[[1]](#footnote-1)

# Data Structures Essentials

## Assignment 2: GameStonk Share Trading

# Background knowledge

In this assignment you will develop a *very* basic stock market with a stock exchange, brokers that process trades, and trades for different companies’ stocks. In this assignment we have:

* A **listed company** is a company whose shares are bought and sold. Each company has a name, a company code that is an abbreviation of their name, and the current price of their shares.
* A **trade** object contains the number of shares to buy/sell, which company those shares need to be from, and which broker will process that trade.
* A **broker** takes trades from individual users, and puts them in single a queue to process. Because brokers are experts at what they do, they also have a “watchlist” of which companies they recommend people buy shares for. Whilst trades should be processed on a first-in, first-out queue, some unethical brokers might decide to delay when certain trades are processed by putting them at the back of the queue so they could process their own trades first. This is what we’ll be looking at in this assignment.
* A **stock exchange** has a collection of brokers that can buy and sell stocks on it. Each time the stock exchange processes trades, it asks each broker for the next trade in their queue to process. The exchange then processes that trade which causes the company on the trade’s share price to go up or down. This process is then repeated to processes additional trades.

## What you will learn in this assignment

This assignment will help you understand the following concepts:

* Creating your own linked lists
* Basic use of Priority queues
* Basic use of maps/hash maps
* compareTo() for comparing objects
* Basic exceptions
* Unit testing and Junit

# Academic Integrity

Any work submitted must represent your own knowledge and efforts. This means that whilst you can use Eclipse’s auto-generation functions for making functions, you **cannot** use Chat GPT or other larger code-generation tools, or get direct assistance from others to write your code.

As part of your git history, it should show your knowledge and refactoring of the code to improve it over time, showing that the project submitted was your own work, and was developed over a period of time.

If we have concerns regarding if your submission is your own, we may interview you about your submission and knowledge of the code and how it works, and how to generate similar code.

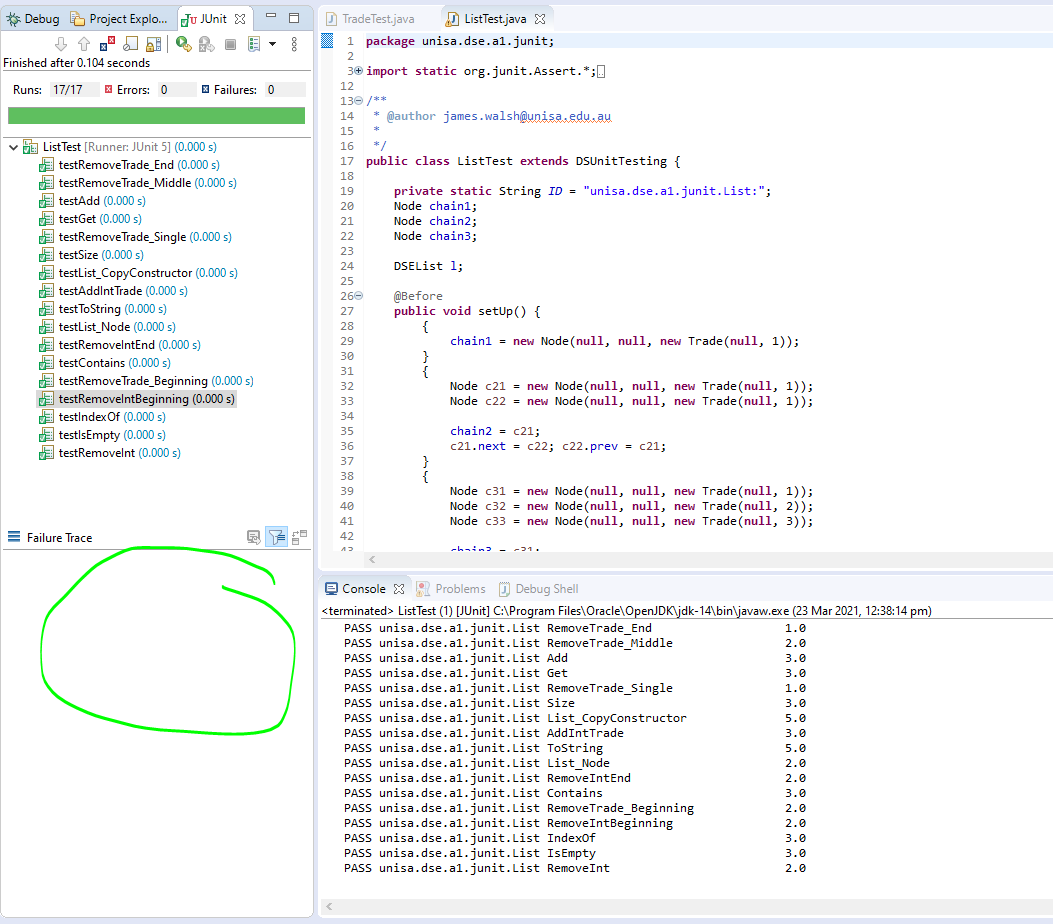
Whilst you can look at other materials and resources online to gain understanding and knowledge, the code written must be your own and represent how you think the code should function. Whilst you can talk to other students about your submission at a high level (functions, ideas, class relationships, etc), you are not allowed to share code.

