
Object-Oriented Programming
Software Workshop 1

Assignment 2: Bingo!

Set by
Brian Mitchell and Jacqueline Chetty

Early Testing Deadline:
Tuesday 16th November 2021 2pm GMT

Final Submission Deadline:
Sunday 21st November 2021 5pm GMT

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

You must use **OpenJDK 11**

Overview

1	Introduction	1
2	Important points	1
3	Marking scheme	3
4	Coding tasks	3
5	Non-functional requirements	10
6	Assignment IntelliJ project	11
7	Testing your code	11
A	General tips	12
B	Rules	16
C	Feedback reports	17
D	Submission instructions	18

Table of contents

1 Introduction	1
2 Important points	1
3 Marking scheme	3
4 Coding tasks	3
4.1 Overall functional requirements	4
4.2 BingoRunner	6
4.3 Toolkit	6
4.4 Defaults	7
4.5 BingoController	7
4.6 BingoCard	8
5 Non-functional requirements	10
5.1 Code requirements	10
5.2 Submission requirements	10
6 Assignment IntelliJ project	11
6.1 Obtaining the assignment project	11
6.2 Structure of the assignment project	11
6.3 Loading the assignment project	11
7 Testing your code	11
A General tips	12
A.1 Before you begin coding	12
A.2 While you are coding	15
B Rules	16
C Feedback reports	17
D Submission instructions	18
D.1 Before you submit	18
D.2 How to submit	19
D.3 Missing the deadline	19
D.4 Early test deadline	20
D.5 Extensions for welfare reasons	20
D.6 Assignment cut-off dates	20

List of figures

1 Setting the run configuration	13
2 Viewing differences between output	14

1 Introduction

You must complete a menu-driven text-based **limited** implementation of a bingo game. In real life, bingo¹ is a competitive game of chance where numbers are repeatedly drawn in a true random sequence from a pre-determined set by a person, known as a ‘caller’, or by a machine. Fresh numbers are drawn until the first player to realise they have matched a winning pattern on their bingo card announces this win aloud, typically shouting ‘house’ or ‘bingo.’ If multiple players complete a winning pattern simultaneously, the winner is the first to shout out. A bingo card, also known as a ‘ticket,’ is a grid of (mostly) non-consecutive numbers. The actual configuration of bingo cards varies between different versions of bingo games, so the configuration of cards used in this assignment might differ from cards you have seen before.

To keep the size of the assignment manageable and so it can be auto-marked reliably, you will not be implementing a full bingo game — in particular there will be no randomisation of numbers, definitely no shouting out, and no prizes for winning, although there are assignment marks to be ‘won’.

A complete and correct assignment will be able to:

1. set the dimensions of the bingo cards (rows by columns);
2. set the numbers contained on every bingo card in the game;
3. process sequences of numbers to be ‘called’;
4. mark matched numbers on all bingo cards currently in use;
5. test whether a bingo card is a fullhouse winner.

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) plus analysing 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,

¹ [https://en.wikipedia.org/wiki/Bingo_\(British_version\)](https://en.wikipedia.org/wiki/Bingo_(British_version))

- 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; 40
 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; 45
 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; 50
 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); 55
 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. 60

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; 65
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**; 70
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. 75

3 Marking scheme

Question	Description	Marks
1	Toolkit	12
2	BingoRunner	8
3	BingoCard	40
4	BingoController	40
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** and **must not** lists, then it will probably obtain a reasonable pass mark — deductions for late penalties and rule-breaking notwithstanding. To obtain a higher mark, your submission must also meet the requirements listed under **should** and **should not**.

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

105

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.

110

Overall, your program **must**:

115

1. [†] only ever use **one** Scanner to read from the keyboard;²
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**:

120

1. [‡] **import** any other libraries apart from the ones that are allowed;
2. [‡] change any signatures of classes, class fields, or methods;
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 Scanners.

125

130

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;

135

² The one to use is provided in the skeleton code.

5. be efficient in terms of execution speed³ 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 **imports** 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 **ints** 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.

