Advanced Software Engineering (CSSE7023)
Assignment 2 — Semester 1, 2024

School of EECS
The University of Queensland

Due May 24th 16:00 AEST

I hear and I forget, I see and I remember, I do and I understand.
— Confucian Aphorism

Revision 1.0

Overview This assignment builds on Assignment 1 and will extend your practical experience developing a Java program, including a GUI. Additionally, you must develop and submit JUnit tests for some of the classes in the implementation. You are encouraged to write tests for *all* your classes and for the GUI part of your implementation as well, but you do not have to submit these. You will be assessed on your ability to

- implement a program that complies with the specification,
- develop JUnit tests that can detect bugs in class implementations,
- and develop code that conforms to the style conventions of the course.

Task In this assignment, you will modify and extend the SheeP spreadsheet application from Assignment 1. You will be provided with an extended version of a working implementation of SheeP in which the GUI is implemented using the Java Swing library. For the first part of this assignment, you will have to implement a new GUI that uses the JavaFX library, without altering the look and feel of the user interface. You then have to extend the functionality of SheeP in two ways: (1) provide support for two built-in functions `MEAN` and `MEDIAN` that calculate the arithmetic mean and median of a list of expressions, including an updated parser, and (2) provide a menu with options to Save the current spreadsheet in a pre-defined format to a file and Open a previously saved spreadsheet. In addition, you will be required to develop JUnit test cases for the classes that implement the `MEAN` and `MEDIAN` functionality and for the `ComplexScanner` class that has been provided to you.

Common Mistakes Please carefully read Appendix A. It outlines common and critical mistakes which you must avoid to prevent a loss of grades. If at any point you are even slightly unsure, please check as soon as possible with course staff.

Plagiarism All work on this assignment is to be your own individual work. By submitting the assignment you are claiming it is entirely your own work. You *may* discuss the overall general design of the application with other students. Describing details of how you implement your design with another student is considered to be **collusion** and will be counted as plagiarism.

You may **not** copy fragments of code that you find on the Internet to use in your assignment. Code supplied by course staff (from *this* semester) is acceptable, but must be clearly acknowledged as described in the next paragraph.

You may find ideas of how to solve problems in the assignment through external resources (e.g. StackOverflow, textbooks, ...). If you use these ideas in designing your solution you ***must*** cite them. To cite a resource, provide the full bibliographic reference for the resource in file called `refs.md`. The `refs.md` file **must** be in the root folder of your project. For example:

```
> cat refs.md
[1] E. W. Dijkstra, "Go To Statement Considered Harmful," _Communications of the ACM_,
    vol 11 no. 3, pp 147-148, Mar. 1968. Accessed: Mar. 6, 2024. [Online]. Available:
    https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html
[2] B. Liskov and J. V. Guttag, Boston: _Program development in Java: abstraction,
    specification, and object-oriented design_. Boston: Addison-Wesley, 2001.
[3] T. Hawtin, "String concatenation: concat() vs '+' operator," stackoverflow.com,
    Sep. 6, 2008. Accessed: Mar. 8, 2024. Available:
    https://stackoverflow.com/questions/47605/string-concatenation-concat-vs-operator
>
```

In the code where you use the idea, cite the reference in a comment. For example:

```java
/**
 * What method1 does.
 * [1] Used a method to avoid gotos in my logic.
 * [2] Algorithm based on section 6.4.
 */
public void method1() ...

/**
 * What method2 does.
 */
public void method2() {
    System.out.println("Some " + "content."); // [3] String concatenation using + operator.
}
```

You must be familiar with the university's policy on plagiarism.

`https://uq.mu/rl553`

If you have questions about what is acceptable, please ask course staff.

Generative Artificial Intelligence You are required to implement your solution on your own, ***without*** the use of generative artificial intelligence (AI) tools (e.g. ChatGPT or Copilot). This is a learning exercise and you will harm your learning if you use AI tools inappropriately. Remember, you will be required to write code, by hand, in the final exam.

Specification

Specification documents of classes that you must implement are provided in the form of JavaDocs.

- Implement the classes and interfaces ***exactly*** as described in the JavaDocs.
- Read the JavaDocs carefully and understand the specification ***before*** programming.
- Do not change the public specification of any required class in ***any*** way, ***including*** changing the names of, or adding additional public: inner classes, inner interfaces, methods, or fields.
- You are encouraged to add additional ***private*** members, classes, or interfaces as you see fit.
- You are encouraged to add additional non-inner classes or interfaces as you see fit to the `sheep.parsing` and `sheep.ui.graphical.javafx` packages.
- JUnit 4 test cases that you submit must only test the public and protected methods as specified in the JavaDocs. They must not rely on any other members, classes, or interfaces you may have added to the classes being tested. You may create private members and other classes in your test code.

You can download the working SheeP implementation and the JavaDoc specifications for Assignment 2 from BlackBoard (Assessment → Assignment Two). You can also access the JavaDoc for Assignment 2 at the link below.

`https://csse7023.uqcloud.net/assessment/assign2/docs/`

Getting Started

To get started, download the provided code from BlackBoard. This archive includes a working version of the solution to Assignment 1 as a Swing application. Create a new *IntelliJ* project following the JavaFX Guide on BlackBoard. Extract the archive in a directory and move it into the newly created project.