# Getting Started

This assignment is an Eclipse project with existing class files that you will be required to complete in varying forms by changing method bodies, return types, creating getters, and setters. In addition to the classes you will be editing, there are a number of Junit tests that are used by a marking program to give you and indication of how well your code is working. Any changes you make to the class files listed below should be in-line with this requirement specification, any documentation in the code itself, and passing the tests. You do not need to edit any of the test files, or the marker, we will be using our own version of them for marking. However, you may find it useful to edit the test files for debugging purposes.

## Getting it running

1. Download the ZIP from the course web page and unzip.
2. In Eclipse > File > Open Projects from File System and open the project
3. There are two ways you can run the tests:
   1. The first will run the entire marking program and give you an overall score and marks. Go into the AssignmentMarker.java file in the Junit package and run the file. The console will show you the output from the test marker. Note that you’ll see lots of things going wrong and it doesn’t run completely! That’s ok, we’ll work on that. As you complete more of the code, more of the assignment marker will be able to run.
   2. The second way is to go into one of the Junit test files specifically, e.g. open up “ListTest.java” and hit run. You should see Junit tests appear in your window and console, showing the output of each test run:



In the bottom left section of the screenshot above it will show a stack trace for any test you click on that failed. This will be what you can use to start debugging what’s happening.

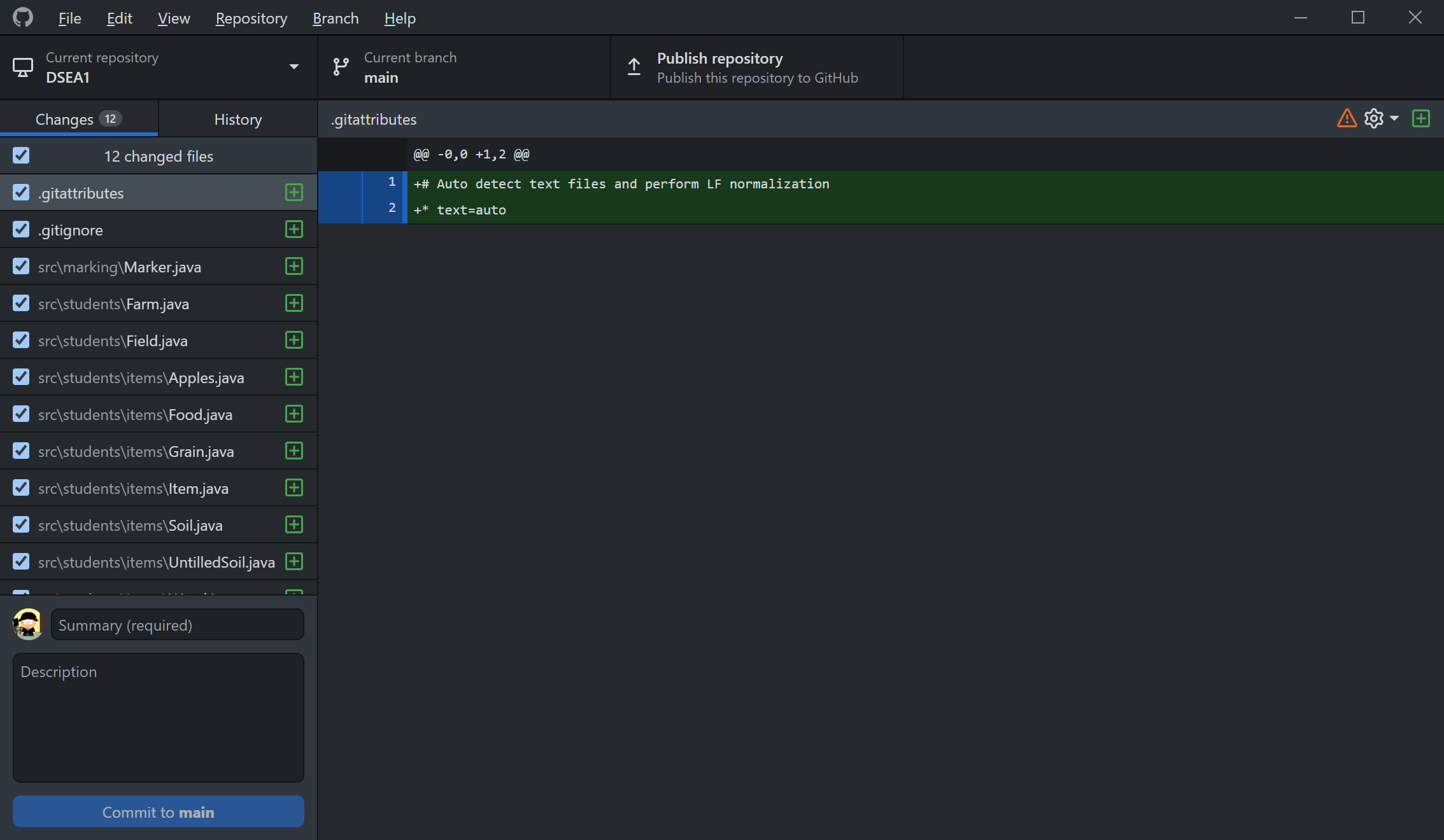
Using the above, you can run the whole marker, or just an individual set of test for a specific class, which makes it easier to focus on one class at a time. As you go through and complete functionality in each class, run the tests for that class to see how successful you were in passing those tests.

