Design rationale

This basically filters and checks if stock price is less than 500 and then obtains the maximum price of these resulting stocks.

We used Java's functional features like maps, filters, and lambdas. Our pickShareFunctional method is single lined with no local variables.

In our pickShareFunctional class, findHighPrices function returns

```
stream.map(this::safeFoo).filter(ShareUtil.isPriceLessThan(500)::test).max(Comparator.comparing(x -> x.price)).get();
```

We have 10 static stocks in a list

```
public static final List<String> symbols= Arrays.asList("IBM","AAPL","AMZN","CSCO",
"SNE","GOOG","MSFT","ORCL","FB","VRSN");
```

Due to API limits to the number of calls per minute, thus, program is unable to make enough API calls in under a minute to get the 10 stocks' information. This is the main limitation but overall, it does produce required results correctly with the assumption that stock prices have not changed during runtime. For 2 stocks, both having stock with highest price, the stock chosen is determined by the `.max()` in the findHighPrices function and the result could be arbitrarily chosen.

Testing

Manual testing is done to ensure program validity.

Test ID	Price	Expected Output	Actual Output	Status
1	500	AAPL & price	AAPL & price	PASS
2	1000	AAPL & price	AAPL & price	PASS
3	4000	AMZN & price	AMZN & price	PASS

Known error, faults and defects

The fault of the program the quota limit on the API call with 5 calls per minute limit. With 10 stocks there for the program to check, it will take at least 1 minute to run regardless of design criteria, and this results in a delay. Also due to the quota limit on the API (5 calls per minute) a consecutive run within one minute may return a value of 0 for the price.

What about functional style exception handling? Can I learn anything from Rust here?

In Rust, panics are equivalent to exceptions and there is a cost to traditional exception model and thus rust comes in for systems that cannot have performance hit from exception handling. Result/Option gives a straightforward route as on failure we are forced to check exhaustively matching cases over an Option/Result or explicitly making it into a panic.

RUST

```
let f = match f {
Ok(file) => file,
Err(error) => panic!("Problem opening the file: {:?}", error),
};
```

JAVA

```
Optional<Foo> possibleFoo = doSomething();
possibleFoo.ifPresentOrElse(
foo -> { /* ...do some work with foo... */ },
() -> { /* ... do some work when no foo found... */ }
);
```

Java gives us java.util.Optional to represent nonexisting value which we can handle in a functional manner. In this case, ifPresentOrElse allows us to handle case where Optional is empty.

Out of the three steps explained above in part (a), which step do you think is responsible for most of this time?

The step responsible for the longest time out of the 3 was step 1 '1- Create a list of ShareInfo filled with the price for each of the symbols in Shares'.

Calculate the execution time as in part (b) again, but now with parallelStream() instead of stream(). Explain why the execution times are different?

Using stream, / /TIME: on AVG 4200 milliseconds (Based on 10 trials).

Using parallel stream, TIME: on AVG 850 milliseconds (Based on 10 trials).

Parallel stream is faster due to concurrent execution of tasks. This is because parallel stream incorporates parallel processing on multiple cpu cores. Hence the process of task is distributed, this will enhance scheduling and essentially the program threads will get more cpu time than a single core execution.