# L16 Homework: Market Inventory Manager

## Problem Description

Hello! Please make sure to read all parts of this document carefully.

In this assignment, you will be applying your knowledge of file I/O and exceptions to build the system an employee can use to manage the existing inventory in a market that buys from suppliers and sells to customers and read/write the inventory state from/to a file.

## Solution Description

You will write 4 classes, each in their own file (named accordingly).

- Visibility for required elements:
  - All classes, constructors, and methods should be public **unless otherwise stated**.
  - All variables should be private.
- You may add to aid you (if you want):
  - Methods, if they are private.
  - Variables, if they are private static final, and their type is a primitive type, a wrapper class, or String.
- You may not add any other classes or constructors.

### CannotFulfillTransactionException

A checked exception describing a reason why the market is unable to fulfill a transaction. It should directly extend the top class for checked exceptions and have a 1-arg String constructor for the message.

### MalformedInventoryFileException

A checked exception describing a reason why the file containing the information of the market is not valid. It should directly extend the top class for checked exceptions and have a 1-arg String constructor for the message.

### Product

It represents products for the market inventory, as well as some information about them. The class will have:

- The following instance variables: `id` (`String`), `buyPrice` (`int`), `sellPrice` (`int`), `quantity` (`int`).
- 4-arg constructor taking the above in that order and setting them.
- Getters for all instance variables, setter for quantity.
- `toString()` method that returning a String with all instance variables comma-separated ("id,buyPrice,sellPrice,quantity").

## MarketInventoryManager

It is used to manage the market inventory. It can load the inventory and the money the market has from a file, do buy/sell transactions, and store the information back to a file. It is also a java program. It will have:

- The following instance variables: `products` (`Product[]`), `money` (`int`).
- private 2-arg constructor taking the above in that order and setting them.
- `static MarketInventoryManager fromFile(File)` method that creates and returns a `MarketInventoryManager` instance if the file has the valid information to create one. The method may propagate `IOException` and `MalformedInventoryFileException` in some cases.
    - If the file cannot be read, the method should propagate whatever `IOException` happens.
    - A valid file always has a 1-line header containing the number of products and the current money the market has, in that order and separated by a space. Both need to be non-negative integers. If that isn't the case or the header has otherwise missing or extra information, the method should end abruptly by propagating a MalformedInventoryFileException with the message "Invalid inventory header"
    - If the header is valid, the method should continue looking at the file for the product information. If there aren't enough lines to describe all the products (each in one line), the method should end abruptly by propagating a MalformedInventoryFileException with the message "Product information incomplete"
    - Next, it will do the following for each line that describes a product, in order:
        - If the line isn't composed of exactly four comma-separated values, the method should end abruptly by propagating a MalformedInventoryFileException with the message "Product information invalid for product [product number]" (without square brackets). Note that this is not the product id, the first product in the file is product 1, the second is product 2, etc.
        - Otherwise, if the first value (the product id) is empty or contains spaces, then it should also finish with the same runtime exception as the above item.
        - Otherwise, if the product id is the same as the one in another product previously loaded, it should end abruptly by propagating a MalformedInventoryFileException with the message "Duplicate product ID: [id]" (without square brackets)
        - Otherwise, if any of the other values (in the order `buyPrice`, `sellPrice`, `quantity`) is not a non-negative integer, the method should end abruptly by propagating a MalformedInventoryFileException with the message "Product information invalid for product [product number]" (same as first item)
            - Note that any of them can be 0, and the `buyPrice` could be larger than the `sellPrice`.
        - Otherwise, the line is valid (it aligns with the Product toString format), and the method should create a new `Product` and store it in the array that will be used to construct the `MarketInventoryManager` if the file is fully valid.
    - If all products are valid, the method returns an instance of `MarketInventoryManager` with the appropriate products and money.
- `void buy(String, int)` method that is used when buying products from suppliers. It takes a product id and a quantity. The method may propagate CannotFulfillTransactionException in some cases.