Project Overview

The design of the application is essentially the same as for Assignment 1, but some classes have been added to the implementation as described below.

sheep.core Provided, as for Assignment 1.

sheep.sheets Updated from Assignment 1. You must implement `Sheet::encode()` and `SheetBuilder::load()`.

sheep.expression Provided, as for Assignment 1.

sheep.expression.basic Provided, as for Assignment 1.

sheep.expression.arithmetic Updated from Assignment 1. Note that the `Arithmetic` class is now a subclass of `Operation` (rather than `Expression`) and a sibling class of `Function`. You must implement the `Mean` and `Median` classes.

sheep.parsing Updated from Assignment 1. You must implement the `ComplexParser` class. A new class `ComplexScanner` has been provided to assist you in writing this class.

sheep.fun Provided, as for Assignment 1.

sheep.ui Provided application constraints for the ui.

sheep.ui.graphical Provided application constraints for just GUIs.

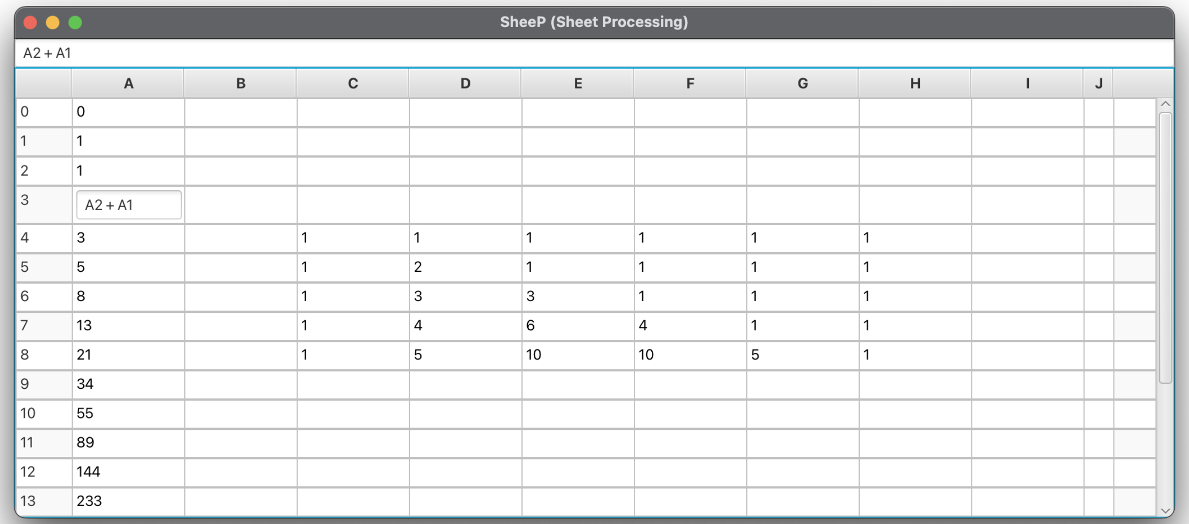
sheep.ui.graphical.swing Provided version of Assignment 1's GUI implemented using Swing.

sheep.ui.graphical.javafx You must implement a new GUI using JavaFX that has the same look and feel as sheep.ui.graphical.swing.

Stages

As for Assignment 1, this assignment will be implemented in stages to encourage incremental development. You should finish each stage before moving on to the next one. At each stage, ensure that you *thoroughly* test your implementation.

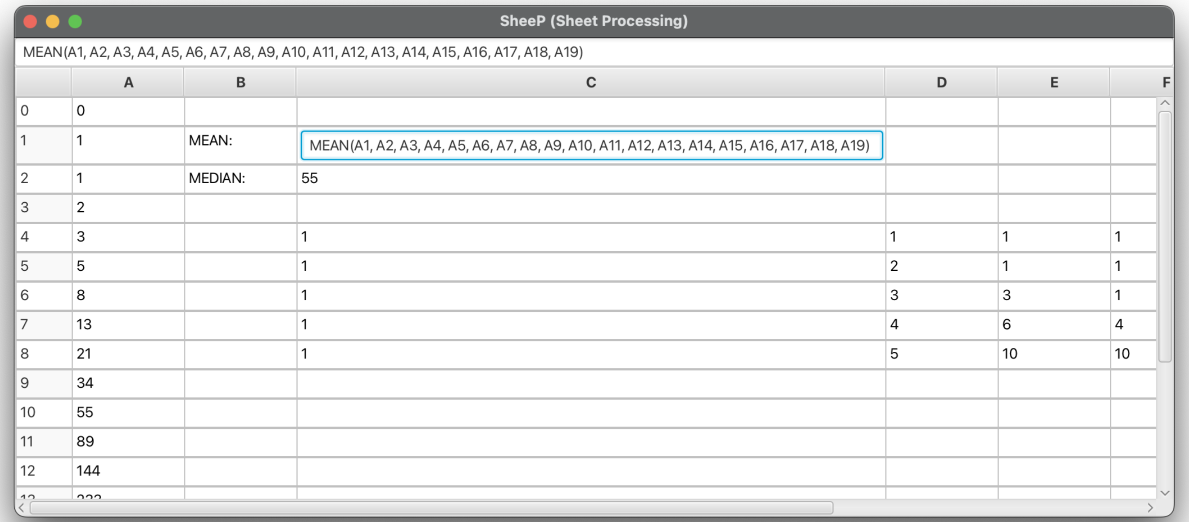
Stage 0 Organisations sometimes decide that they want to move over from one platform (such as hardware or software) to another. In the first stage of this assignment, we will mimic such a scenario and you need to change the SheeP implementation to use the JavaFX library instead of the Java Swing library. To provide backward compatibility, the program should continue to work with the Java Swing library, but any new functionality will only be implemented using the JavaFX library. A command-line argument is used in the `main` method to control which particular library is used. If the program is executed with the single argument "legacy", then the implementation will use the old GUI. If there are no arguments or any unrecognised arguments, the implementation will use new JavaFX version. After completing this stage, and with the appropriate data loaded into the spreadsheet, the SheeP implementation should display as below, irrespective of which GUI library is used.


SheeP (Sheet Processing)

A2 + A1

	A	B	C	D	E	F	G	H	I	J
0	0									
1	1									
2	1									
3	A2 + A1									
4	3		1	1	1	1	1	1		
5	5		1	2	1	1	1	1		
6	8		1	3	3	1	1	1		
7	13		1	4	6	4	1	1		
8	21		1	5	10	10	5	1		
9	34									
10	55									
11	89									
12	144									
13	233									

Stage 1 Extend the SheeP implementation to support the two built-in spreadsheet functions `MEAN` and `MEDIAN`. This will require changes to the parser. A new class `ComplexScanner` has been provided to assist you with this. To become familiar with the `ComplexScanner` class, you must implement JUnit test cases for this class. These will form part of the assessment for this assignment. You must then implement the following classes:
`sheep.parsing.ComplexParser`
`sheep.expression.arithmetic.Mean`
`sheep.expression.arithmetic.Median`
Your JUnit test cases should test all the public and protected methods specified in the JavaDoc of the classes `ComplexScanner`, `Mean` and `Median`. You must submit these tests and they *must* be located in the following files:
`test/sheep/parsing/ComplexScannerTest.java`
`test/sheep/expression/arithmetic/MeanTest.java`
`test/sheep/expression/arithmetic/MedianTest.java`
After completing this stage, and with the appropriate data loaded into the spreadsheet, the SheeP implementation should display as below.

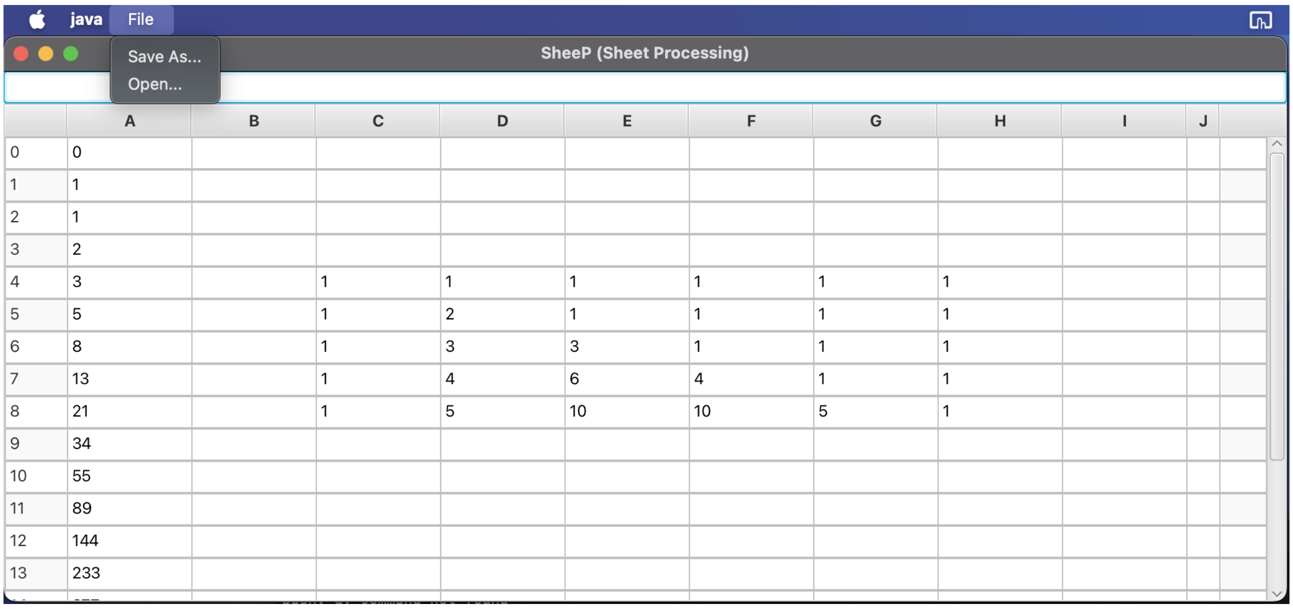


MEAN(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, A18, A19)

	A	B	C	D	E	F
0	0					
1	1	MEAN:	MEAN(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, A18, A19)			
2	1	MEDIAN:	55			
3	2					
4	3		1	1	1	1
5	5		1	2	1	1
6	8		1	3	3	1
7	13		1	4	6	4
8	21		1	5	10	10
9	34					
10	55					
11	89					
12	144					