Please note that whilst we have provided SOME tests, it does not mean the tests cover everything. Additional tests will be used to mark you assignment, so please review your code and make sure it’s not just passing the tests you have, but also handling any other scenarios it should be.

## Getting it into GitHub Desktop

GitHub Desktop is GitHub’s tool for managing code using [Git](https://en.wikipedia.org/wiki/Git), which is a source control tool that, in short, allows you to take snapshots of your code as you, and easily merge it with what others are working on. For this assignment, you’ll just be using git locally on your own machine to take commits (i.e. a snapshot) of your code as you go through development.

1. Download and install GitHub Desktop from <https://desktop.github.com/>
2. One installed and open, go File menu > “Open Local Repository”
3. Open the root of the A2 folder you downloaded. You might see a “.git” folder in your download, you want to open the folder that contains this “.git” folder.
   1. If you get a warning about “This directory does not appear to be a Git repository” you’ve selected the wrong folder.
4. You should now see the below.



1. There are no commits in your git yet. It’s a good idea to make an initial commit of your clean project. Make sure all your files are selected, and down the bottom in the “Summary” field, type “Initial Commit”. You can then, if you wish, add extra information about what might have changed since the last time you committed code.
2. Then press “Commit to main” button.
3. You will notice that all the files in the left side of the app have disappeared. This is because that panel shows new changes only. When you change files, they’ll appear there for you to commit.
4. In the top left you can now switch into the “History” tab and see your previous commit, what files it contained, and what changes were in those files.

Graphical user interface, text, application

Description automatically generated

1. You can, if needed, revert back to any of your prior commits if you make a mistake and want to “undo” it, however BEWARE! If you do rollback, make sure you don’t lose work you want to keep!

You MUST commit your code at regular intervals (from step 5 above). This will teach you to commit your code as you get things working. What you commit doesn’t have to be perfect, and you can refine your code later on, but it is important to commit your work as you go so that you have snapshots showing it’s development over time, and your trial-and-error as you learn and improve on it.

**Your git repository and its history are part of your marks!** Make sure to commit each time you get things to a good point.

There are lots of tutorials online for using GitHub Desktop, with their official documentation for making commits available at <https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/making-changes-in-a-branch/committing-and-reviewing-changes-to-your-project>.

# Completing the assignment

Please review the below, along with the actual source files comments, and the tests themselves for additional information.

