1. **Personal information:**

Hoang Xuan Gia Khanh
101769017
Data Science
2023-2024
18/04/2024

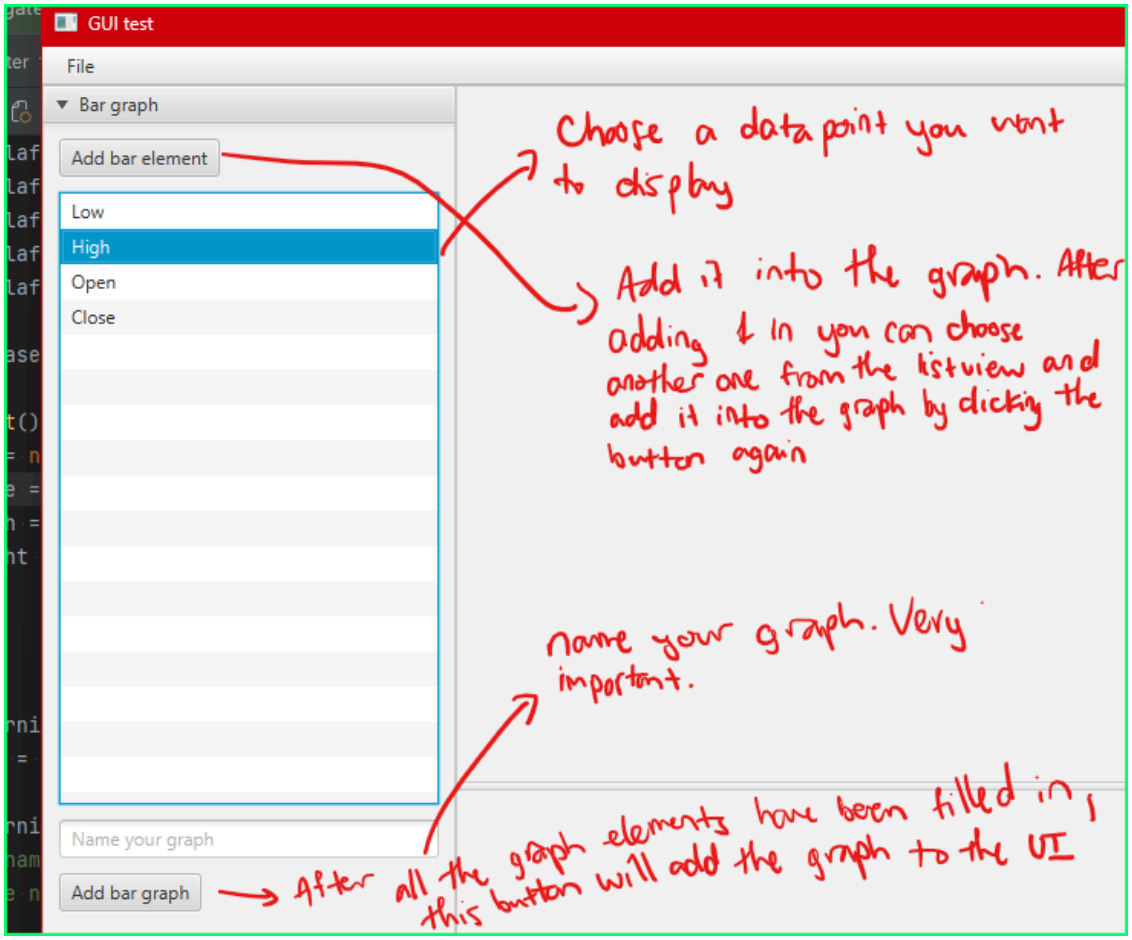
2. **General description**

In the simplest sense the program produces visual representation, dashboard, of the input stock data and lets users interact with said visual representation. The user gives the program a Json file with relevant data. Then the user can choose which data points they want to be visualized on several types of graphs and charts: scatter plot, column plot and pie chart along with supporting Cards, and the program will produce the corresponding visualization. The program can work with predefined files, which contains all of the graphs and cards so that the program can immediately display them when the user uploads them. The program also allows users to save their dashboard into a predefined file, so that the program will display the same graphs and cards the user had when they saved it.

3. **UI**

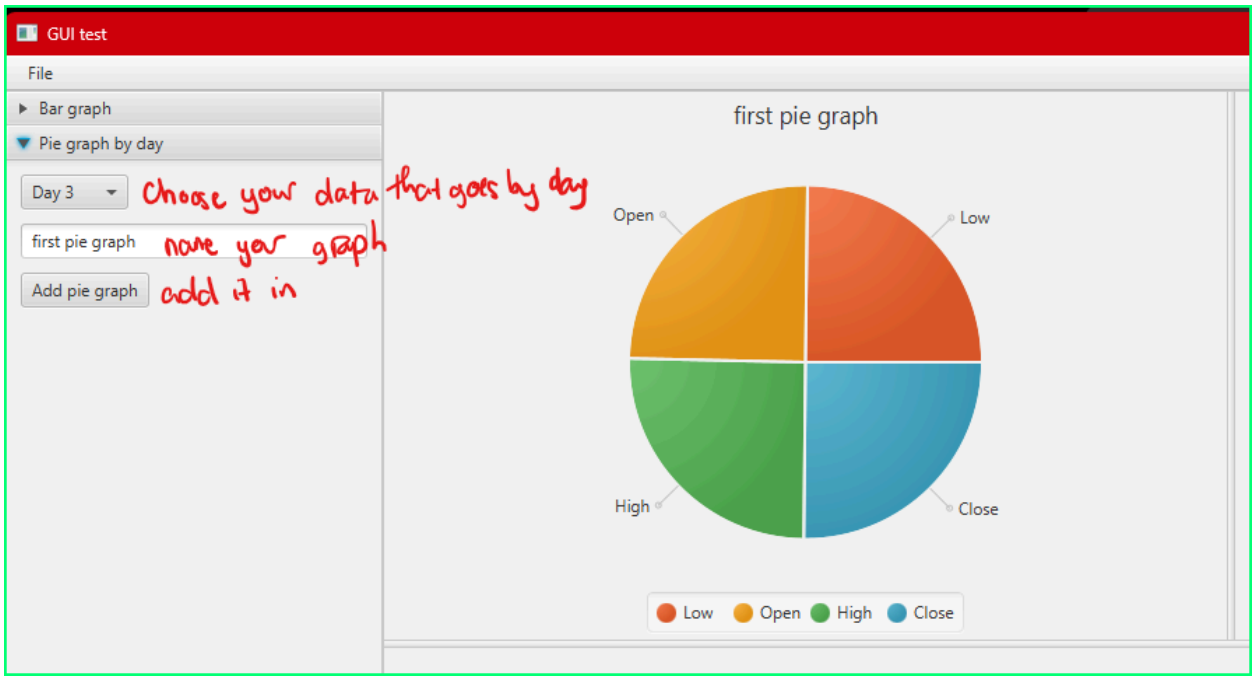
When the user start the program, it looks like this:

On the left side of the screen there is an accordion that contains all of the interaction that the user can do with their dashboard. The first 5 titledpane: Bar graph, Pie graph by day, Pie graph by type, Scatter graph and Cards contains all of the buttons, comboBox, listview, textfield that the user can use to create their graphs/cards.
Let's take a look at the Bar graph tiltedpane:

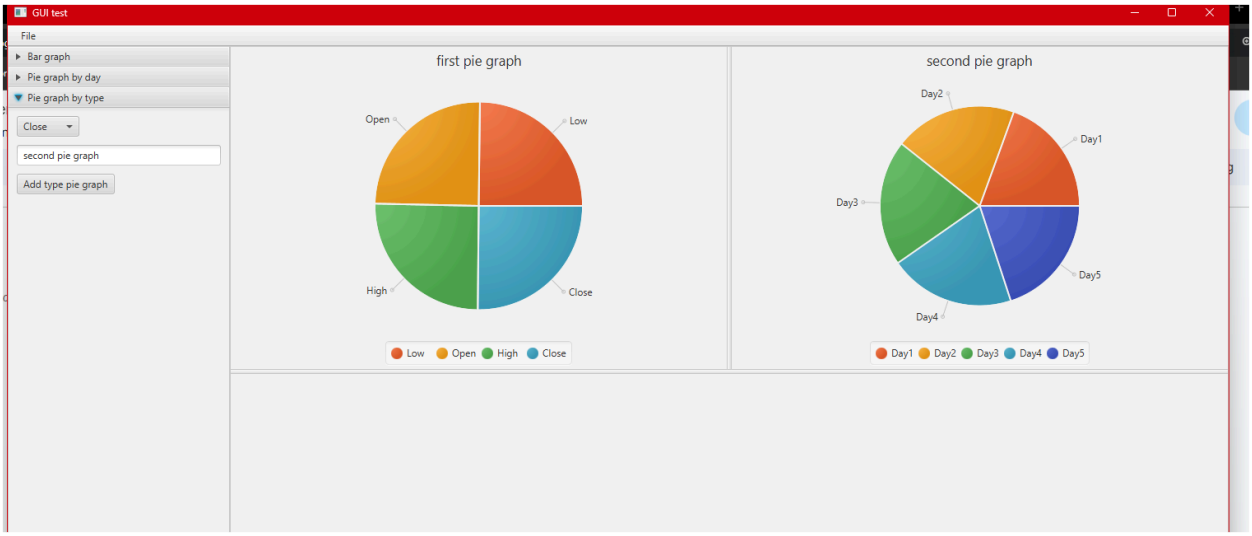


When you click the Add bar graph button, something like this will appear:

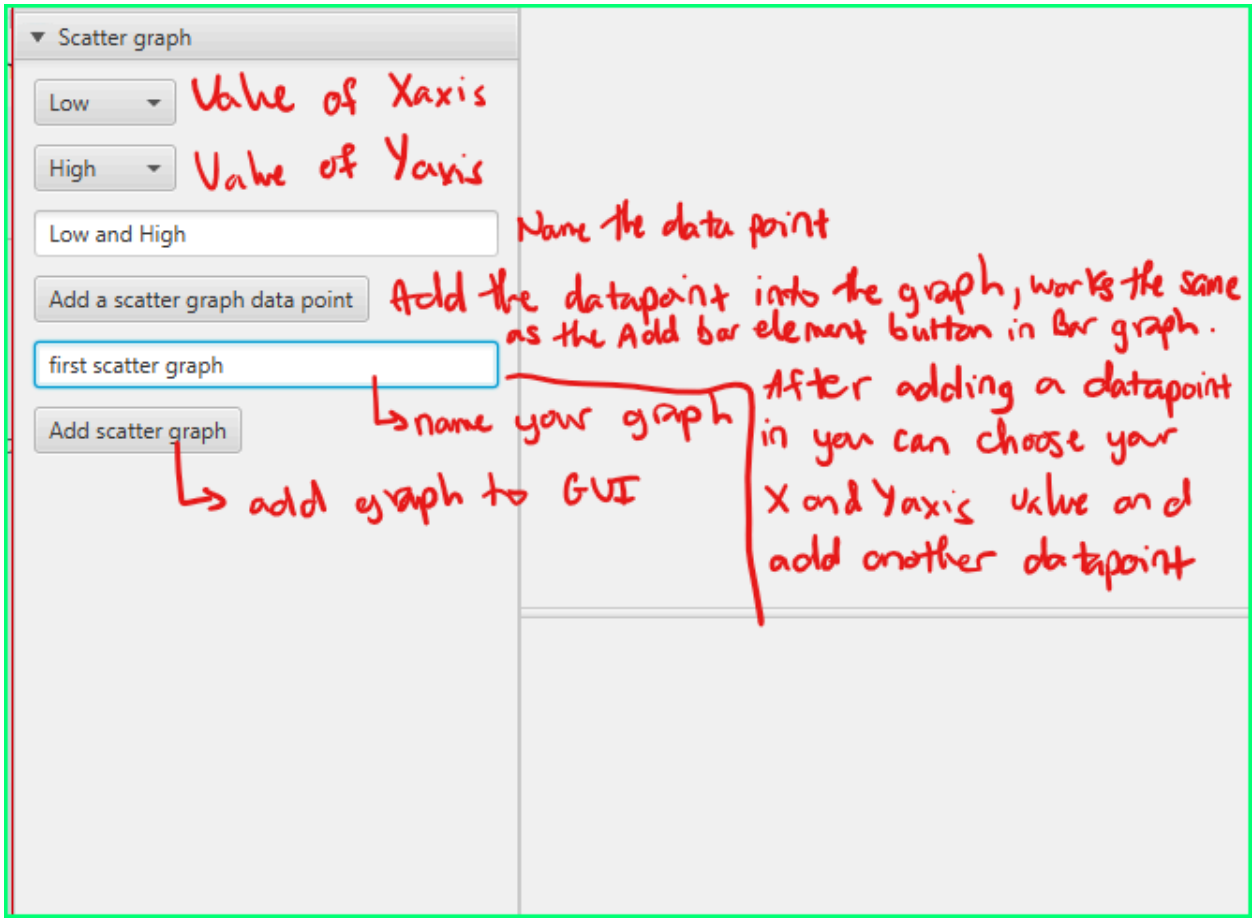
With pie graph by day:



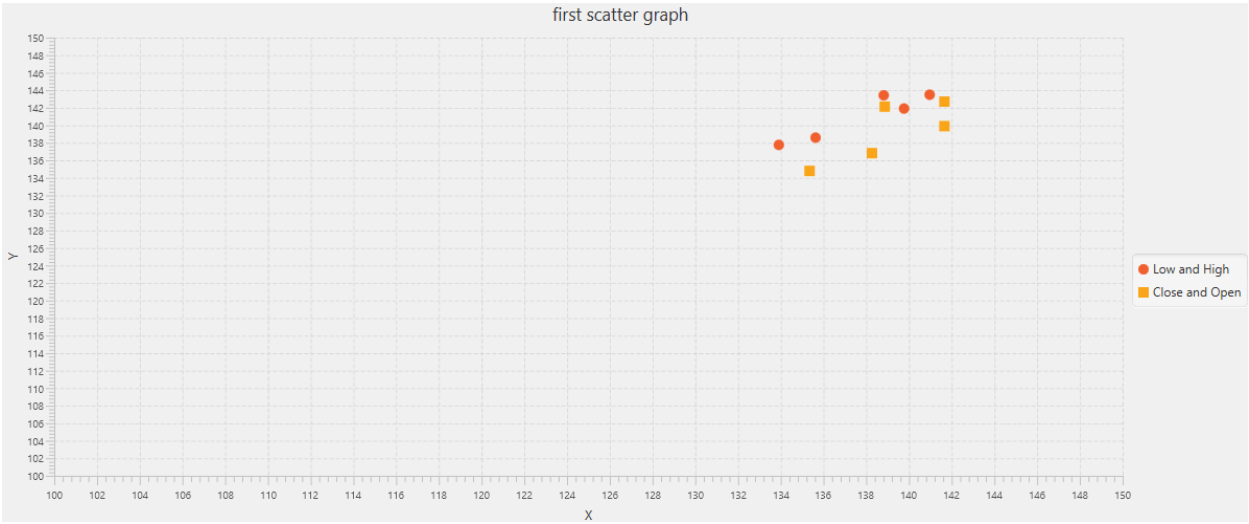
Pie graph by type works the same just the data goes by type of stock value instead of day of stock value:



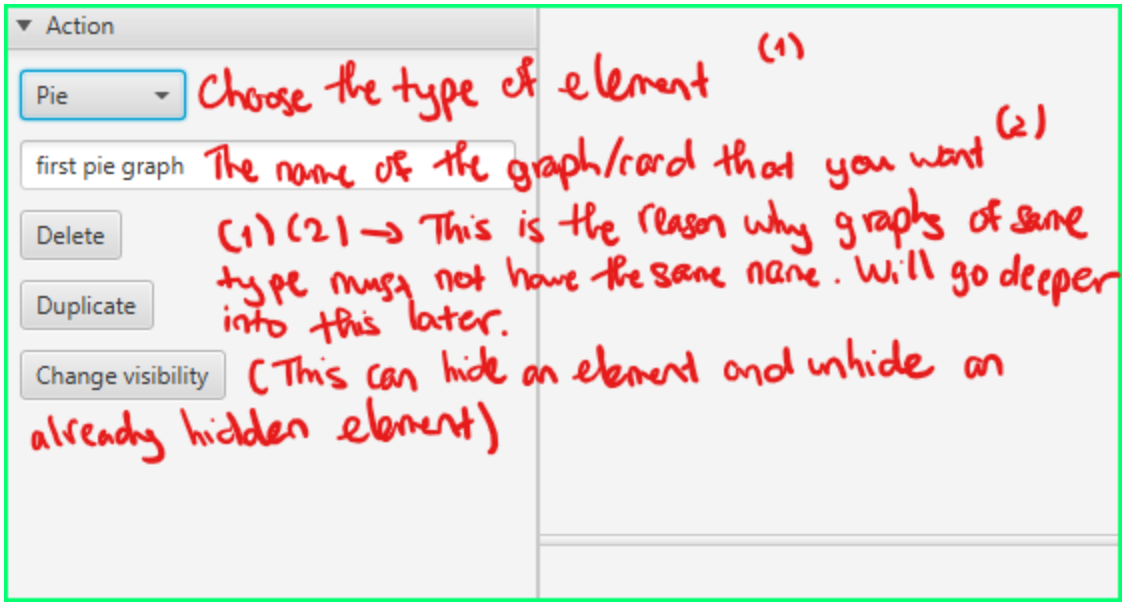
For scatter graph it's a bit more complicated, you have to combine two type of stock data as the value of your Xaxis and Yaxis to create a single serie in your scatter plot:



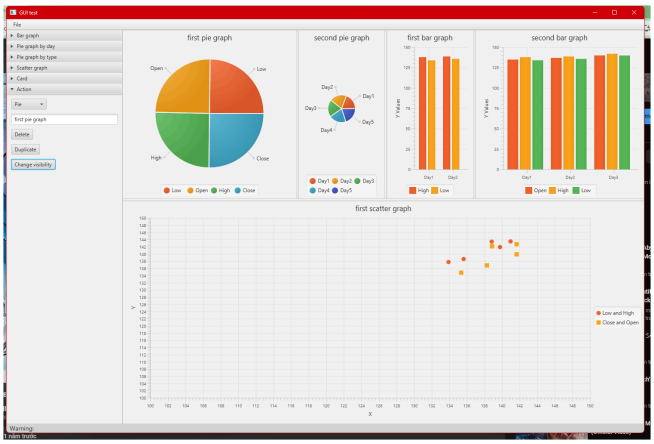
Your graph would look like this:



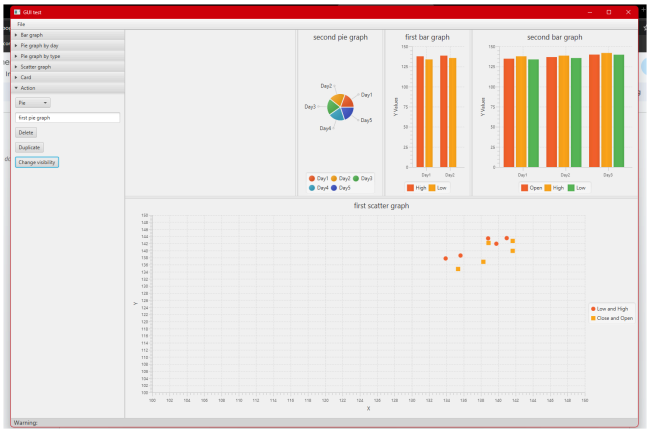
For the Action tiltedpane, contains all of the command that you can apply to the graphs/cards currently on your UI:



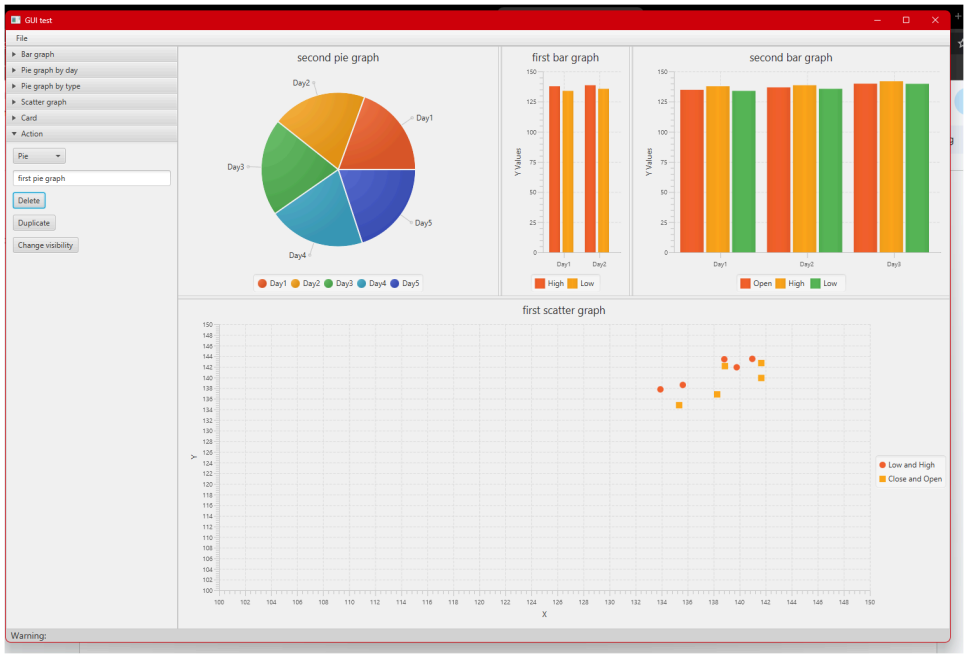
Before hiding the first pie graph:



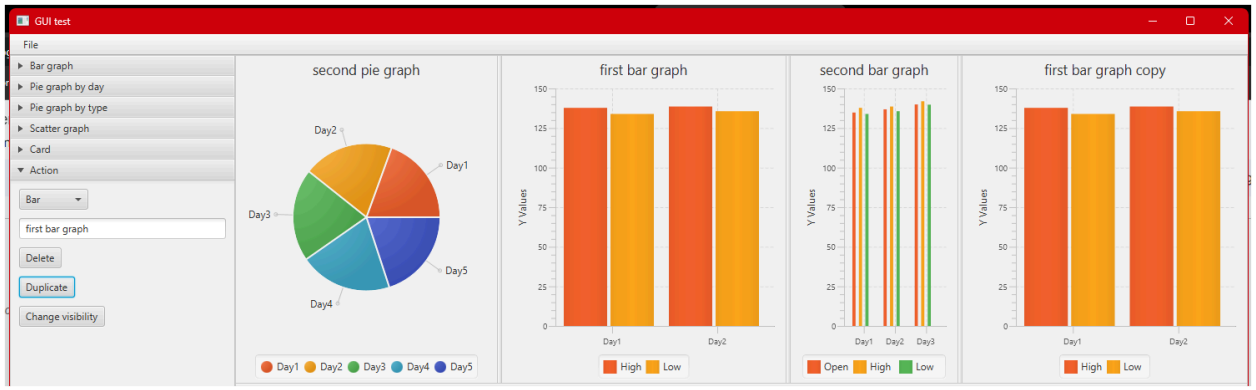
After hiding it:



