

SOFTENG 281: Object-Oriented Programming
Assignment 2 (20% of final grade)
Due: 9:00pm, 19th April 2021 (End of Week 6)

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

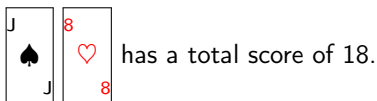
- Gain confidence modelling an Object-Oriented Programming problem.
- Gain confidence programming in Java.
- Apply OOP concepts such as inheritance, polymorphism, and encapsulation (information hiding).
- Apply good coding practices such as naming conventions and code reuse.

1 The BlackJack Game

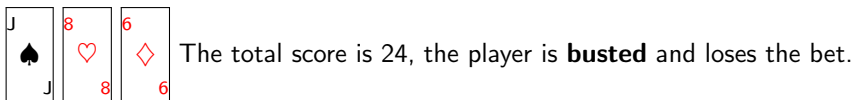
In this assignment, you will implement the BlackJack game! You will also implement some Artificial Intelligence (AI) that will play against you! Can you implement an AI smarter than you? Let's find out in this assignment.

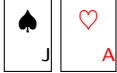
We will implement a **very** simplified version of BlackJack (no doubling or splitting). The rules are as follows:

1. The goal of **blackjack** is to beat the dealer's hand without going over **21**.
2. We play with a standard **52 cards** deck, which has 13 cards of each suit, namely clubs (♣), diamonds (♦), hearts (♥) and spades (♠). Each suit has 13 cards: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King.
3. Jack, Queen, King are worth 10 (rank = 10). **Aces are worth 1 or 11**, whichever makes a better hand. The ranks of the remaining cards are their corresponding numbers.
4. Before the hand starts, the player bets some chips between 1 and 100.
5. Each player hand starts with two cards. The ranks of these two cards are summed together. This is the current score of the hand. For example:



6. Then the player can '**hit**' to ask for another card, or '**hold**' to hold the total and end the turn.
7. If the total goes over 21 the player is **busted**, and the dealer wins regardless of the dealer's hand (i.e., the dealer wins even if it also exceeds 21). For example, if the player asked for another card ('**hit**'):



8. If the player has 21 from the start:  this is a **blackjack** and the player wins 1.5 times the bet, regardless if the dealer also does blackjack or not.
9. Then is the turn of the other players.
10. When all the players finish their turn, the dealer plays. When the dealer finishes their turn, we need to check which player wins against the dealer.
11. A winning player gets paid 1x the bet. If the player wins with a blackjack, they get back 1.5x the bet. For example, if the original bet was 100 chips a winning player gets back the original bet (100 chips), plus 100 chips from the dealer. If the player won with a blackjack, they get back the original bet (100 chips), plus 150 chips from the dealer. Intuitively, a player loses the original bet if the dealer wins.

You have been given an incomplete implementation of the game, you need to implement the missing parts. More specifically, you have four tasks to complete. Some classes should not be modified but you need to inspect them to understand how the game is implemented. Other classes (when specified) should be modified. You are encouraged to add helper methods or instance fields to these classes where necessary to structure and simplify code, but **you should not change any existing method declarations** (except where specified). You will find some comments inside the code that will help you to complete the assignment:

ADDHERE You need to add one or more statements in that location

TODO You need to implement the method

KEEPME You need to keep the statement

FIXME You need to change the statement

These comments are there to help you to complete the tasks. However, you might add new methods and instance fields if you think appropriate even if there are no special comments.

2 Files provided

Makefile This Makefile will allow you to build and run this assignment. You can either build and run the code in your machine via Eclipse or via command line. You can also run your project in replit.com. Ultimately, once you finish your project and before you submit, you will want to **test your project using the command line in replit.com** as this is how your project will be marked. The main commands relevant to this file are:

make Compiles the Java classes and runs the test cases.

make dependencies Finds for the commands java and javac, if it does not find them it triggers an error make: *** [dependencies] Error 1 . In such a case please follow the course help videos to properly setup your machine.

make clean Removes the Java binary files (*.class) in the bin folder.

make build Compiles the Java classes.

make run Runs the test suite (in Eclipse it is equivalent to run the BlackJackTestSuite class).

make play Runs the game (in Eclipse it is equivalent to run the BlackJack class).

make all This is needed by Eclipse, as it looks for a target named all (this command is equivalent to invoking make).