    - If quantity is negative, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Quantity is negative"
    - Otherwise, if the provided product id cannot be matched to any of the products in the market, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Product ID not found"
    - Otherwise, if the market doesn't have enough money to buy the product for the requested quantity, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Not enough money to buy"
    - Otherwise, the product is bought (adjusting the remaining quantity), and the money is decreased according to the product's buy price.
- `void sell(String, int)` method that is used when selling products to customers. It takes a product id and a quantity. The method may propagate CannotFulfillTransactionException in some cases.
    - If quantity is negative, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Quantity is negative"
    - Otherwise, if the provided product id cannot be matched to any of the products in the market, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Product ID not found"
    - Otherwise, if the market doesn't have enough quantity of that product to sell, the method should end abruptly by propagating a CannotFulfillTransactionException with the message "Not enough quantity to sell"
    - Otherwise, the product is sold (adjusting the remaining quantity), and the money is increased according to the product's sell price.
- `String[] marketProducts()` method that returns an array with all the product IDs, in the same order as the products are.
- `void saveToFile(File)` method that saves the state of the market inventory to a file, according to the format in the `fromFile` method. The method should propagate an `IOException` if any occurs.
- main method
    - The program expects a filename through console arguments. If the number of arguments isn't exactly 1, the program should print on its own line "Usage: java MarketInventoryManager <inventory-file>" and finish.
    - Otherwise, the program should attempt to create a MarketInventoryManager from the filename. If that fails with any exception, the program should print on its own line "Error: [exception message]" (without the square brackets) and finish the program
    - Otherwise, the program should prompt the employee to enter commands to operate on the inventory, in a loop. For each command, it will print "> " and wait for a command in the same line.
        - If the command isn't "quit", "products", "buy [product id] [quantity]", or "sell [product id] [quantity]" (buy/sell without square brackets, and assume product id doesn't have spaces), it should print on its own line "Error: Invalid command" and re-prompt the employee for another command. Note that for buy/sell it should also do if the quantity isn't an integer.
        - Otherwise, if the command is quit, it should finish the command loop.
        - Otherwise, if the command is products, it should print on its own line the product IDs of all products, separated by a comma and space if there are multiple.

        - Otherwise, if it’s the buy command, it should do the buy operation. If that fails with any exception, the program should print on its own line "Error: [exception message]" (without the square brackets) and re-prompt the employee for another command.
        - Otherwise, it should do the sell operation. If that fails with any exception, the program should print on its own line "Error: [exception message]" (without the square brackets) and re-prompt the employee for another command.
    - After the command loop finishes, the program should save the updated state of the inventory in the same file as it was read from. If that fails with any exception, the program should print on its own line "Error: [exception message]" (without the square brackets). The program then ends regardless of the outcome.

## Testing Your Solution

You can test Java programs (classes that have a main method) by interacting with the console and verifying that the interaction with the user is correct. For parts of the HW that are not covered by a Java program, you can write your own driver classes to test those components (do not submit any additional class you write).

## Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import **java.io.File**, **java.io.IOException**, **java.io.PrintWriter**, **java.util.InputMismatchException**, **java.util.NoSuchElementException**, and **java.util.Scanner**. You are not allowed to import any other classes or packages.

## Feature Restriction

There are a few features and methods in Java that overly simplify the concepts we are trying to teach. For that reason, do not use any of the following in your final submission:

- **var** (the reserved keyword)

## Checkstyle

The Checkstyle deduction limit for this assignment is 30 points, counting Javadoc errors. Refer to the course guide for the use of Checkstyle and review the autograder Checkstyle report on your submissions.

## Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, we require that you place a collaboration statement as a comment at the top of at least one .java file that you submit. That collaboration statement should say either:

*I worked on the homework assignment alone, using only course materials.*

or

*In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].*

If the autograder encounters issues processing your collaboration statement, begin it with "Collaboration Statement: ".

<u>Allowed Collaboration</u>

When completing homeworks for CS 1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- approved:
  - "Hey, I'm really confused on how we are supposed to implement this part of the homework.
  - What strategies/resources did you use to solve it?"
- disapproved:
  - "Hey, it's 10:40 on Thursday... Can I see your code? I won't copy it directly I promise"

In addition to the above rules, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

## Turn-In Procedure

Upload the files listed below to the corresponding assignment on Gradescope:

- CannotFulfillTransactionException.java
- MalformedInventoryFileException.java
- MarketInventoryManager.java
- Product.java

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

Important Notes

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit .class files or .jar files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications