6. 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 you need 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 could easily result in a zero score, which it will if the program does not compile.

C.2 How to submit

1. `Build` 〉 `Rebuild Project`
2. `File` 〉 `Export` 〉 `Project to Zip file…`
3. Rename the resulting `zip` file to `JavaAssign1_Givenname_Familyname_studentIDnumber.zip`
4. 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
5. Upload the renamed `zip` file to Canvas for the Java course Assignment 1

Check you have uploaded the correct version:

6. Download your submission from Canvas   465
7. Move that `zip` file somewhere else
8. Unpack that `zip` file
9. Load the unpacked project into IntelliJ
10. Check that it is definitely the version you intended to submit
11. Check that the latest `zip` file does not also contain a `zip` file of an earlier   470
    version of your assignment.

C.3   Early test deadline

If you submit before the testing deadline, **Monday 18th October 2021 2pm BST**,
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   475
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 ex-   480
plained in §6 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   485
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.

C.4   Extensions for welfare reasons   490

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.