³ We test this by limiting the amount of time your code program is allowed to run for.



Do not be misled by the amount of methods and class fields which use **public** visibility. Using **public** everywhere is not good software engineering because the ‘scope’ of items (constants, variables, methods, classes) should generally be as limited as possible to minimise the chances of other code doing damage. We have made many of the items in this assignment **public** simply to facilitate auto-testing. This allows us to access these items from a completely different class so we can test parts of your code without having to run your whole code. This potentially helps us give you marks if your overall submission is incomplete or imperfect.

Limitations:

155

1. Assume all cards in the game have the same size;
2. The input sequence of numbers will only terminate once a winner is found;
3. The only winning sequence is fullhouse;
4. Assume all numbers are positive numbers, to a maximum of 99;
5. There could only be one winner, therefore if multiple players complete a fullhouse simultaneously, the chosen winner will be the player that has the card numbered with the lowest index. For example, if players with cards 0, 1 and 2 win simultaneously, then card 0 counts as the winner.

160

4.2 BingoRunner

165

The BingoRunner class provides the entry point to the program.

It **must**:

1. create and execute a new BingoController;
2. print the appropriate goodbye message including any preceding or succeeding blank lines, which you must determine from analysing the ExpectedOutput files.

170

It **should NOT**:

1. have any **import** statements.

4.3 Toolkit

Toolkit is a utility class that provides functionality mostly related to input and output.

175

It **must**:

1. use the Scanner provided

2. be able to get a line of raw input as a `String`
3. be able to trim leading and trailing spaces from a line of input (a `String`); 180
4. use the appropriate method to print a message and return a (trimmed) line of input;
5. print an array of `String` according to requirements you will have to derive from analysing the `ExpectedOutput`.

It **should**:

185

1. perform basic cleaning up of input before returning it by trimming leading and trailing white space.

4.4 Defaults

The `Defaults` class provides home for some constants and access methods (getters and setters) to variables which must be accessed (read or write) through a single point. This class is fully implemented and no changes / additions are required to be made. 190

It **must**:

1. provide the constants required of it;
2. provide the controlled variables required of it; 195
3. provide getters and setters for the controlled variables.

It **must NOT**:

1. have any **import** statements.
- This class is fully implemented and requires no further changes.

4.5 BingoController

200

The `BingoController` class provides much of the program's functionality, including displaying and acting upon menu options.

It **must**:

1. provide a text menu for the user to operate your program including building a menu from a one-dimension array of `Strings`; 205
2. number those items according to the examples in the `ExpectedOutput` files;
3. provide a default size for the size of card rows and columns according to the specification from `Defaults`;
4. use and remember a new size for rows inclusive-or columns if the user changes them via the menu option; 210
5. use the required prompts for requesting various pieces of information according to the examples in the `ExpectedOutput` files;

6. maintain a data structure that stores zero or more BingoCard objects to be used during the game; 215
7. identify BingoCards in the order created using consecutive integer indices starting from zero;
8. have the ability to create a new BingoCard with the appropriate dimensions;
9. be able to store that new BingoCard; 220
10. be able to tell a new BingoCard what its numbers are;
11. be able to tell all BingoCards to reset themselves;

It **must not**:

1. change the provided menu text;
2. change the order or numbering of the menu items; 225
3. add any items to the menu;
4. remove any items from the menu.

It **should**:

1. implement all the functionality described in the main menu's options;
2. use getters and setters appropriately; 230
3. be able to print a BingoCard's numbers as a two-dimensional grid;
4. be able to list all the cards stored in the internal data structure to match the style in the ExpectedOutput files;
5. be able to change the separator between numbers to any arbitrary String;
6. be able to identify the index number of a card (the internal index to the BingoGame's data structure) 235

4.6 BingoCard

The BingoCard class does more than represent a real world bingo card because this is object-oriented programming and the actions are embedded in the object. A great example to illustrate this is to consider cooking a vegetable.⁴ In the real world you would think 'Here is a raw potato: shall I boil it, bake it, roast it, mash it, sauté it, or be British and make chips?' Whatever you choose (we know it's chips), **you** perform that cooking action **on** the potato. This means you know how to boil / bake / whatever⁵ a potato. This is a familiar way of doing things, even if you don't normally do your own cooking — or didn't until starting university. This distribution of labour is closely reflected in the imperative programming paradigm. 240

However the object-oriented world view is very different: it is essentially the other way round. An object-oriented potato can be thought of it as genetically

⁴ Tony Simons, University of Sheffield, used the example of cooking an egg.