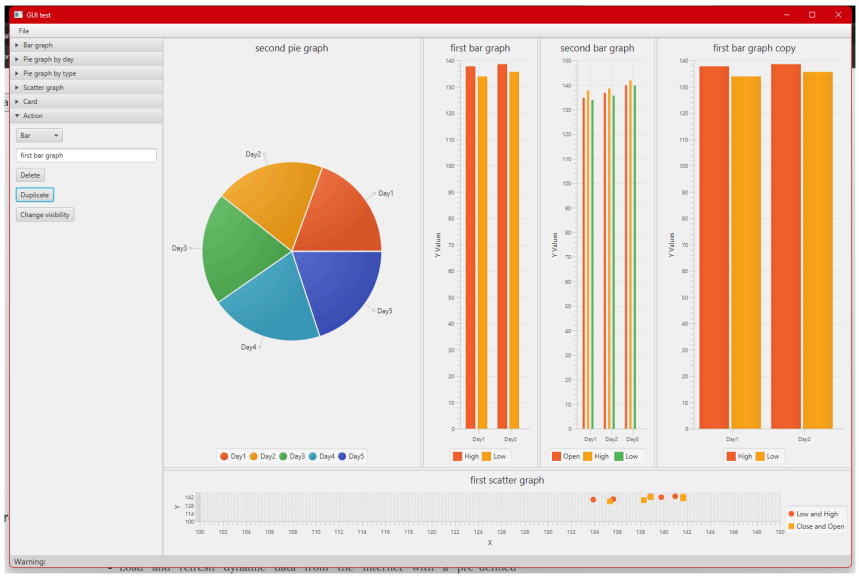
After deleting it:



Duplication:



User can adjust tile sizes:



Now let's take a look at file on the MenuBar at the top left of the screen:

Open data file: The user will upload a Json file containing all the stock value data onto the program. If the user tries to create graphs/cards without uploading anything, the program will just default to every value being zero. If the user uploads any faulty files, the program will do the same thing, a warning about this will appear on the bottom left of the screen. So without uploading any data file or faulty file your graphs would look like this:

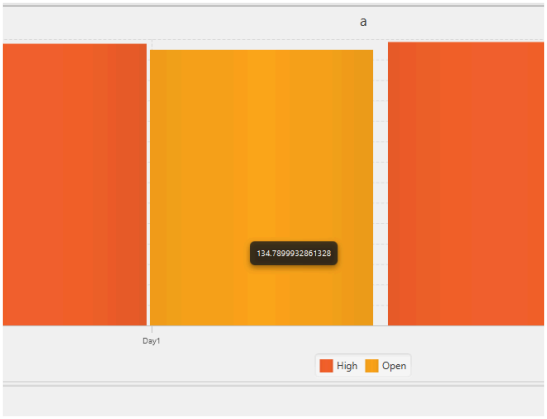
Open config file: The user clicks on this to upload the predefined file. After clicking on this, the program will delete all of the existing graphs/cards on the GUI and replace them with whatever's in the predefined files. When you upload faulty files, there will be a warning and the program just default to deleting every graph/card you have on your screen.
Load: The user will click on this to "save" the current state of the GUI
Save file: The user click on this to save the predefined Json file that contains the state of the GUI from the latest time of clicking "Load"

Let's take a look at the little Warning label on the bottom left of the GUI. When user do something they are not supposed to do, instead of carrying out the user command, the program warning label will display the messages according to the following situations:

- When the user give the same name to graphs of the same type
- When the GUI run out of space to add more graphs/cards
- When the user creates a graph/card without give it a name
- When the user applies action to a graph/card that does not exist on the GUI

- When the user uploads faulty file

Hovering your mouse over the data point will display its value:



4. Program structure

The UML is provided as a separate PDF file because it's hard to read if I just take a screenshot and paste it here.

The program consist of 5 objects:

- testExtractor: produces data for the generation of graphs and cards. The default data is zero for every value, stored in the variable "stock". Data files can be uploaded by calling the method readFile. Data is turned into the correct form and grabbed from the following methods: barChoose (for bar graph), pieDayChoice (for pie graph by day), pieTypeChoice (for pie graph by type and cards), pairData (for scatter graph) by calling the keys to get the values.
- Component: include the 3 classes of graphs which take a string as its parameter for the title and a Seq which varies based on the type of graph as their parameter for the data. They all share the Compo trait, which has the getType, getGraph and getTitle method. Trait Compo doesn't have method getData because each type of graph's data is of different data type.
- Cards: Has some helper methods to calculate the necessary values and the class Card which takes a string as its title, a string as its kind (min, max, sum, avg, sd), and Seq[Double] as its data. Because the class is of trait Compo it also has a getGraph method even though the name of the method doesn't really make sense. This method returns the card node, which is a VBox that contain the display value.
- Writer: This object contains the tools to deal with predefined files. The method bringDataToList() is called to update the current state of the dashboard into the writtenData string variable. By grabbing all the Compo objects from the 4 maps in the GUI and turn them into Json string. The readConfi method reads from a predefined file and turn it into a tuple4 with the same type as the 4 graph/card maps in the GUI. The method writeConfi write the writtenData string variable into a new file.
- dashboardGUI:

The readFile method of testExtractor is called whenever the user wants to upload a datafile. The "stock" variable in testExtractor is changed according to the data file. When the user choose their own data points for visualization, the values of the comboBox and Listview where they choose from matches with the keys in the key value pairs that belong to the maps that the

methods in testExtractor return. These data points are then stored in a variable which is gonna be the data input data for Compo objects. Then the method “increase” in object dashboardGUI is called, which adds the node into the GUI, making the graphs are cards appear.
In dashboardGUI there are 4 maps that keep track of all the graphs/cards that are currently on the GUI, the key being their title and the value are of class Bar/Scatter/Pie/Card (the reason for the existence of these 4 maps is explained below in section 6: Data structures). Whenever a map is added or deleted, the key value pairs of these 4 maps are altered accordingly.

