1. Collection Choices

Market Class: Collection Object: ArrayList<Company>

Reasons:

Dynamic Resizing: ArrayList can dynamically resize, adjusting based on the number of companies present in the market, thereby saving space.

Fast Access: Provides fast access to elements, which is beneficial for operations involving market-wide data processing.

Company Class: Collection Object: ArrayList<Double>

Reasons:

Dynamic Resizing: ArrayList can dynamically resize, which is suitable for storing the company's historical stock prices that grow over time.

Fast Index Access: It provides fast access to elements by index, which is essential for analyzing stock price trends efficiently.

Investor Class: Collection Object: ArrayList<Company>

Reason:

Dynamic Resizing: ArrayList can dynamically resize, accommodating the changing number of companies an investor is interested in.

Sequential Access: Provides fast sequential access, which is beneficial for iterating through the list of companies to update investments or make decisions.

1. OOP Features
   1. Single Responsibility Principle

The functional design adheres to object-oriented principles. In this project, inheritance is deemed inappropriate.

The Single Responsibility Principle (SRP) states that every class in the program should have responsibility over a single functionality of the program; a class should do one thing. This principle is incorporated into our class design: all the classes have their own, separate purpose.

For instance, the Company class is focused on managing all details and operations related to a company. It includes attributes such as the company name, current stock price, and price history. Additionally, the Company class provides methods to update the stock price, get the current price trend, and calculate the percentage change in stock price. By encapsulating all functionality directly related to the company within a single class, we ensure that the class is highly focused on its core responsibilities.

The Market class is responsible for managing a collection of companies. It provides methods to add, remove, and retrieve companies. By limiting the Market class's responsibilities to managing company objects, we make it easier to maintain and extend.

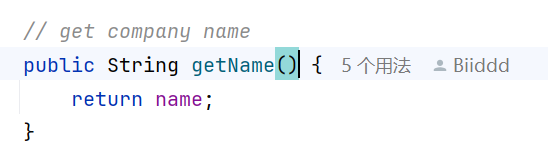
The Investor class focuses on handling the investor's portfolio and investment decisions. It manages a list of companies the investor is interested in and the corresponding investment strategy for each company. This way, the Investor class can execute buy and sell operations based on different strategies.

By adhering to the SRP, we ensure that each class is focused and easier to maintain or extend.

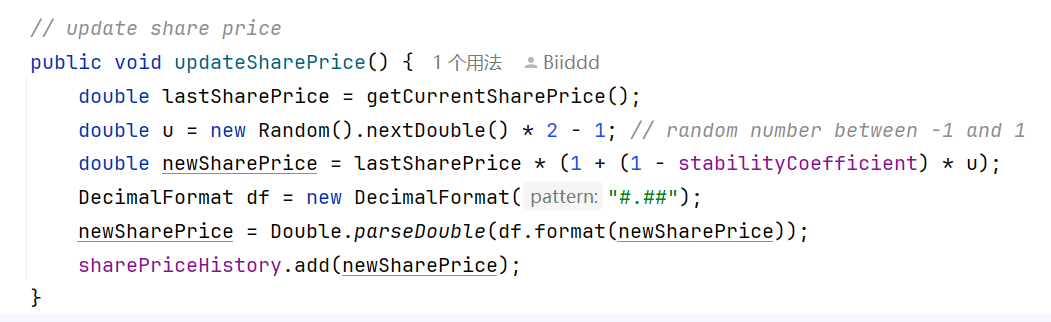
* 1. Encapsulation

Encapsulation is another fundamental OOP principle applied in our design. All the class attributes are private, ensuring that they cannot be accessed or modified directly from outside the class. This promotes data hiding and protects the internal state of objects. To interact with these attributes, we provide public getter and setter methods. This approach ensures that any changes to the internal implementation do not affect other parts of the program, enhancing maintainability and robustness.

For instance, consider the Company class. It manages various details related to a company, such as its name and historical stock prices. These attributes are private to prevent direct manipulation from outside the class. Instead, we use getter and setter methods to interact with these attributes safely.