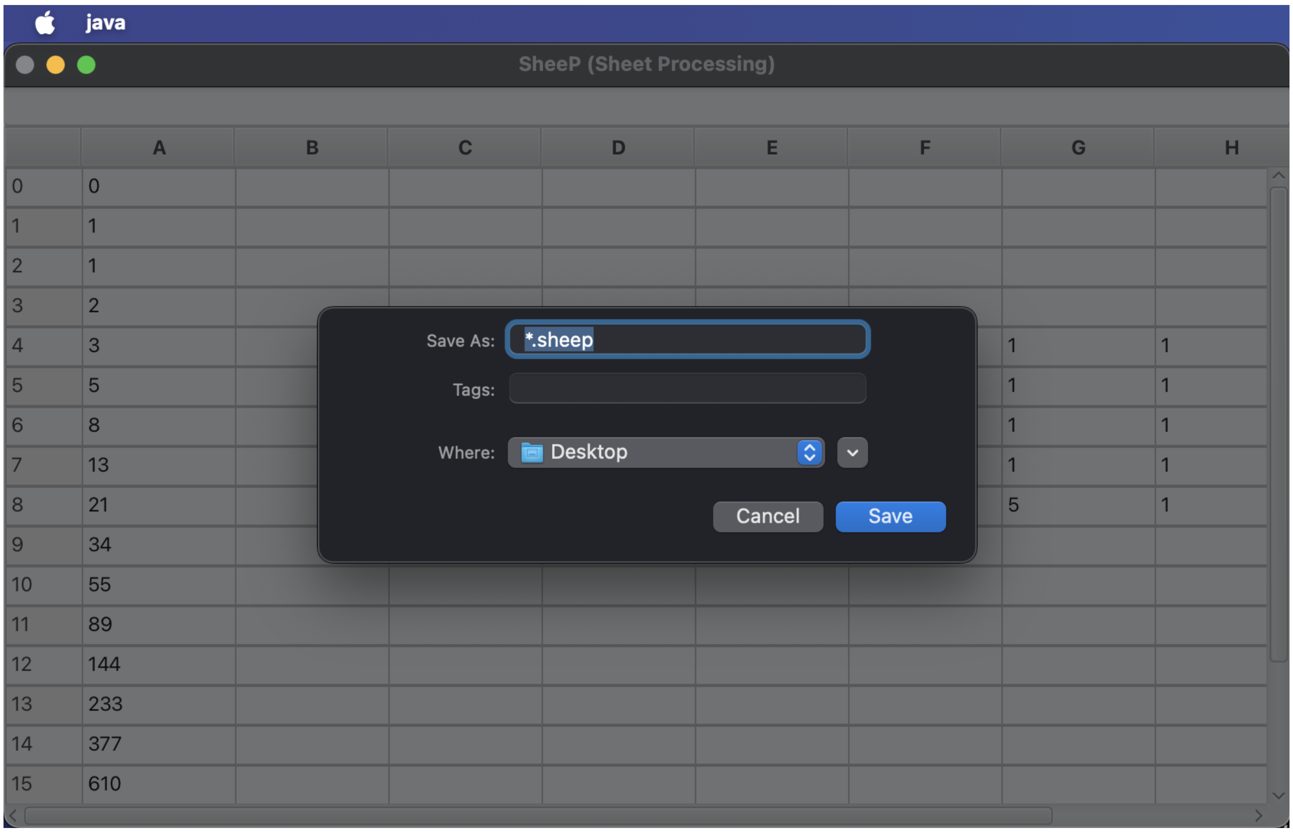
Stage 2 Extend the implementation to support a `File` menu with `Save As` and `Open` options to save the content of a spreadsheet to a file and to load a previously saved spreadsheet from a file. First you must implement `Sheet::encode()` and `SheetBuilder::load()`. Then design an appropriate menu bar for your UI to expose the features provided by `sheep.ui.UI::addFeature()`, ensuring that you provide an appropriate implementation of the `Prompt` interface. Hint: Consider using the `javafx.stage.FileChooser` class to help implement the Prompt interface. After completing this, your implementation of the `File` menu should behave as shown in the following screenshots. Note that the exact look and feel may vary depending on the operating system used to implement the functionality (the screenshots below were generated on a Mac). This is acceptable. The critical aspect is that the `Save As` and `Open` menu options should allow the user to save the current spreadsheet and open a previously saved spreadsheet to/from any file.

File: The menu bar.



	A	B	C	D	E	F	G	H	I	J
0	0									
1	1									
2	1									
3	2									
4	3		1	1	1	1	1	1		
5	5		1	2	1	1	1	1		
6	8		1	3	3	1	1	1		
7	13		1	4	6	4	1	1		
8	21		1	5	10	10	5	1		
9	34									
10	55									
11	89									
12	144									
13	233									

Save As: The save dialog.