src/main/java/nz/ac/auckland/softeng281/a2/BlackJack.java This class is the entry point of the game. You may add to this file if you want to, but do not change the existing declarations. *You must modify this class to complete all tasks.*

src/main/java/nz/ac/auckland/softeng281/a2/Card.java This class declares a Card instance, which is composed of a rank and a suit. Note that the class relies on Enums. For defining constants, Java Enums are usually preferable instead of static final int or String variables. This leads to less error-prone code because with enum only the applicable values can be used. Instead, if we use Strings for constants any (wrong) String value can be erroneously used. *You should not modify this class.*

src/main/java/nz/ac/auckland/softeng281/a2/Deck.java This class defines the Deck of cards and allows basic operations like drawing a card and shuffling the deck. *You should not modify this class.*

src/main/java/nz/ac/auckland/softeng281/a2/Utils.java Utility class. *You should not modify this class.*

src/main/java/nz/ac/auckland/softeng281/a2/Participant.java This class defines a Participant of the game, who can be either a Player or a Dealer. The class is abstract, and thus cannot be instantiated. This means that you cannot do `Participant p = new Participant();`. The purpose of this abstract class is to function as a base class implementing common behaviours (to avoid repeated code) and defining a common interface (with abstract methods). The classes `HumanPlayer`, `BotPlayer`, and `Dealer` extend the `Participant` class inheriting its functionalities. *You may add to this file if you want to, but do not change the existing declarations.*

src/main/java/nz/ac/auckland/softeng281/a2/HumanPlayer.java This class defines you, the player. It reads the input of the player from the standard input (the console). *You should not modify this class.*

src/main/java/nz/ac/auckland/softeng281/a2/BotPlayer.java This class defines a bot that plays automatically. The Bot has the same objective as you, defeating the Dealer. *You must modify this class to complete Task 1.*

src/main/java/nz/ac/auckland/softeng281/a2/BotDealer.java The Dealer is a special player of the game, it doesn't place bets. The goal of the dealer is to defeat at least half of the players. *You must modify this class to complete Task 2.*

src/main/java/nz/ac/auckland/softeng281/a2/Hand.java This class represents a hand of the game. Each participant of the game has associated a list of hands that represent the history of all the hands since the beginning of the game. *You may add to this file if you want to, but do not change the existing declarations.*

src/test/java/nz/ac/auckland/softeng281/a1/BlackJackTestSuite.java This Java file contains the JUnit test cases for this assignment. These test cases make sure that you have implemented most of the assignment correctly. Making sure that your program passes all of the tests you have been given will not guarantee that you will get 100% of the marks for this assignment, but it will mean that you get many of the marks for the parts that work according to those tests. Writing your own tests or extending the ones you have been given is strongly recommended, as there are some requirements in this assignment that the existing tests do not cover. Look at the assignment specifications, and think carefully about which specifications were explicitly mentioned but there aren't test cases provided to you. It is important that you do not change the existing tests. Be aware that your code must still work with the original tests.

3 Tasks

Before proceeding with the tasks read carefully the code and try the initial version of the game yourself. Run `make play`, or if you are using Eclipse run the `BlackJack` class. The game will ask you how much you want to bet and will show you your first two cards. By typing "hit" or "hold" you can decide your next move. When you finish your turn the game asks you if you want to play again or not. In the initial version of the game you are playing alone :(. In this assignment, you will implement two bot players and a bot dealer to play together and against you! You will also implement some game analysis to identify the best player. Let's begin!

3.1 BotPlayer [Task 1]

As soon as you compile and run the tests (with `make` command, or if you are using Eclipse run the `BlackJackTestSuite` class), you will notice the following output:

MAKE:

FAILURES!!!

Tests run: 8, Failures: 8

OR

Eclipse:

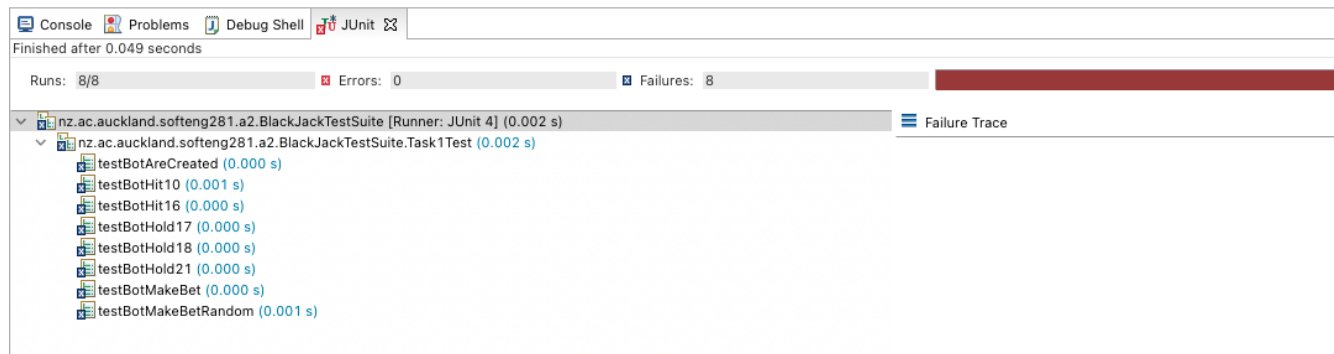


Figure 1: 8 failing test cases - Task 1

This is telling you that 8 test cases were run, none of them passed, and all of them failed. If you scroll up a little bit in that output, you will find more details on which particular tests failed.

The test `testBotAreCreated()` fails because there are no `BotPlayer` in the game yet. Go to `BlackJack.java`; you need to add two `BotPlayer` objects in the players list. Their names should be "Bot1" and "Bot2". If you do this correctly and re-run the tests, you will see that the first test will pass. We still have seven tests to go for Task 1.

MAKE:

FAILURES!!!

Tests run: 8, Failures: 7

OR

Eclipse:

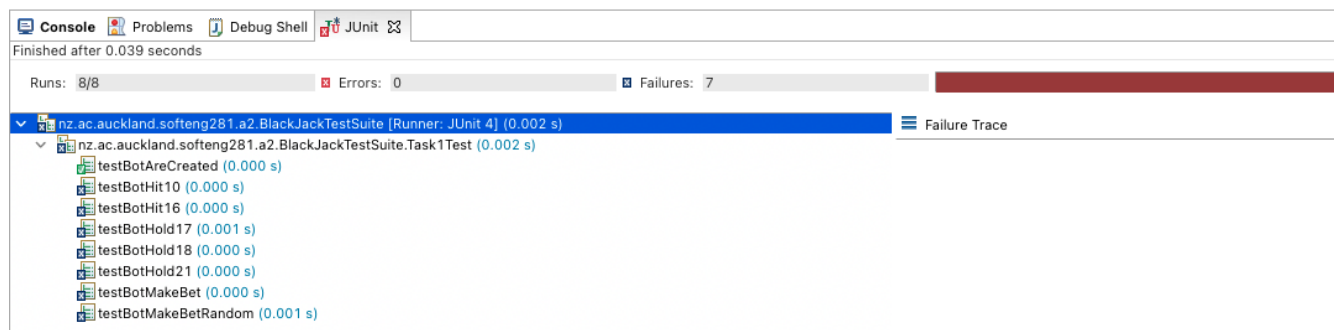


Figure 2: 7 failing test cases - Task 1

Run `make play`, or if you are using Eclipse run the `BlackJack` class. You will see the bots in action – these are not very smart bots. If you open `BotPlayer`, you will know why:

```

package nz.ac.auckland.softeng281.a2;

import java.util.Random;

/**
 * you should change this class for TASK 1
 */
public class BotPlayer extends Participant {

    public BotPlayer(String name) {
        super(name);
    }

    @Override
    public Action decideAction() {
        // TODO
        return new Random().nextBoolean() ? Action.HIT : Action.HOLD; // FIXME
    }

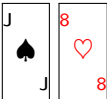
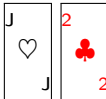
    @Override
    public int makeABet() {
        // TODO
        return -1; // FIXME
    }
}


```

Bots decide randomly and never bet anything!

Complete the TODOs and FIXMEs. The expected behaviours of the methods are the following:

decideAction() returns Action.HIT if the total score of the current hand is less than 17, Action.HOLD otherwise

(≥ 17). For instance, with this hand  the bot must choose Action.HOLD, while with this hand  the bot must choose Action.HIT.

 the bot must choose Action.HIT.

makeABet() returns a random integer number between 1 and 100 (inclusive).

You might ask yourself "How can I know the total score of the current hand?". Recall that the class BotPlayer extends the Participant class. As such, all the fields and methods of the Participant class can be accessed by the BotPlayer class. So it should be easy to get the score of the current hand to implement the decideAction method.