When the user wants to “save” the current state of the program, they click on load which will update the writtenData variable into a Json string that reflects the state of the GUI. When the user wants to save the current state of the program into a file, method writeConfi is called. When the user wants to upload a predefined file the method readConfi is called from dashboardGUI. The return value of this method will then be the input value of method loadConfig in dashboardGUI, which does all the job of updating the GUI and the 4 maps accordingly.

The choice to use objects to divide and contain all of the methods, variables is because I find it convenient to just call the variables/classes/methods from other files in the same package using ObjectName.method/variable/className (like Component.Bar for example)

What I could’ve done better:
Looking back at my project, I can say that my code is very messy. Class Card is of trait Compo, while the code for trait Compo is in another file in the package. There are classes with the same name in different objects: object Component, Object Card and object Writer have classes that have the same name, which is confusing. I could’ve changed the name of those classes in Writer but for some reason errors started to appear so I just left them like that. I may have taken the saying “If it works, don’t touch it” a bit too seriously.

The code for delete/duplicate/change visibility button in my GUI looks like a wall of text because it’s just the same action repeated 4 times for 3 types of graphs and cards. I could’ve changed them into methods that I store in some separate object in a separate file in the same package and call them to keep things concise.

The interaction between dashboardGUI and Writer is awkward. Because I cannot access the maps in dashboardGUI from Writer, I have to update the maps in Writer from dashboardGUI, which means I have to create 4 more maps variables in Writer. These maps should’ve been stored in a different object file in the same package to make it less redundant and to keep the dashboardGUI filled with as few non-GUI element as possible.

5. Algorithms

The algorithm for calculating the display values for number cards is rather simple.
For the maximum, minimum and sum value, I just call the max, min and sum function on the input Seq of Doubles.

For average I just call sum and divide the return value by the size of the input Seq[Double]

For standard deviation I just followed how to calculate it from a KhanAcademy tutorial. I map the input data into the square of its distance to the mean. Then I call the sum function on that Seq divide by its size. Then I calculated the square root of the whole thing.

6. Data structures

a. In object testExtractor (dataFileReader.scala):

The data is extracted into object of type Quote (`case class Quote(volume: Seq[Double], low: Seq[Double], open: Seq[Double], high: Seq[Double], close: Seq[Double])`) low, open, high and close are the relevant data here, we ignore volume. The numerical data is stored into Seq of type Double.

Object testExtractor contains extraction methods that return Maps that have String as their keys and Seq of type Double as their value (except method pairData which has Seq[(Double, Double)] as its values).

The keys of type String match with the values the comboBox(es) and the ListView in the GUI. That is how the program is able to extract the requested data and turn them into graphs/cards.

b. In the GUI:

The mutable maps allBar, allPie, allScatter, allCard contain all of the graphs/cards that have been added into the GUI, 4 maps for 3 different types of graphs and cards. The keys are the title of the graphs/cards and the values are the graphs/cards themselves. When the user adds or deletes a graph/card, the key value pair of the respective graph will also experience the same thing. They serve the purpose of keeping track and easy access when the user wants to delete, duplicate, hide elements.

This is why graphs/cards of the same type cannot have the same name.

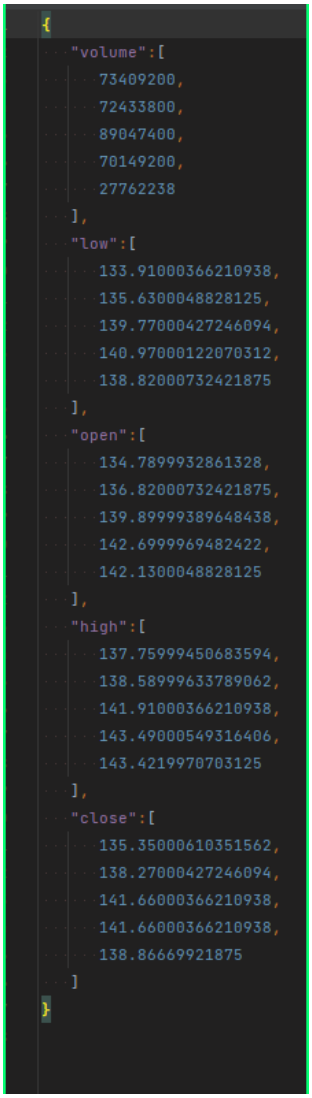
c. In object Writer:

The graphs are contained in Seq with the Scatter, Bar, Pie, Card version of object Writer. These classes contain the title and the data of their respective classes in object Component.

The choice to store Data in Seq is to take advantage of the rich set of operations provided by the Scala collections library, but the main reason is because of the need for immutability, implying that the data collection is treated as immutable unless explicitly stated otherwise. This can enhance code clarity and reduce the chance of unintended side effects. This is why even though storing the Data in a Buffer may not make much of a difference, it tends to be more suitable when mutability is a strict requirement. The data of the dashboard in my program doesn't need to be altered so using Seq will align with my purpose in this program.