⁵ Microwave probably.

modified to the point it has become partially intelligent. This means **your object-oriented potato knows how boil / bake / microwave itself**.

250

So in the imperative paradigm you do something to the potato, which can also be viewed as the potato having something done to it:

```
Potato myPotato;
```

255

```
boil(myPotato)
bake(myPotato)
roast(myPotato)
mash(myPotato)
```

In this example all the actions are some form of method that take a potato as a parameter. This means the cooking functionality is stored separately from the item being cooked. But in the object-oriented paradigm, the functionality is inside the object itself:

260

```
Potato myPotato;
```

265

```
myPotato.boil();
myPotato.bake();
myPotato.roast();
myPotato.mash();
```

In the real world, a paper bingo card is a passive object that has actions done to it: a player crosses numbers off, a player checks if it is a winner. The object-oriented bingo card is different because it 'knows' how to mark its own numbers and check if it's a winner.

270

BingoCard **must**:

1. be able to reset itself;
2. be able to store the set of numbers that will hopefully be matched during the game;
3. be able to record (mark off) when a number on the card matches the number that has been 'called;'
4. report if it is a winner based on the winning configuration fullhouse - all numbers marked off;
5. get and set the size of the card's rows and columns.

275

280

It **should**:

1. use getters and setters appropriately including using setters inside the constructor;
2. use a `StringBuilder` for efficiency instead of overwriting a `String` with a concatenated version of itself;
3. get and set the card's numbers from and to a data structure that is different from the card's internal data structure.

285



Item 3 from the **should** list is not computationally efficient because it involves changing the representation of the same data multiple times. This may be undesirable in speed-critical systems. However if it does successfully hide the internal storage mechanism of a Bingocard's numbers. Information hiding, layers of abstraction, and loose coupling are key techniques to master in designing software and hardware.

290

5 Non-functional requirements

5.1 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:

295

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.

300

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.

305

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.

310

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

6 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 §4 Coding tasks while following the rules set out in §B Rules and §D Submission instructions.

315

6.1 Obtaining the assignment project

Go to [Canvas](#), then go to the Java course, then to Assignment 2.⁶ 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.

320

6.2 Structure of the assignment project

The IntelliJ project has a `src` folder which contains `BingoRunner` 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 `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.

325

330

6.3 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 §5 Non-functional requirements.

335

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.

340

⁶ Direct link: <https://canvas.bham.ac.uk/courses/56084/assignments/330197>

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)
3. set the option for **Redirect input**
4. set the option for **Save console output to file**
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 have already been created
7. choose **OK** to save the changes

Steps 2–5 can be seen in [Figure 1](#), page 13.

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 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;