Regarding makeABet(), how to return a random integer between 1 and 100 (inclusive)? Well, remember that Google and StackOverflow are the most important tools for a developer! For example:

<https://stackoverflow.com/questions/5271598/java-generate-random-number-between-two-given-values>

is answering this question, using the class java.util.Random. However, it is very important to **always** read carefully the description of the code and **always** try and test the code before using it. Especially in Stackoverflow.com, never just copy and paste code. Often there are multiple answers to the same question, you need to understand the one that is suitable in your case. For example, the beginning of the accepted answer says that

```

Random r = new Random();
You could use e.g. r.nextInt(101)

```

This is not the right solution in our case because it would return a number between 0 and 100 (inclusive), while we want a number between 1 and 100 (inclusive). Also, in general r is not a good name for a variable; *rnd* or *random* are better choices in this case.

If you implemented the methods correctly, you should see

MAKE:

```
OK (8 tests)
```

OR

Eclipse:

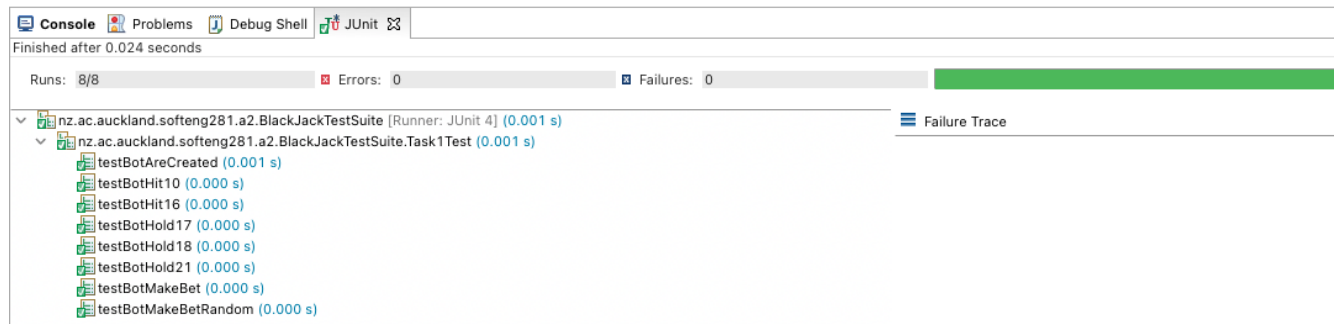


Figure 3: all test pass - Task 1

If you have passed all the test cases provided to you, then you can be rest assured you have earned all the marks for Task 1!

Run `make play`, or if you are using Eclipse run the `BlackJack` class. You will see the bots in action. Are they playing better?

Now you are ready to move onto Task 2.

3.2 BotDealer[Task 2]

Once Task 1 is completed, you will implement the bot that will act as the dealer of the game. Your goal (as the human player), and the goal of the other bot players, is to defeat the dealer.

Go to the top of **BlackJackTestSuite.java**, and uncomment `BlackJackTestSuite.Task2Test.class`:

```
@RunWith(Suite.class)
@SuiteClasses({
    BlackJackTestSuite.Task1Test.class,
    BlackJackTestSuite.Task2Test.class,
    //BlackJackTestSuite.Task3Test.class,
    //BlackJackTestSuite.Task4Test.class,
    //BlackJackTestSuite.YourTests.class
})
```

As soon as you compile and run your code with the `make` command (or if you are using Eclipse run the `BlackJackTestSuite` class), you will notice failing test cases.

First, you need to add the dealer to the `Game`. The name must be "Dealer". You should correctly set the instance field `dealer` in the class `BlackJack`. Then you need to let the dealer play after all the players have finished their turn (but before the invocation of the method `checkWinner()`).

If you did so, the first failing test will pass. Run make play, or if you are using Eclipse run the Blackjack class. You will see that the dealer is also playing the game.

Now open the class BotDealer.

```
package nz.ac.auckland.softeng281.a2;

import java.util.List;
import java.util.Random;

/**
 * you should change this class for TASK 2
 */
public class BotDealer extends Participant {

    private List<Participant> players;

    public BotDealer(String name, List<Participant> players) {
        super(name);
        // ADDHERE
    }

    @Override
    public Action decideAction() {
        // TODO
        return new Random().nextBoolean() ? Action.HIT : Action.HOLD; // FIXME
    }

    @Override
    /**
     * do not touch this method
     */
    public int makeABet() {
        // the Dealer doesn't bet so is always zero
        return 0;
    }
}
```

The dealer is making random decisions, which is not very smart. Instead, we want that the dealer decides Action.HIT if its current hand is losing against at least 2 players' current hands (recall that we have three players in total: you and two bots), otherwise, the dealer decides Action.HOLD.

