

Introduction to Programming

Level 4

Programming Portfolio Report

[S244679]

[4382]

# Background

process of the key decisions made about packages, classes, attributes, methods, and parameters

To begin programming the application, the first step was to analyse the requirements of the program, which included identifying the data to be stored and the necessary operations to be performed on that data. In this case, the program needed to store information about items sold in a store and support operations such as adding, removing, updating, and searching for items.

Once the attributes had been identified, the next step was to determine the appropriate data types that would be used to represent them. In this program, ‘String’ data type is used for the name of the store, the store's address, and the item description. int data type is used for item ID and quantity in stock, while ‘BigDecimal’ is used to represent the item's unit price and total price. The parameters were used to pass necessary data between the main method and the Store class in each method.

Next, it was necessary to define the classes that would store the data. For this program, two classes: Store and Item. After that, the properties were defined so that they could be used to store the data. In the Item class, properties such as ID, description, unit price, quantity in stock, and total price were included. In the Store class, a property was added to store a list of items.

Following this, constructors (methods used to create instances of the class) were defined that would be used to perform operations on the data. For the Item class, methods were created to get and set the properties, and a ‘toString()’ method was added to return a string representation of an Item object. For the Store class, methods were created to add, remove, update, and search for items in the store.

The accessor methods (getters and setters) then needed to be defined for each attribute of the class, which are used to retrieve or update the values of the attributes. The Item class has accessor methods for each of its attributes, including the get methods for ID, description, unit price, quantity in stock, and total price.

After defining the methods, the structure of the classes needed to be refined to ensure that they were efficient and easy to use. This involved breaking down complex methods into smaller, more manageable methods and organising the code in a logical and readable way.

The program implements the Iterator pattern using the ‘ArrayList ‘class from the Java Collections framework, which provides a way to efficiently iterate over a collection of objects. Control structures, such as if-else statements and for loops, were used to manipulate data and iterate over collections.

Finally, it was time to test the program to ensure that it worked as intended. To achieve this, test cases and edge cases were created for each method, and the output was verified to ensure that it was correct.

# 

# Implementation details

**Information Flows**

This program involves two main types of information flow: file input/output and user input/output.

File Input/Output: The program uses two text files, "items.txt" and "transactions.txt", to store information about items and transactions in the store. These files are read by the program to load data into memory and written to when new or updated data needs to be stored.

User Input/Output: The program communicates with the user through the console by utilising the Scanner class. A menu of options is displayed, and the user can select an option by entering a number. Based on the user's input, the program performs actions such as adding, updating etc. . Additionally, the program generates a daily transaction report and outputs confirmation messages or reports to the user.

**A step-by-step breakdown of the different elements (classes, methods etc.) of the app:**

**Store Class**

Import statements:

These are used to import various classes that are needed in the program. For example, the java.io classes are used for reading and writing files, the java.util classes are used for various utility functions such as handling dates, and the ‘java.math.BigDecimal’ class is used for precise decimal arithmetic.

class:

*public class store {*

This is the main class of the program. The program starts executing from the main method inside this class.

main method:

*public static void main(String[] args) {*

This is the main entry point of the program. It initialises the files, loads items into memory, and starts a loop to allow the user to interact with the program.

Initialising files:

*File itemsFile = new File("items.txt");*

*File transactionsFile = new File("transactions.txt");*

*try {*

*itemsFile.createNewFile();*

*etc…*

This code initialises two files, "items.txt" and "transactions.txt", and creates them if they don't exist.

Loading items into memory:

*ArrayList<Item> items = new ArrayList<>();*

*try (Scanner fileScanner = new Scanner(itemsFile)) {*

*// Skip the first line if it is a header*

*Etc….*

This code loads the contents of the "items.txt" file into memory as a list of Item objects. It uses a Scanner object to read the file and creates an Item object for each line in the file.

User interaction loop:

*while (!exit) {*

*// Display menu options*

*System.out.println("\nPlease choose an option:");*

*System.out.println("1. Add a new item");*

*System.out.println("2. Update an existing item");*

*System.out.println("3. Remove an item");*

*System.out.println("4. Search for an item");*

*System.out.println("5. Generate daily transaction report");*

*System.out.println("6. Exit");*

*System.out.print("Option: ");*

*Etc….*

The loop runs until the user chooses to exit the program by selecting option 6. The menu options are displayed using ‘System.out.println()’ statements, and the user's input is read using a Scanner object. If the user enters an invalid input, an ‘InputMismatchException’ is caught, and the program displays an error message and continues the loop.