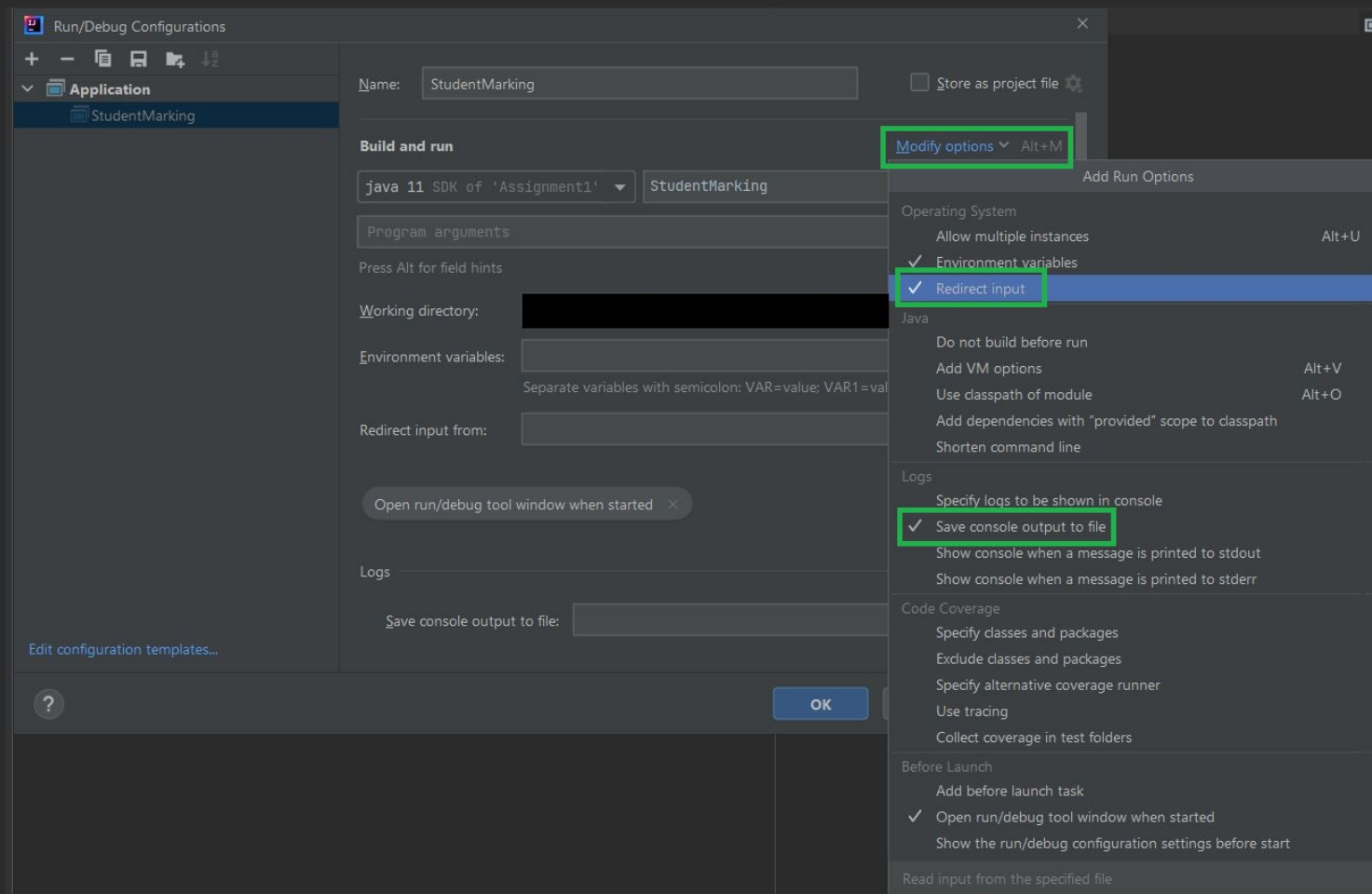


Figure 1 Setting the run configuration to Redirect input from a test file and Save console to file. Note the output file must already exist.