Note that the Dealer never makes a bet. So we will not implement the makeABet method.

A player wins against the dealer (i.e, the dealer loses against a player) if the player has blackjack **or any** of these conditions are met:

- the player has a total score ≤ 21 , while the dealer is busted (> 21).
- both the player and the dealer have a total score ≤ 21 , but the score of the player is greater ($>$) than that of the dealer.

You can notice that the constructor of the BotDealer takes as an input the name (Dealer) and the list of players playing the game. Then, the dealer can easily get the current hands of the players and check how many players will defeat if it decides to hold (i.e. to terminate the hand).

You can implement the logic as you wish.

After you implemented the logic of the dealer, all tests should pass. If you have passed all the test cases provided to you, then you can be rest assured you have earned all the marks for Task 2!

You can run `make play`, or if you are using Eclipse run the `BlackJack` class. You will see the dealer making reasonable decisions.

Now you are ready to move onto Task 3.

3.3 Game outcome *[Task 3]*

Now we need to implement a method that tells us after each game which player won against the dealer.

Go to the top of `BlackJackTestSuite.java`, and uncomment `BlackJackTestSuiteTestSuite.Task3Test.class`:

```
@RunWith(Suite.class)
@SuiteClasses({
    BlackJackTestSuite.Task1Test.class,
    BlackJackTestSuite.Task2Test.class,
    BlackJackTestSuite.Task3Test.class,
    //BlackJackTestSuite.Task4Test.class,
    //BlackJackTestSuite.YourTests.class
})
```

As soon as you compile and run your code with `make` command (or if you are using Eclipse run the `BlackJackTestSuite` class), you will notice failing test cases.

You need to implement the method `checkWinner()` in the `BlackJack.java` file. The expected behavior is that it has to print `System.out.println(player.getName() + " wins")`; for each player that won against the dealer in the current hand.

Note that for Task 3, we provided only half of the test cases that we will use for marking. As such, if all the provided tests pass you are sure that you will get **at least** half of the points assigned to Task 3. You are encouraged to write additional test cases to test your code. However, additional test cases will not be marked.

3.4 Game statistics *[Task 4]*

Now you need to implement a method that tells who is the best player considering all the hands.

Go to the top of `BlackJackTestSuite.java`, and uncomment `BlackJackTestSuiteTestSuite.Task4Test.class`:

```
@RunWith(Suite.class)
@SuiteClasses({
    BlackJackTestSuite.Task1Test.class,
    BlackJackTestSuite.Task2Test.class,
    BlackJackTestSuite.Task3Test.class,
    BlackJackTestSuite.Task4Test.class,
    //BlackJackTestSuite.YourTests.class
})
```

As soon as you compile and run your code with `make` command (or if you are using Eclipse run the `BlackJackTestSuite` class), you will notice failing test cases.

You need to implement the method `printPlayerHighestGain()` that prints the player (among `Player1`, `Bot1`, and `Bot2`) with the highest gain.

More specifically, the expected behaviour is the following:

`printPlayerHighestGain()` prints `System.out.println("The player with the highest gain is: " + name + " with " + totalGain + " chips")`; where `name` is the player name with the highest gain considering **all** the hands played in the game. If two or all players have the same highest gain, the method prints one of them. `totalGain` is the amount earned by the player with the highest gain.

For example, let's assume a player makes 5 hands betting the following chips: 50, 100, 10, 20, 30. Let's assume that they win in the second and third hands (the second hand won with a blackjack). The **totalGain** is $-50 + 150 + 10$

$-20 - 30 = 60$. Basically, you sum what you earn and subtract what you lost. If a player loses more than they win, the `totalGain` will be a negative number.

Note that for Task 4 we provided only half of the test cases that we will use for marking. As such, if all the provided tests pass you are sure that you will get **at least** half of the points assigned to Task 4. You are encouraged to write additional test cases to test your code. However, additional test cases will not be marked.