7. Files and internet access

I decided to work with Json files because there are readily available tools and libraries that would help me parse the Json data: Circe and Upickle



This is how the dataFile would look like. Each of the stock value types is a Seq with 5 numerical values of type Double. Volume can be ignored but the rest have to have exactly 5 numerical values each.


```
{
  "bar": [
    {
      "theTitle": "a",
      "theData": [
        ["Low", [133.91000366210938, 135.6300048828125, 139.77000427246094,
140.97000122070312, 138.82000732421875]],
        ["Open", [134.7899932861328, 136.82000732421875, 139.89999389648438,
142.6999969482422, 142.1300048828125]]
      ]
    },
    {
```

```
    "theTitle": "a copy",
    "theData": [
     ["Low", [133.91000366210938, 135.6300048828125, 139.77000427246094,
140.97000122070312, 138.82000732421875]],
     ["Open", [134.7899932861328, 136.82000732421875, 139.89999389648438,
142.6999969482422, 142.1300048828125]]
    ]
   }
  ],
  "pie": [
   {
    "theTitle": "b",
    "theData": [135.6300048828125, 136.82000732421875, 138.58999633789062,
138.27000427246094]
   },
   {
    "theTitle": "c",
    "theData": [133.91000366210938, 135.6300048828125, 139.77000427246094,
140.97000122070312, 138.82000732421875]
   },
   {
    "theTitle": "bcopy",
    "theData": [135.6300048828125, 136.82000732421875, 138.58999633789062,
138.27000427246094]
   }
  ],
  "scatter": [
   {
    "theTitle": "d",
    "theData": [
     "a",
     [
      [133.91000366210938, 137.75999450683594],
      [135.6300048828125, 138.58999633789062],
      [139.77000427246094, 141.91000366210938],
      [140.97000122070312, 143.49000549316406],
      [138.82000732421875, 143.4219970703125]
     ]
    ]
   }
  ],
  "card": [
   {
    "theTitle": "e",
    "kind": "Min",
```

"theData": [135.35000610351562, 138.27000427246094, 141.66000366210938, 141.66000366210938, 138.86669921875]
}
]
}

This is what a predefined file would look like. After the user clicks save file on their dashboard, the data would be loaded and turned into a Confi object (check class Confi in object Writer) then that object gets turned into a Json string stored in a variable in object Writer.

Object Writer include its own version of class Bar, Pie, Scatter and Card (I should've given them different names from the ones in object Component, I tried changing their names but the file reading no longer works for some reason so I just leave them like that)

Object Writer contains class Confi, which is what this Json file is, a single Confi object. Class Confi contains the following parameters:

- bar: a Seq of all Bar objects
- pie: a Seq of all Pie objects
- scatter: a seq of all Scatter objects
- card: a Seq of all card objects

(the Writer's version of Bar, Pie, Scatter and Card. Not the object Component's version)

Class Bar, Pie, Scatter and Card all have the parameters "theTitle" and "theData", the methods getTitle and getData method, Card in addition also has the parameter "kind" and method getKind.

The input data type of object Writer's version of class Bar, Pie, Scatter and Card matches with object Component's version. That is why I can invoke all the "get" methods object Writer's version of those classes and get their return value as input for object Component's version of those classes.

8. Testing

Object Cards calculation methods: I used unit testing. I create my own List of Doubles, calculate the max, min, sum, avg, sd (standard deviation) using an online calculator. Then I compare the return value of the calculation methods to the values that the online calculator produces. This is the same as my what I have in my technical plan

Object testExtractor: I extend it with App (`object testExtractor extends App:`). I wrote println commands that print out the return values of all of the methods and check if the methods work or not. Then I check if the return values have the correct format and if the numerical values match with the ones in the input json file. This is also the same as what I have in my technical plan.

Object Component: I test the classes in a testGUI. I created a Compo.Bar/Pie/Scatter/Card object and called their getGraph methods. I then use the return value as the root of this testGUI

then run it. I can determine if the classes and their methods work properly or not according to what is displayed on the testGUI. This is not the same as what I have in my technical plan, because I was planning to test these classes using unit testing back then.

Object Writer: Same as testExtractor, I just invoke the println command for the methods and variables I have.

9. Known bugs and missing features

I've managed to implement all of the features in the moderate requirements.
All the action that the user performs that could potentially create bugs like:

- The allBar, allPie, allScatter, allCard maps containing elements that are not displayed in the GUI.
- When duplicating, there's no new key value pair in the aforementioned maps. Which means that the new graph/card created from duplication isn't registered into those maps.
- When deleting the a graph/card from the GUI, its key value pair isn't deleted from those maps
- Typing in the name of an element that doesn't exist in those maps
- When the user try to read faulty files

I've dealt with them all.
There is this thing in dashboardGUI, which displays 3 red errors with the message "Wrong expression" that points at the bunch of if else clauses in my code for duplication. The program still runs normally despite the presence of those errors.

10. 3 best sides and 3 weaknesses

3 weakness:

1. Having to keep track of all of my graphs/cards in 4 separate maps, which means the user would have to specify the type of graph/card and its name before being able to interact with it.

   The reason I had to use 4 separate maps for each type instead of putting them all into one single map is because all of the graphs have different input data (Seq[(String, Seq[(Double, Double)])] for scatter graphs, Seq[Double] for pie graph and Seq[(String, Seq[Double])] for bar graph). That's why even if they all share the same Compo trait I cannot implement a getData method for trait Compo. So when implementing duplication, where I need to call the getData method, I have to access the graph/card from their respective map to ensure the correct data type to input the duplication graph/card.

2. The way I implement my duplication/delete/changeVisibility button: The code looks unappealing to read because it looks like there's so much going on but I'm basically just doing the same thing 4 times for 3 different types of graph and card

3. I could have optimized the process of turning the data from the content of the 4 maps into a json file and vice versa. Since I cannot access the maps from the GUI from object

Writer, I have to write the exact same 4 maps in object Writer, and transport the data from the GUI to those maps using a method I wrote in the GUI file, which does not look very nice aesthetic wise.