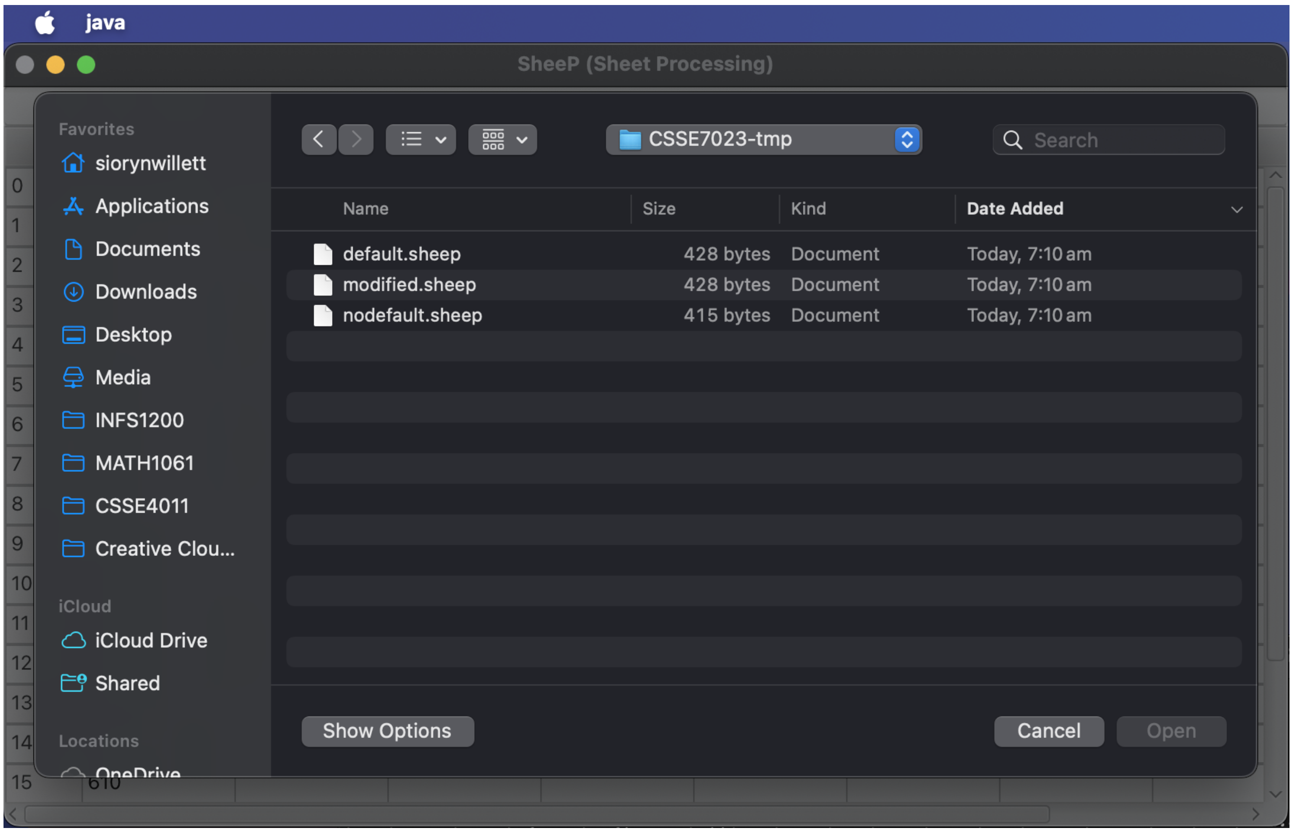
Save As: *.sheep

Tags:

Where: Desktop

	A	B	C	D	E	F	G	H
0	0							
1	1							
2	1							
3	2							
4	3						1	1
5	5						1	1
6	8						1	1
7	13						1	1
8	21						5	1
9	34							
10	55							
11	89							
12	144							
13	233							
14	377							
15	610							

Open: The open dialog.



Name	Size	Kind	Date Added
default.sheep	428 bytes	Document	Today, 7:10 am
modified.sheep	428 bytes	Document	Today, 7:10 am
nodefault.sheep	415 bytes	Document	Today, 7:10 am

Grading

Five aspects of your solution will be considered in grading your submission:

1. *Automated functionality test*: the following classes that you must implement will have JUnit unit tests associated with them that we will use to test your implementation:
`sheep/expression/arithmetic/Mean`
`sheep/expression/arithmetic/Median`
`sheep/parsing/ComplexParser`
`sheep/sheets/Sheet`
`sheep/sheets/SheetBuilder`
The percentage of test cases that pass will be used as part of your grade calculation.

2. *JUnit test cases*: your JUnit test cases will be assessed by testing both correct *and* faulty implementations of the `ComplexScanner`, `Mean`, and `Median` classes. The percentage of implementations that are appropriately identified as correct or faulty will be used as part of your grade calculation.

3. *Manual functionality test*: to ensure that the look and feel of your GUI implementation is the same as the original implementation, and to ensure that the `Save As` and `Open` menu options are properly invoked by the GUI, a small scenario with a number of steps in it will be manually executed by course staff. The faults identified during these steps will be used as part of your grade calculation.

4. *Automated style check*: as for Assignment 1, style checking on your code will be performed using the *Checkstyle* tool[1]. The number of style violations identified by the *Checkstyle* tool will be used as part of your grade calculation. **Note:** There is a plug-in available for *IntelliJ* that will highlight style violations in your code. Instructions for installing this plug-in are available in the Java Programming Style Guide on BlackBoard (Learning Resources → Guides). If you correctly use the plug-in and follow the style requirements, it should be relatively straightforward to do well for this part of the assessment.