The program performs different actions based on the user's input using a switch statement. For example, if the user selects option 1, the ‘addItem()’ method is called to add a new item to the inventory. Similarly, if the user selects option 2, the ‘updateItem()’ method is called to update an existing item in the inventory.

generateReport method:

*private static void generateReport(File transactionsFile, ArrayList<Item> items) {*

*// Load transactions into memory*

*ArrayList<Transaction> transactions = new ArrayList<>();*

*try (Scanner fileScanner = new Scanner(transactionsFile)) {*

*etc…*

The method takes in two parameters - the transactionsFile which is a File object representing the transactions file, and items which is an ArrayList of Item objects representing the inventory.

It initialises an empty ArrayList of Transaction objects called transactions to hold the transactions from the file. It will then read the transactions from the file and creates a Transaction object for each line in the file. If a line has fewer than 6 parts (valid transaction), it is skipped.

The created Transaction objects are added to the transactions ArrayList. It initialises an empty ArrayList of Transaction objects called todayTransactions to hold the transactions that occurred on the current date. It gets today's date using the LocalDate.now() method. Next, it must loop through all the transactions in the transactions ArrayList and check if each transaction's date is equal to today's date. If it is, the transaction is added to the todayTransactions ArrayList.

It prints out the header for the daily transaction report and today's date, along with the details of each transaction that occurred on the current date in a formatted table. The details include the transaction ID, description, quantity sold, amount, stock remaining, and transaction type.

searchItem method:

*private static void searchItem(ArrayList<Item> items) {*

*Etc…*

The method begins by printing a prompt for the user to enter a search keyword.

The user's input is then read from the console using a Scanner object.

A new ‘ArrayList’ called ‘searchResults’ is created to hold the search results.

A loop is used to iterate through each item in the items ArrayList that was passed to the method.

For each item, the method checks if the item's description contains the search keyword (case-insensitive). If the keyword is found in the description, the item is added to the searchResults ArrayList.

Once the loop has finished, the method checks if any search results were found. If searchResults is empty, the method prints a message indicating that no items were found.

If search results were found, the method prints a table of the search results. The table has columns for item ID, description, unit price, quantity in stock, and total price.

removeItem method:

*private static void removeItem(ArrayList<Item> items) {*

*etc…*

The method starts by prompting the user to enter the ID of the item they want to remove using System.out.print("Enter item ID to remove: ").

The ID entered by the user is read from the console using int id = scanner.nextInt();.

Since scanner.nextInt() only reads the integer value and not the newline character, a call to scanner.nextLine() is made to consume the newline character left in the input buffer.

The method then initialises a variable itemToRemove to null. This variable will be used to store the item object that matches the ID entered by the user.

The method then loops through the items ArrayList using a for loop and checks if each item's ID matches the ID entered by the user. If a match is found, the itemToRemove variable is set to the item object and the loop is exited using the break statement.

If the itemToRemove variable is not null, it means an item was found that matches the ID entered by the user. In this case, the remove() method of the items ArrayList is called with the itemToRemove variable as an argument to remove the item from the ArrayList. The method prints "Item removed." to the console.

If the itemToRemove variable is null, it means no item was found that matches the ID entered by the user. In this case, the method prints "Item not found." to the console.

updateItem method:

*private static void updateItem(ArrayList<Item> items) {*

*etc…*

The method updateItem takes an ArrayList of Item objects as input. The method prompts the user to enter the ID of the item to update. The user's input is read in and stored in the id variable.

A loop is used to search through the ArrayList for an Item object with an ID that matches the user's input. If found, that Item object is stored in the itemToUpdate variable.

If the itemToUpdate variable is not null, meaning an Item object was found, the method prompts the user to enter a new description for the item. The user's input is read in and stored in the newDescription variable.

If the newDescription variable is not empty, the setDescription method is called on the itemToUpdate object to update its description.

The method then prompts the user to enter a new unit price for the item. The user's input is read in and stored in the newUnitPriceString variable. If the newUnitPriceString variable is not empty, the Double.parseDouble method is used to convert the string input to a double value and store it in the newUnitPrice variable.

If the input cannot be converted to a double, a NumberFormatException is caught and an error message is printed. Assuming that no error occurs and the newUnitPrice variable is not 0, the setUnitPrice method is called on the itemToUpdate object to update its unit price.

The method then prompts the user to enter a new quantity in stock for the item. The user's input is read in and stored in the newQtyInStockString variable. If the newQtyInStockString variable is not empty, the Integer.parseInt method is used to convert the string input to an integer value and store it in the newQtyInStock variable.

If the input cannot be converted to an integer, a NumberFormatException is caught and an error message is printed.

If no error occurs and the newQtyInStock variable is not 0, the setQtyInStock method is called on the itemToUpdate object to update its quantity in stock.

Finally, the setTotalPrice method is called on the itemToUpdate object to update its total price based on the new unit price and quantity in stock values.