You may be required to change the parameter types or return types of function calls as part of the assignment. You’ll need to use your understanding of how the code should operate to make the required changes.

## Step One – DSEList.java

In part 1, you will create your own implementation of the Java LinkedList Collections class. There is an «interface» provided for List. In Step Two, you will then use your Linked List implementation as the basis for making a generic version that can store any type of object.

DSEList will extend the List class defined in List.java. The implementation will be a double-linked list and must implement the abstract methods from List.java.

DSEList should have one data member: public Node head. Others can be added if you require them.

The Node objects used by the list store basic String objects.

Implement the following methods in the List class:

* Constructor: implement a blank constructor which takes no parameters.
* Constructor: implement a constructor accepting one Node (containing a String object). The constructor should set head to the given Node.
* Copy constructor: implement a copy constructor accepting a DSEList object. The copy constructor should perform a deep copy of the DSEList passed to the constructor: the new DSEList should not contain references to the Node objects in the second DSEList. (The two DSELists should be independent: changing the contents of Node objects in one DSEList should not affect the other).
* public boolean add(String obj): The add method should append the specified object to the end of the List.
* public boolean isEmpty()
* public int size()
* public String toString(): this should return a String created by concatenating each Nodes toString(). A single space: ‘ ’ should be inserted between each Nodes toString(). No trailing space should be inserted. For example, if the list contains 3 Node objects, an appropriate toString() return value could be ‘1 2 3’, but not ‘123’ or ‘1 2 3 ’ [note the trailing whitespace]. For further details, refer to the unit tests supplied with the assignment.
* public boolean equals(Object other): two DSEList objects are equal if they contain the same Strings in the same order.

## Step Two – DSEListGeneric.java

The second part of the assignment is to take the code you have written for your list and make it generic. This will allow the list’s nodes to store ANY objects, not just Strings. **You should have to write almost no code for this step,** instead you should only need **copy and refactor your existing code** from the functions in DSEList.java into DSEListGeneric.java, and make very small changes to it to enable generics. The generic list should use the NodeGeneric class for its nodes.

DSEListGeneric should support the same functions as above for DSEList, however any references to the String type that the list stores should be replaced with the generic type that’s passed in when the generic list is created at runtime. Again, you should have to add no additional logic for this step, rather you are just copying and refactoring your existing DSEList functions into DSEListGeneric with very minor changes to the signature of methods and their contents. As a hint for converting a class from a non-generic class into a generic class, compare the Node and NodeGeneric classes. They achieve the same thing, however the second one supports generics.

## Step Three

Now that we’ve got the ability to store a list of things, we can start to build out the rest of the trading simulator. Generally you should be able to go through this list top-to-bottom in order when implementing things, however occasionally you may need to complete or at least start a function listed later in the document if it’s required by a function you’re trying to complete.

### ListedCompany.java

A listed company is a company that can have its shares bought and sold on a securities exchange.

* public String getName(): public getter for "name";
* public String getCode(): public getting for "code"
* public int getCurrentPrice(): public getter for "currentPrice"
* public ListedCompany(String code, String name, int currentPrice): Should store the three parameters into the instance variables
* public int processTrade(int quantity): should increase or decrease the value of the currentPrice variable depending on the quantity of stock as the parameter. The price should increase by "quantity / 100" amount, and never drop below 1 in price. For a “sell” the quantity will be negative (price goes down), and positive for a buy (price goes up).

### StockBroker.java

Stock brokers take trade orders on behalf of others and process the trades on the securities exchange. The broker must track all their pending orders so they know which trade to process next. Brokers also track a watchlist of companies they advise their clients to purchase, however some dodgy brokers may encourage users to buy a certain stock, but then not process their trades on time as expected!

* private PriorityQueue<Trade> pendingTrades: Should be an instance variable using Java’s PriorityQueue class to store Trade objects.
* private DSEListGeneric<String> watchlist – Should be an instance variable of the DSEListGeneric class to store objects of type String.
* public DSEListGeneric<String> getWatchlist() – Public getter for watchlist. It should return a new list, rather than the current list. Modifying the list returned by getWatchlist (e.g. removing an item) should not affect the original version of the list held by the StockBroker.
* public Boolean addWatchlist(String companyCode) – Adds a company code to the watch list. Return false if item is already in the list or is null, true otherwise if added to the list
* public String getName() – Gets the broker’s name.
* public StockBroker(String name) – Create a broker with given name.
* public boolean placeOrder(Trade order) – Adds the Trade to the pendingTrades list if it's not null and not already in there.
* public Trade getNextTrade()
* public int getPendingTradeCount()

### Trade.java

Trade objects represent a specific number of shares to be bought in a specific company. Each trade object is also associated with the stock broker who will be processing that trade.