Important: code style

Although for this assignment your code will **not be marked** for good programming practices, we encourage to follow standard Java naming conventions and code style as this will be checked in future assignments. In particular:

- **method names:** Methods should be verbs, in camel case with the first letter lowercase, with the first letter of each internal word capitalised (e.g., `decideAction`).
- **variable names:** Variable names should be nouns in camel case with the first letter lowercase, and with the first letter of each internal word capitalised (e.g., `initialAmount`). Variable names should not start with underscore `_` or dollar sign `$` characters, even though both are allowed. Variable names should be short yet meaningful. One-character variable names should be avoided except for temporary "throwaway" variables (e.g., iterator `i` of a `for` loop).
- **correct indentation and brace placement:** You should enforce correct indentation of your Java classes.
- **code comments:** Comments in the code should give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. Do not comment every single line of code (!), only where necessary.
- **avoid duplicated code:** If you find yourself copy-and-pasting your code in different parts of your Java class, consider creating a private method to avoid duplicated code and promote code reuse. This improves maintainability because when code is copied, bugs need to be fixed at multiple places, which is inefficient and error-prone.
- **private fields:** Make all fields private. This ensures consistency of the member data because only the owning member can make changes to it. If external classes need to read or modify the fields, use getter and setter methods.

Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.
- Although you may add more methods to the classes, you must leave unchanged the signature of the original methods.
- Do not move any existing code files to a new class, file, or directory.
- You may add more tests as you please. To do so, uncomment `BlackJackTestSuite>YourTests.class` and add more tests inside public static class `YourTests`. This file will not be marked, it's just for you if you wish to make more test cases. Be aware that your code must still work with the original version of the `BlackJackTestSuite.java` file, which we will reuse as originally shared with you.

Frequent Submissions (GitHub)

You **must** use GitHub as version control for all assignments in this course, starting with this assignment. To get the starting code for this assignment, you must accept the GitHub Classroom invitation that will be shared with you. This will then clone the code into a **private** GitHub repository for you to use. This repository must remain private during the lifetime of the assignment. When you accept the GitHub assignment, you can clone the repository either to your own computer or work on it via replit.com. For the latter, a Repl badge/image is automatically added to the README.md file when you accept the GitHub invitation. By clicking on this, you will be able to run your code in the online replit.com IDE.

As you work on your assignment, **you must make frequent git commits**. If you only commit your final code, **it will look suspicious and you will be penalised**. In the case of this assignment, you should aim to commit your changes to GitHub every time you pass a few test cases (at the very least, once for every task completed). You can check your commits in your GitHub account to make sure the frequent commits are being recorded. **Using GitHub to frequently commit your changes is mandatory in this course**. In addition to teaching you to use a useful software development skill (version control with git), it will also protect you two very important ways:

1. Should something terrible happen to your laptop, or your files get corrupted, or you submitted the wrong files to Canvas, or you submitted late, etc, then you are protected as the commits in GitHub verify the progress in your assignment (i.e. what you did and when), and
2. If your final code submission looks suspiciously similar to someone else (see academic honesty section below), then GitHub provides a track record demonstrating how you progressed the assignment during that period.

Together, GitHub is there to help you.

Submission

You will submit via Canvas. **Make sure you can get your code compiled and running via command line** when running make in replit.com. Submit the following, in a single ZIP archive file:

- A **signed and dated Cover Sheet** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can generate and download this in Canvas, see the Cover Sheet entry.
- The entire contents of the **src** folder you were given at the start of the assignment, including the new code you have written for this assignment. Ensure you **execute make clean before zipping the folder** so your submission does not include any executable files (your code will be re-built for marking).
- Do NOT nest your zip files (i.e. do not put a zip file inside a zip file).

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions. Get into the habit of downloading them again, and double-checking all is there.

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help (struggling a little with it will help you learn, especially if you end up solving it). If you still need help, check on Canvas (if it is about interpreting the assignment specifications) or ask in the Lab help clinics (if you would like more personal help with Java). Under no circumstances should you take or pay for an electronic copy of someone else's work.
- **All submitted code will be checked using software similarity tools**. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.

- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
 - Always do individual assignments by yourself.
 - Never show or give another person your code.
 - Keep your Repl workspace private, and do not share your Repl with anyone.
 - Never put your code in a public place (e.g. Reddit, public GitHub repository, forums, your website).
 - Never leave your computer unattended. You are responsible for the security of your account.
 - Ensure you always remove your USB flash drive from the computer before you log off.
 - Frequently commit your code to GitHub. This provides a track record of your work and will allow the teaching team to follow your footsteps as you completed your assignment. If you do not frequently commit your code, it will look suspicious.

Late submissions

Late submissions will incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)