[ If the itemToUpdate variable is null, meaning the item was not found in the ArrayList, a message is printed to indicate that the item was not found ]

addItem method:

*private static void addItem(ArrayList<Item> items) {*

*etc…*

The method begins by prompting the user to enter an item ID. Next, the method checks if the item ID already exists in the list of items. If the ID already exists, the method prints an error message and returns, preventing the user from adding a duplicate item. If the ID does not already exist, the method prompts the user to enter the item description, unit price, and quantity in stock. The method then calculates the total price of the item by multiplying the unit price by the quantity in stock. the method creates a new Item object with the provided information and adds it to the list of items. Of which it then prints a message indicating that the item was added, along with the total price of the item.

addTransaction method:

*private static void addTransaction(Transaction transaction, File transactionsFile) {*

*etc…*

The method addTransaction takes two parameters: transaction and transactionsFile.

The first line of the method uses a try-with-resources statement to create a FileWriter object to write to the transactionsFile. The true parameter in the constructor of the FileWriter indicates that the writer should append to the file if it already exists. The write method of the FileWriter object is called, which formats the transaction object as a string using the String.format method, and writes the resulting string to the file.

The String.format method formats the transaction data as a comma-separated string (Cutajar, J. (2018),, with the fields in the following order: transaction ID, transaction description, quantity sold, transaction amount, stock remaining, and transaction type.

If the write operation is successful, the message "Transaction recorded." is printed to the console. If an IOException occurs during the write operation, the catch block is executed and the message "Error writing to file: " followed by the exception message is printed to the console.

getNewTransactionId method:

*private static int getNewTransactionId(File transactionsFile) {*

*etc…*

Initiali the lastTransactionId variable to 0.

Open the transactionsFile using a Scanner.

Loop through each line of the file using a while loop and the hasNextLine() method.

Read the current line using the nextLine() method and split it into an array of String values using the split() method with a comma (",") as the delimiter. The resulting array will contain the transaction ID as the first element.

Parse the transaction ID String to an int using the Integer.parseInt() method and assign it to a local variable called id.

If the id is greater than the lastTransactionId, update lastTransactionId with the new id. If there are no more lines to read, the loop will exit.

If an IOException occurs during file reading, catch the exception and print an error message.

Return lastTransactionId + 1 as the new transaction ID.

(method reads the last transaction ID from the file, increments it by one, and returns the new transaction ID. This ensures that each new transaction is assigned a unique ID that is one greater than the previous transaction ID.)

recordTransaction method:

*private static void recordTransaction(Transaction transaction, ArrayList<Item> items, File transactionsFile) {*

*etc…*

The method takes in a Transaction object, an ArrayList of Item objects, and a File object as parameters. A null Item object is initialised to be updated with the new transaction data.

A for-each loop is used to iterate over each Item object in the ArrayList of Item objects. The loop checks if the ID of the current Item object matches the ID of the transaction object being passed in.

If there is a match, the current Item object is assigned to the itemToUpdate variable and the loop is exited. If there is no match, the loop continues to the next Item object in the ArrayList.

Once the loop has completed iterating over all the Item objects, itemToUpdate will either be the matched Item object or null if no match was found.

**Item Class**

Item(int id, String description, BigDecimal unitPrice, int qtyInStock, BigDecimal totalPrice): This is a constructor for the Item class that takes five parameters and initialises the corresponding fields of the class.

Item(int id2, String description2, BigDecimal unitPrice2, int qtyInStock2): This is another constructor for the Item class that takes four parameters but unit testing has shown it cannot be initialised to the totalPrice field.

Integer getId(): This is a getter method that returns the id field of the Item object.

double getTotalPrice(): Is a getter method that calculates and returns the total price of the item based upon its unit price and quantity in stock. Contains the necessary use of the multiply method of the BigDecimal class to perform the multiplication and setScale method to round the result to two decimal points.

String getDescription(): getter method - returns the description field of the Item object.

double getUnitPrice(): getter method - returns the unitPrice field of the Item object as a double.

int getQtyInStock(): getter method - returns the qtylnStock field of the Item object as an int.

**Transaction Class**

public Transaction(Date date, String customerId, List<TransactionItem> items): This is a constructor method that takes a Date object representing the transaction date, a String representing the customer ID, and a list of TransactionItem objects representing the items sold in the transaction.

public Transaction(int id, String description, int qtySold, double amount, int stockRemaining, String transactionType): This is another constructor method that takes several parameters representing different aspects of a transaction.

public Date getDate(): This method returns the transaction date as a Date object.

public Object getItemsAsString(): returns a string representation of the items sold in the transaction.

public double getAmount(): Returns the total amount of the transaction as a double.

public int getQtySold(): returns the number of items sold in the transaction as an int.

public int getItemPrice(): returns the price of the item sold in the transaction as an int.

public String getId(): returns the transaction ID as a String.

public String getDescription():returns the transaction description as a String.

public String getStockRemaining():returns the remaining stock of the items sold in the transaction as a String.

public String getTransactionType(): returns the type of transaction as a String.

public LocalDate getTransactionDate(): returns the transaction date as a LocalDate object.