5. *Manal style check*: as for Assignment 1, the style and structure of your code will also be assessed by course staff. However, in this case, your performance on these criteria will also be used as part of your grade

[1]The latest version of the course *Checkstyle* configuration can be found at `http://csse7023.uqcloud.net/checkstyle.xml`. See the *Style Guide* on BlackBoard for instructions on how to use it in *IntelliJ*.

calculation. It is therefore critically important that you read and understand the feedback for this part on Assignment 1, so that you can address any issues in this assignment. See Appendix B for criteria that will be used to assess the readability of your code.

Appendix B shows how the above are combined to determine your grade for the assignment.

Automated aspects of the assessment

Three aspects of assessment will performed automatically in a Linux environment: execution of JUnit test cases, running your JUnit test cases on correct and faulty implementations, and automated style check using *Checkstyle*. The environment will not be running Windows, and neither *IntelliJ* nor *Eclipse* (or any other IDE) will be involved. OpenJDK 21 with the JUnit 4 library will be used to compile and execute your code and tests. To prevent infinite loops, or malicious code, from slowing down Gradescope, any test that takes longer than 10 seconds to execute will be killed and identified as failing. All tests should execute in a small fraction of a second. Any test taking longer than a second to execute indicates faulty logic or malicious code. Similarly, any of your JUnit test cases that take longer than 20 seconds to execute on one of the correct/faulty implementations, or that consume more memory than is reasonable, will be stopped.

IDEs like *IntelliJ* provide code completion hints. When importing Java libraries they may suggest libraries that are not part of the standard library. These will ***not*** be available in the test environment and your code will ***not*** compile. When uploading your assignment to Gradescope, ***ensure*** that Gradescope says that your submission was compiled successfully.

Your code must compile.
If your submission does not compile, **you will receive a grade of one**.

Submission

Submission is via Gradescope. Submit your code to Gradescope *early and often*. Gradescope will give you some feedback on your code, but it is *not* a substitute for testing your code yourself.

What to Submit Your submission ***must*** have the following internal structure:

`src/`	Folders (packages) and `.java` files for classes that you modified or created for this assignment.
`test/`	Folders (packages) and `.java` files for the JUnit classes that you created for this assignment.
`refs.md`	File containing the references for any citations in your code.

Included in the root directory of the provided code are the files `bundle.sh` and `bundle.bat`. For MacOS and Unix users, double-click the `bundle.sh` file to execute it. For Windows users, double-click the `bundle.bat` file to execute it. This will create a `submission.zip` file for you to upload to Gradescope.

You can create the submission zip file yourself using a zip utility. If you do this, you must ***ensure*** that you do not ***miss*** any files or directories and that you do not ***add*** any extra files. We recommend using the provided `bundle` scripts.

A complete submission would contain the following files in their specified directories.

```
src/sheep/expression/arithmetic/Mean.java
src/sheep/expression/arithmetic/Median.java

src/sheep/parsing/ComplexParser.java
src/sheep/parsing/*

src/sheep/ui/graphical/javafx/SheepApplication.java
src/sheep/ui/graphical/javafx/*

test/sheep/expression/arithmetic/MeanTest.java
test/sheep/expression/arithmetic/MedianTest.java
test/sheep/parsing/ComplexScannerTest.java

refs.md

* Any number of files other than the ones specified in the JavaDoc
```

Ensure that your classes and interfaces ***correctly*** declare the package they are within. For example, `Mean.java` should declare `package sheep.expression.arithmetic;`.

Only submit the `src` and `test` folders and the `refs.md` file in the root directory of your project.
Do *not* submit ***any*** other files (e.g. no `.class` files, IDE files, or Mac *.filename* files).

Provided Files A small number of the unit tests used for assessing *functionality* will be provided in Gradescope. These will be used to test your submission, each time you upload it.

These are meant to provide you with an opportunity to receive feedback on whether the basic functionality of your classes works correctly or not. Passing all the provided unit tests does ***not*** guarantee that you will pass all the tests used for functionality grading.

Similarly, correct implementations for the `ComplexScanner`, `Mean`, and `Median` classes are used to run the JUnit tests that you submit, to ensure that tests do not fail on a correct implementation. Note that for assessing your JUnit tests we will also use faulty implementations of these classes.

The provided code on BlackBoard contains a `resources` directory. In this directory is a handful of sample spreadsheets in the correct format to be loaded by the application. Each sheet should load correctly except for `broken.sheep` which is used to demonstrate an example of a file which should fail to load. `hard.sheep` is an example of a sheet that should load correctly, but if saved will generate a different file. `hard-fixed.sheep` demonstrates what `hard.sheep` would look like if it were saved. You can use these files to check your implementation of loading and saving spreadsheet files. You should test your implementation using other spreadsheets as well.

Assessment Policy

Late Submission You must submit your code ***before*** the deadline. Code that is submitted after the deadline will receive a late penalty as described in section 5.3 of the course profile. The submission time is determined by the time recorded on the Gradescope server. A submission is not recorded as being received until uploading your files completes. Attempting to submit at the last minute may result in a late submission.