* public Trade(StockBroker broker, String listedCompanyCode, int shareQuantity): Should store the three parameters into the instance variables
* public int compareTo(Trade other): Compare this trade with another trade. Please see JavaDoc in code for more informationStep Four (Optional) – Command Line Interface

### UntradedCompanyException.java

This is class should be an Exception that can be thrown when an unknown company code is used.

* public UntradedCompanyException(String companyCode): This should allow any exception thrown using this class to show the message “TSLA is not a listed company on this exchange”, assuming the companyCode “TSLA” was passed in as the parameter.

### SecuritiesExchange.java

* public SecuritiesExchange(String name)
* public boolean addCompany(ListedCompany company)
* public boolean addBroker(StockBroker broker)
* public int processTradeRound() throws UntradedCompanyException

## Optional Step Four (no marks allocated)

No marks are allocated for this final step and it is completely optional. You can still obtain a 100% for the assignment without completing this final section.

SecuritiesExchange has an additional function runCommandLineExchange(), that is sent a Scanner object. If you’re looking to push yourself a little further, this should act as a stub where you can develop your own list of commands to be processed by the exchange to add trades to brokers, process a trade round and exit. Whilst the stub for this method is given to you, as is a simple setup to automate testing of commands from a file, they will need to be extended by you. Once you can process commands from command line or a text file, who knows what’s next?!?!?! Accepting trade orders over a network?

* public int runCommandLineExchange(Scanner sc)

# Marking scheme

Ensure clean, consistent coding style. There is no official style guide for the assignment other than using the standard Java conventions. We can only award marks for code you have written, so if you don’t have any code, you don’t get any marks for style.

Your code should be commented and be easy for us to follow and understand, without too many comments. If it is not easily apparent what something is doing, ensure you comment what it’s doing and why. We can only award marks for code you have written, so if you don’t have any code, you don’t get any marks for commenting.

Marks summary is below. As part of the marking scheme, we will also be reviewing your Git history for that part. **If you haven’t committed that section/class appropriately, for example you just a large, fully functional and complete class in one commit, you may lose up to 50% of the marks for that element.** Smaller/one-line functions obviously can only be committed once complete, which is perfectly fine, but for your larger elements, we need to be able to see it’s history and how it has evolved.

|  |  |
| --- | --- |
| **Code passes required tests**  Your code passes the tests we run on it. Whilst you are given some tests for the classes, we may use additional tests for the final marking. The proportion of marks per class will be similar to the assignment marker you are given. This means the majority of the marks will still be for the List and ListGeneric classes. Please see the marking program for further breakdown of marks. | 80% |
| **Code style**  Clean, consistent coding style. There is no official style guide for the assignment other than using the standard Java conventions. We can only award marks for code you have written, so if you don’t have any code, you don’t get any marks for style. | 10% |
| **Code commenting**  Your code should be commented and be easy for us to follow and understand, without too many comments. If it is not easily apparent what something is doing, ensure you comment what it’s doing and why. We can only award marks for code you have written, so if you don’t have any code, you don’t get any marks for commenting. | 10% |
| **Total** | 100% |

# Submitting

Please ZIP your entire root folder of the project and upload to LearnOnline. This folder should include your src directory.

**We must receive your .java source files from the “src” folder in order to mark any code.**

## Due date

Please see the course website for the due date. Unless you have an extension, there is **NO LATE SUBMISSION**. Late submission without an extension will result in a mark of 0.

Please regularly upload a snapshot of your code to the site so that we have a copy. This means if something does happen and you can’t submit your latest version, we still have an earlier version to mark.

## Extensions

If you need an extension for an on-going issue, e.g. medical, access plan, etc., please submit a request via LearnOnline to be reviewed. If you have any issues please email me directly via email. **Having other assignments or work commitments is not a valid reason for extension.**

1. Image courtesy of <https://wave.fr/reddit-gamestope-bourse-300900> [↑](#footnote-ref-1)