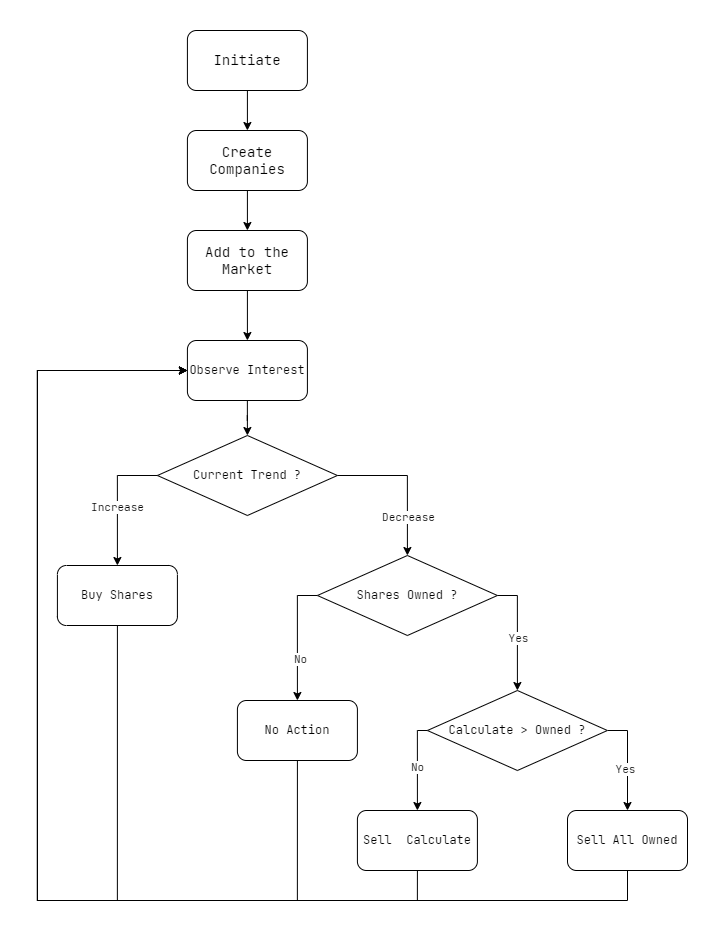




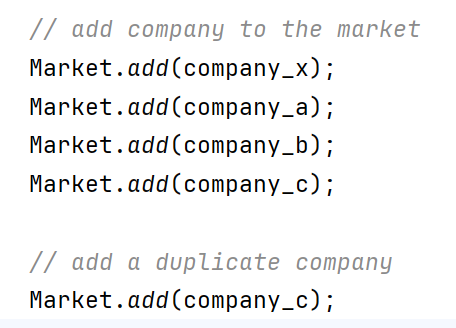
In this example, the name and sharePriceHistory attributes are private. The `getName`, `getCurrentSharePrice`, `getSharePriceHistory`, `getCurrentTrend` methods provide controlled access to these attributes. The `updateSharePrice` methods allow modification of the stock prices list, but within the constraints defined by the class.

1. Testing Description

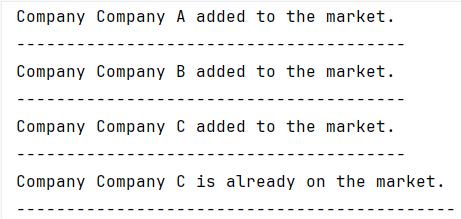
All tests are executed in the Controller class. The code covers all methods, and in the report, I will describe them by class. The testing logic is as follows: create companies and add them to the market, then create an investor. The starting capital of the investor can be modified in the wallet parameter. The investor manually adds companies to their list of interests and then observes the stock prices of all the companies they are interested in. If the stock price rises, a buy operation is performed, where the number of shares to buy is calculated as sharesToBuy = maxTransaction / currentPrice, rounded down. If the stock price falls, shares will be sold, and the number of shares to sell is calculated as sharesToSell = maxTransaction / currentPrice. Note that if the current number of shares held is less than the calculated sharesToSell, sharesToSell needs to be reassigned to the current number of shares held. This simulation will loop for update times, and the update parameter can be modified at line 58 in the Controller class.

Here is my flowchart:

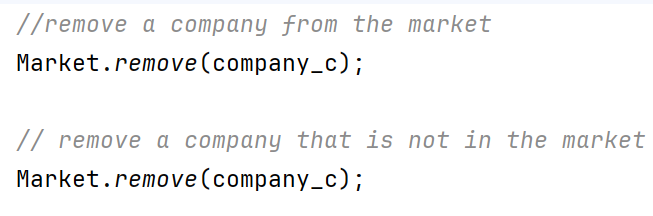
* 1. Market class

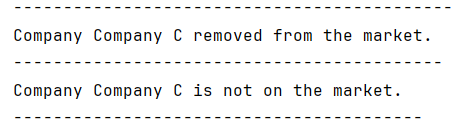
In the Market class, there are three methods involved: add, remove, and getCompanies. First, create four companies, then add them to the market, with Company C being added twice.

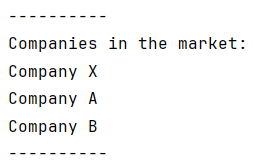
Add Company C twice

Observe the program output:

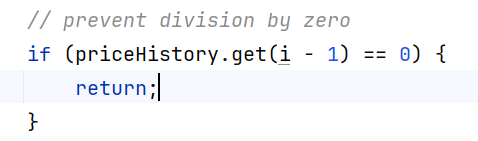
Next, delete Company C. When deleting, check if it exists; if it does, delete it directly, otherwise, throw an exception. Test the exception handling by attempting to delete a company that is not in the market.

remove Company C twice

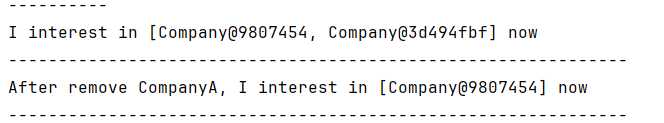
The program output is as follows:

After these operations, we use the `Market.getCompanies` to print out the companies which are still in the market:

* 1. Company class

In the Company class, there are numerous methods for accessing and updating attributes, which will not be elaborated upon here. Among them, the updateTrend method holds particular significance. We employ a currentTrend variable to store the current stock price trend. When calculating |s2 - s1| / s1, it's essential to note that while the new stock price is typically greater than zero according to the stock price calculation formula, it can approach zero. Since stock prices are rounded to two decimal places, the possibility of reaching zero exists. Indeed, such occurrences have been observed in my testing. Therefore, it's imperative to handle exceptions gracefully. Below is the exception handling code:

* 1. Investor class

Investor class is relatively simple. In the Controller class, we add CompanyX and CompanyA to test the addInterest method. After adding them, we call investor.getCompaniesOfInterest method to output the current list of interested companies. Then, we use the removeInterest method to remove CompanyA from the list of interested companies. We output the current list of interested companies again to check if the removal was successful. The code output is as follows:

* 1. BasicStrategy class

The invest method in the BasicStrategy class contains the logic for buying and selling stocks, including when to execute these actions as mentioned earlier. The sharesToBuy method includes the logic for buying stocks, which involves calculating the quantity to buy. Similarly, the sharesToSell method includes the logic for selling stocks and calculates the quantity to sell, while also handling situations where the number of stocks held is less than the calculated quantity.

1. Model Discussion