You may submit your assignment to Gradescope as many times as you wish before the due date. There will be two submission links on Gradescope, one for "on-time" submissions and one for "late" submissions. If you have an extension for the assignment, you will submit your assignment via the "late" submissions link. Your last submission made to the "on-time" submission link, before the due date, will be the one that is graded, unless you make a submission to the "late" submission link. If a misconduct case is raised about your submission, a history of regular submissions to Gradescope, which demonstrate progress on your solution, could support your argument that the work was your own.

You are ***strongly*** encouraged to submit your assignment on time, or by the revised deadline if you have an extension. Experience has demonstrated that most students who submit their assignments late lose more grades due to the late penalties than they gain by making improvements to their work.

Extensions If an unavoidable disruption occurs (e.g. illness, family crisis, etc.) you should consider applying for an extension. Please refer to the following page for further information.

`https://uq.mu/rl551`

All requests for extensions must be made via my.UQ, ***before*** the submission deadline. Do not email the course coordinator or other course staff to request an extension.

Re-Grading If an *administrative error* has been made in the grading of your assignment, please contact the course coordinator (csse7023@uq.edu.au) to request this be fixed. For all other cases, please refer to the following page for further information.

`https://uq.mu/rl552`

Change Log

Revision: 1.0

If it becomes necessary to correct or clarify the task sheet or JavaDoc, a new version will be issued and an announcement will be made on the course BlackBoard site. All changes will be listed in this section of the task sheet.

A Critical Mistakes

THINGS YOU MUST AVOID

This is being heavily emphasised here because these are critical mistakes which **must** be avoided.

Code may run fine locally on your own computer in *IntelliJ*, but it is ***required*** that it also builds and runs correctly when it is executed by the automated grading tool in Gradescope. Your solution needs to conform to the specification for this to occur.

- Files must be in the correct directories ***(exactly)*** as specified by the JavaDoc. If files are in incorrect directories *(even **slightly** wrong)*, you may lose grades for functionality in these files because the implementation does not conform to the specification.
- Files must have the correct package declaration at the top of every file. If files have incorrect package declarations *(even **slightly** wrong, such as incorrect capitalisation)*, you may lose grades for functionality in these files because the implementation does not conform to the specification.
- You must implement the public and protected members ***exactly*** as described in the supplied documentation (***no** extra public/protected members or classes*). Creating public or protected data members in a class when it is not specified will result in loss of grades, because the implementation does not conform to the specification.
  - You are *encouraged* to create private members as you see fit to implement the required functionality or improve the design of your solution.
- **Do not** import the `org.junit.jupiter.api` package. This is from JUnit 5 and may cause our JUnit tests to fail.
- Do not use ***any*** version of Java other than 21 when writing your solution. If you accidentally use Java features which are different in a version older than 21, then your submission may fail functionality tests. If you accidentally use Java features which are only present in a version newer than 21, then your submission may fail to compile.

B Grading

The following table shows how your grade for this assignment is calculated from the various components. Each of the components are explained in further detail below.

Grade	7	6	5	4	3	2	1
Functionality	$\geq 85\%$	$\geq 75\%$	$\geq 65\%$	$\geq 50\%$	$\geq 45\%$	$\geq 20\%$	any
***Checkstyle* Violations**	≤ 1	≤ 3	≤ 5	≤ 10	≤ 15	≤ 20	any
Code Style and Structure	7	≥ 6	≥ 5	≥ 4	≥ 3	≥ 2	any

The grade you receive is the highest grade for which all three conditions in the corresponding column are satisfied. For example, if your functionality score is 77%, you have 7 style violations, and your code style and structure grade is 3, your assignment grade will be a 4.

Functionality

Functionality will be calculated as a weighted percentage based on three components:

- Automated testing will be performed by JUnit and count for 60%.
- JUnit test marking will be performed by running your JUnit test cases against correct and faulty implementations and count for 20%.
- Manual testing will be performed by course staff and count for 20%.

Automated Testing

As for Assignment 1, this component of the assessment will be the percentage of JUnit tests you pass.

JUnit Test Marking

The JUnit tests you write for a class (e.g. `ClassnameTest.java`) are evaluated by checking whether they can distinguish between a:

> **correct** implementation of the respective class
> e.g. `Classname.java`, made by the teaching staff, and
>
> **incorrect** implementations of the respective class
> *"deliberately made (sabotaged) by the teaching staff"*.

This component of the assessment will be the percentage of implementations that your JUnit tests identifies *appropriately*. To identify a correct implementation *appropriately*, all JUnit tests must pass for that implementation. In other words, if one or more of your JUnit tests fail on a correct implementation, it has not been identified *appropriately*. To identify an incorrect implementation *appropriately*, at least one of the JUnit tests must fail on that implementation. In other words, if all your JUnit tests pass on an incorrect implementation, it has not been identified *appropriately*.

There are some limitations on your tests:

1. if your tests take more than 20 seconds to run, or
2. if your tests consume more memory than is reasonable,

then your tests will be stopped and the implementation will deemed not to have been identified *appropriately*. These limits are very generous (e.g. your tests should take a fraction of a second to run).

Manual Testing

To ensure that the look and feel of your GUI implementation is the same as the original implementation, and to ensure that the functionality of the Save and Open options are properly invoked by the GUI, a small scenario with a number of steps in it will be manually executed by course staff.

Marks will be allocated for each aspect implemented correctly as shown below. These marks will then be converted to a percentage that will be used in the grade calculation.