3 best sides:

1. My method of choosing data points for bar graph and scatter graph: Allow the user more freedom in choosing what data they want to display and how many data points they want to have in their graph
2. My graph class: I think this is the most optimized part of my project. At first I was only creating graphs by invoking a method that would return a graph, which is a deviation from the original technical plan. But when I got down to implementing delete/hide/duplication I decided to switch to making it a class. Just by providing a string for the title and inputting the data, you get yourself a graph. The getTitle and getData methods work well for when I implement duplication (which I personally think is one of the more complicated requirements for this project), where I can just extract the title of the target graph/card and add a "copy" to it, then extract the data using getData and create a new graph/card.
3. My method of saving and opening predefined files. Despite the process being considered a weak point by myself, I consider the functionality to be a strong point since I have made it extremely simple for the user to work with predefined files: just loading and saving, no need for any more specification on the user's side on how they want to save their graph.

11. Deviations from the plan, realized process and schedule

Action	Week
1. Researching	1-2
2. Implementing File reading and testGUI for graph method	2-3
3. Completing the implementation of graph method and cards	3-5
4. Implementing the main GUI and class graph/card	5-7
5. Implementing predefined file saving and reading	8
6. Debugging, cleaning code and writing final documentation	9

The project deviated from the technical plan right at the beginning, where instead of actually implementing anything, I spent the first 2 weeks looking up how to build my project and being rather unproductive in my project.

The way I proceed with my project doesn't match with my technical plan either. Everything started pretty slowly because I spent most of my time experimenting to see what works and what doesn't. I managed to get my file reader method to function properly in the second week. In the third week I was implementing a test GUI and experimenting with random input data for my

graph methods to make sure that they display something and that something is the correct input data.

The 5th week is when I started to implement the main GUI, this is probably the biggest difference from the technical plan. During the process I kind of understood the difference between frontend and backend development and I personally think this project is more frontend orientated. The GUI is where everything in my project comes together and becomes connected. I spent the most of my time fiddling around to learn how to make a GUI. I figured out how to extract the data using the values of comboBox and Listview that match the keys in the key value pairs of the maps that the methods in object testExtractor return. One deviation of the plan is the implementation of the graph method instead of class, but when reaching the development of duplication/delete/change visibility, I realized I need a way to access the graphs that I have created. That's when I switched back to making the graphs and cards a class instead of methods.

Testing, even though in the technical plan was allocated 2 weeks, is not present in the table that I provided above. This is because testing isn't done all at once like I have anticipated, instead it comes with every implementation of different classes, methods and the GUI. So you can say that testing is a part of the implementation itself.

The predefined file saving and uploading isn't even considered in the original technical plan. It is also the final thing that I implemented in my program.

The order of implementation looks something like this:
fileReading -> graph/card methods -> Main GUI -> class graph/card -> Main GUI with duplication/delete/change visibility -> predefined file saving/reading -> debugging

I can confidently say that the time estimate in my plan barely matches reality. It took me twice as much time for file reading and developing class graphs/card. Class Dashboard and the GUI in the original technical plan has been merged into just GUI and also takes almost twice as long. No time allocated for testing because everything is tested along the way. No time allocated to additional features because there's no time to do that. 1 week is allocated for predefined files. The only thing that matches is the writing of this documentation and submission.

To answer the question of what I learned during the process in general terms is a difficult task without making it lengthy. Overall this project is where I get to put everything I learned in O1 and the first few weeks of studio A into practice. Before the project I only understood how things work, but don't really know how to put them to use. I didn't really understand file reading in the course material. But now I know how to work with Json files. I know how to create graphs, I know how to write a unit test. And most importantly, I know how to navigate myself through the process of creating a GUI. Now I realized there's so much creative potential in what you can do with programming and developing a program.

12. Final evaluation

Overall, considering my past experiences (which is none) and the amount of time I spent on the project, I am still proud of myself.

Functionality wise, the program works perfectly fine and fulfills all of the requirements. There is a sense of creativity in the implementation of data point choosing, allowing the user with more variety in creating graphs. Predefined file saving and opening is extremely simple for the user. The GUI is easy to the eyes.

Although the implementation has much to be desired: Code is messy, a description of this has been provided in detail in section 4, which might make it confusing to understand the intention of the programmer by just looking at the code. I think the main reason for this problem is that since this is my first time writing a program, once I've finished a part and it works, I would just leave it there and keep adding on to the program. There is rarely a moment where I would look at my project as a whole and think of how I could rewrite the whole thing to make it more optimized, more concise.

Now that I am at the end of my project, in hindsight I could've make my code much clearer and more optimized for myself:

- I would try to minimize the non-GUI element in dashboardGUI as much as possible. I would write a separate file object that would contain all the methods that carry out delete/duplication/change visibility. The same thing would happen to all the variables that are currently residing in dashboardGUI.
- I would either put the trait Compo in a separate file or just put Cards into object Component.
- I would find of a way that would let me store all of my graphs/cards into one single map instead of 4 separate maps
- Even though I don't know how to, if possible I would like to implement my program in a way that instead of the user having to specify the name of the graph and its type to be able to interact with it, the user can just click on it. The chosen graph/card will be stored in some variable and can then be interacted with. This way removes the need to have different names for graphs and makes it more convenient for the user.

13. References and code written by someone else

Code used to create graphs:
https://github.com/scalafx/scalafx/tree/master/scalafx-demos/src/main/scala/scalafx/scene/chart
Explanation: I have no idea how to create a graph with scalafx without using this. When copied to my project, I did combine parts of the code here into my graph classes

Code used for tooltip (as inspiration):
https://stackoverflow.com/questions/14117867/tooltips-for-datapoints-in-a-scatter-chart-in-javafx-2-2
This just helps me know the way of implementing tooltip, then I wrote that code in scala for my tooltip.

14. Appendices

https://version.aalto.fi/gitlab/hoangk2/khanh-psa2-project

This is the github which contains everything of my project.