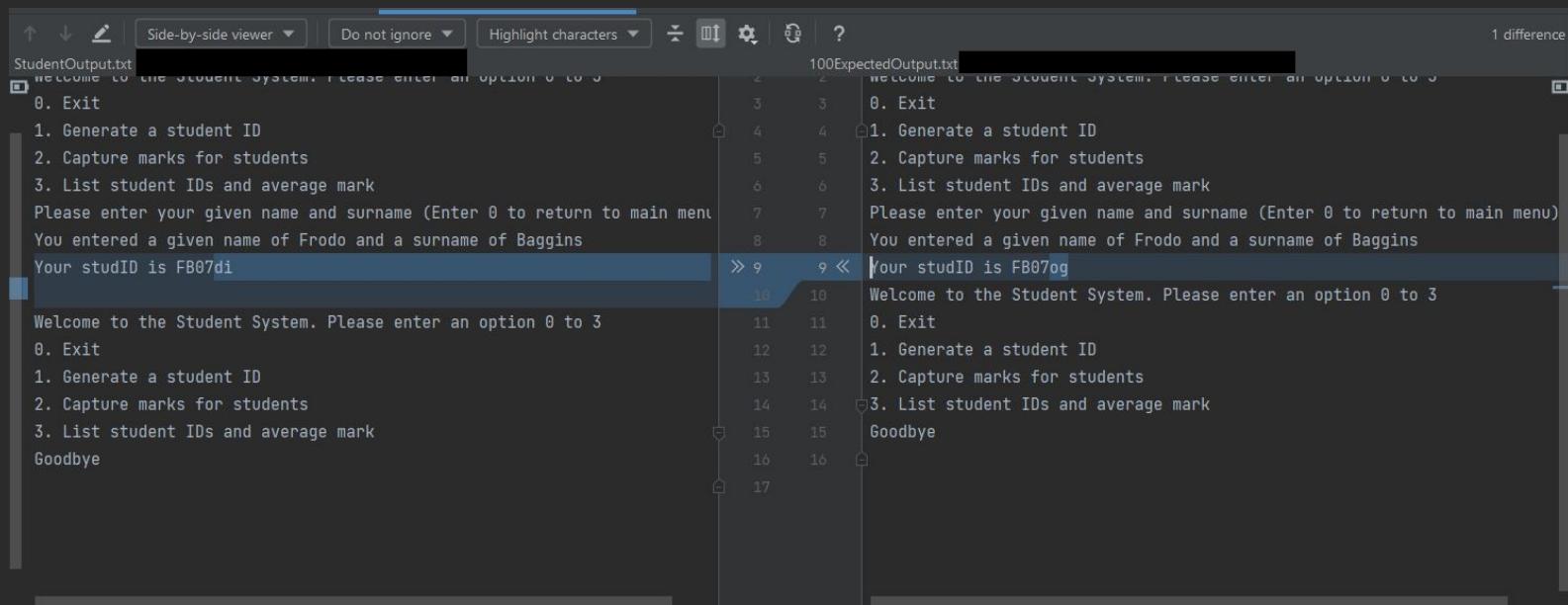








Figure 2 Viewing differences between output: actual output (left) and expected output (right). The image shows two characters are wrong on line 9 followed by an extra blank line in the actual output..

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.

385

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...**   for suggestions to improve or change the code where the cursor currently is
 - 5.2. the **Problems** tab   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...**   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

390

395

400

405

410

415

420

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.

425

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.

430

435

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.

440

445

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

450

455

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.

460

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.

465

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.

470

475

480

C Feedback reports

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.

485

490

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 usually a non-trivial task and it can

495

take days. There may well 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.

500

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.

505

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.**

510

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.

515

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:

520

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 §5.1 Code requirements
5. check **every** class to ensure it is not **importing** anything that is not explicitly allowed on the only official **import** list which is on the assignment Canvas page
6. reformat your code: **Code** **Reformat Code**
7. ensure your code still compiles (yes, again)

525

530

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):





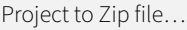
```
public void printBarChart(int high, int low, String studId)
public void printBarChart(String studId, int high)
```

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.   and fix any compilation errors.
2.    and rename the resulting zip file to JavaAssign2_Givenname_Familynam_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 2.
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 but it depends on the assignment. You must submit via Canvas, no other way. Do not email your assignment to any member of the team. If you are unable to submit via Canvas then you have missed the deadline.

D.4 Early test deadline

If you submit before the testing deadline, **Tuesday 16th November 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.

575

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](#).

580

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.

585

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.

590

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.

595

600

As stated in §C [Feedback reports](#), you have three days from when you receive your preliminary report to tell us if you believe the automarker has made a mistake, and you must use only the method of communication specified on the appropriate assignment Canvas page: no other means of communication is acceptable and will be ignored. Requests to investigate must be evidence-based on potential errors not speculative.

605