- Legacy UI still works for legacy behaviour (5 mark)
- JavaFX Appearance (10 marks)
  - Appropriate Title bar (2 marks)
  - Appropriate Formula bar (2 marks)
  - Displays a grid (2 mark)
  - Column and Row headers present and correct (2 mark)
  - Grid preferred height and width approximately respected (2 mark)
- Additional Features (10 marks)
  - `File->Save` and `File->Open` menu items present (2 marks)
  - `File->Save` works correctly for constants (2 marks)
  - `File->Save` works correctly with references and built-in functions (2 marks)
  - `File->Open` works correctly for constants (2 marks)
  - `File->Open` works correctly with references and built-in functions (2 marks)

Automated Style Checking

As for Assignment 1, the *Checkstyle* tool is used for automated style checking and the number of style violations identified for your code is used in the grade calculation as indicated in the table above. For the *Checkstyle* tool, multiple style violations of the same type will ***each*** count as ***one*** additional violation.

Code Style and Structure

As for Assignment 1, the style and structure of your code will be assessed by course staff, but for this assignment it will be incorporated into the grade calculation as indicated in the table above. Style and structure will be graded according to the rubric provided below (which is the same as for Assignment 1). This is then converted to a grade for this aspect of the assessment according to the following table.

Grade	Performance Against Criteria
7	At most one criteria rated as Proficient and all others as Advanced.
6	At most three criteria rated as Proficient, and all others as Advanced.
5	At most one criteria rated as Developing, and all others as Advanced or Proficient.
4	At most three criteria rated as Developing, and all others as Advanced or Proficient.
3	At least four criteria rated as Developing and none as Advanced, or five or more rated as Developing.
2	At least five criteria rated as Developing and none as Advanced, or six or more rated as Developing.
1	All criteria rated as Developing.

The key consideration in grading your code style is whether the code is easy to understand. Code style will be assessed against the following criteria.

Readability

- Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
- Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.)
- Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.

Documentation

- Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
- Informative JavaDoc: Every class, method and member variable4 should have a JavaDoc comment that explains its purpose. This includes describing parameters, return values, and potentially thrown exceptions so that others can understand how to use the method correctly.
- Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method, the logic should usually be clear from the code and JavaDoc. For long or complex methods, each logical block should have an in-line comment describing its logic.

Design & Logic

- Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method.
- Variable Scope: Variables should be declared locally in the method in which they are needed. Class variables are avoided, except where they simplify program logic.
- Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. well-designed loops and conditional statements).
- Encapsulation: Classes are designed as self-contained entities with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.

Rubric

Criteria	Standard		
Readability	**Advanced**	**Proficient**	**Developing**
Program Structure	Whitespace & comments highlight all blocks of logic, making it easy to follow.	Whitespace & comments highlight some blocks of logic, decreasing readability at times.	Whitespace & comments are not used well, decreasing readability in several places.
Identifier Names	All identifier names are informative and well chosen, increasing readability of the code.	Most identifier names are informative, aiding code readability to some extent.	Several identifier names are not informative, detracting from code readability.
Symbolic Constants	All, non-trivial, constant values are informative and well named, symbolic constants.	Most, non-trivial, constant values are informative, symbolic constants.	Only some, non-trivial, constant values are informative, symbolic constants.
Documentation			
Comment Clarity	Almost all comments enhance the comprehensibility of the code. Comments never repeat information already apparent in the code, nor are they verbose.	A few comments are unnecessary to code comprehension. Or, a few comments are overly verbose, reducing the ease with which code can be understood.	Many comments are unnecessary to code comprehension. Or, some comments are overly verbose, reducing the ease with which code can be understood.
Informative JavaDoc	All provided JavaDoc is replicated or improved. Accurate & informative JavaDoc is provided for all new methods & member variables.	All provided JavaDoc is at least replicated. Accurate & informative JavaDoc is provided for most new methods & member variables.	Some JavaDoc is inaccurate, unclear or absent.
Description of Logic	All important or complex blocks of logic are clearly explained or summarised. No stating of the obvious.	Most important or complex blocks of logic are clearly explained or summarised. Almost no stating of the obvious.	Some blocks of logic are poorly explained or summarised. Or, some descriptions are verbose or confusing.
Design & Logic			
Single Instance of Logic	Almost no duplicate code. Additional well designed methods modularise your code.	Some code has been duplicated. You have added some methods to modularise your code.	Large amounts of code are duplicated.
Variable Scope	All member variables are necessary parts of the class' abstraction. None should have been local to methods. Class variables have been avoided or simplify logic.	A few member variables should be local variables. Or, there are a few unnecessary local variables in methods. Class variables have been avoided.	Some member variables are unnecessary or should be local variables. Class variables have been used like modular global variables.
Control Structures	Logic is simple and clear through good use of control structures.	A small number of control structures are unnecessarily complex.	Some poorly designed control structures (e.g. excessive nesting or branching, overly complex logic, multiple unnecessary exit points, ...).
Encapsulation	All classes are independent entities with private state and public behaviour. Methods only directly access their own object's state, never modifying another's.	Most classes are independent entities with private state and public behaviour. Methods rarely directly access or modify another object's state.	Some classes have non-private member variables. Some methods directly access or modify other objects' state.