**Classes and methods used**

java.io.BufferedReader: used for reading text from a character input stream.

java.io.BufferedWriter: used for writing text to a character output stream.

java.io.File: represents a file or directory path name.

java.io.FileNotFoundException: is a subclass of the IOException class and is thrown when an attempt to access a file that does not exist is made.

java.io.FileReader: used for reading character files.

java.io.FileWriter: used for writing character files.

java.io.IOException: general class of exceptions produced by failed or interrupted I/O operations.

java.math.BigDecimal: provides arbitrary-precision floating-point numbers.

java.text.SimpleDateFormat: is used to format and parse dates in a locale-sensitive manner.

java.time.LocalDate: represents a date (year, month, day) in the ISO calendar system.

java.util.ArrayList: is used to create a resizable array, which can be used to store a collection of objects.

java.util.Date: represents a specific instant in time, with millisecond precision.

java.util.HashSet: is used to create a collection that uses a hash table for storage.

java.util.InputMismatchException: is thrown by a Scanner to indicate that the token retrieved does not match the pattern for the expected type.

java.util.Scanner: is used to parse primitive types and strings using regular expressions.

java.util.Set: This interface represents a collection that contains no duplicate elements

The methods used in the program are used to perform various operations on the objects created from these classes. These methods include creating a new file, reading from and writing to files, parsing text data, creating and manipulating objects, and filtering data.

The main method is used to implement the program's menu-driven functionality, while the other methods are used to carry out specific tasks for items in the store's inventory.

**Software Design Principles implemented**

**Store class**

Single Responsibility Principle (SRP): The code is organized into methods and classes that have a single responsibility. For example, the generateReport() method is responsible for generating a daily transaction report, and the searchItem() method is responsible for searching for an item.

Open-Closed Principle (OCP): program is designed to be open for extension but closed for modification. For instance, the addItem(), updateItem(), removeItem(), searchItem(), and generateReport() methods can be easily extended to support new functionality without modifying the existing code.

Dependency Injection Principle (DIP): The program uses constructor injection to inject dependencies into its classes. As a a result, the Transaction class depends on the LocalDate class, which is injected through its constructor.

Don't Repeat Yourself (DRY): avoids duplicating code by reusing existing methods and classes. For example, the code that reads items and transactions from files is reused in several methods.

Separation of Concerns (SoC): The program separates its concerns into different classes and methods. The Transaction class is therefore responsible for representing a transaction, while the store class is responsible for managing items and transactions.

**Item class**

Encapsulation: The class uses private instance variables to hide implementation details and provide access to those variables through public methods *(Samoylov, N. (2018).* This allows for better control over how the variables are accessed and modified, which helps to prevent unintended side effects.

Single Responsibility Principle: The class has a singular responsibility of representing an item with its properties and behaviour.

Constructor Overloading: The class has multiple constructors, which is a form of constructor overloading. This allows for different ways to create instances of the class depending on the parameters passed in.

Immutability: The instance variables are declared as private and are only set in the constructor, which makes the class immutable. This can have benefits such as simplifying the code, improving thread-safety and making the code more reliable.

BigDecimal for monetary calculations: The code uses the BigDecimal class for monetary calculations since it provides more precise and accurate results than floating-point numbers.

**Transaction class**

Single Responsibility Principle (SRP): responsible for representing a transaction and storing its properties, but it should not be responsible for formatting the output of the transaction or computing derived values.

Encapsulation: class encapsulates the data related to a transaction, such as a date, customer ID, and list of items.

Open-Closed Principle (OCP): Transaction class could be extended to support additional types of transactions or properties without modifying the existing implementation.

Dependency Inversion Principle (DIP): Transaction class may be improved if made to depend on abstractions such as interfaces or abstract classes instead of concrete implementations of dependencies such as Date or List<TransactionItem>.

**Programming Concepts used**

Structured programming concepts used in the program are:

sequential execution: the program is executed step-by-step, with each section being executed in the order in which it appears in the code.

selection: the program uses switch statements to choose between different options based on user input.

iteration: uses a while loop to repeatedly display the menu until the user chooses to exit.

Exception handling: program uses try-catch blocks to handle exceptions such as file not found, IO exceptions, and number format exceptions.

Procedural programming concepts used in the program are:

functions: the program uses several functions to perform specific tasks, such as adding a new item or generating a transaction report.

variables: the program uses variables to store data, such as the list of items and the list of transactions.

parameters: some functions take parameters, such as the list of items or the file containing transaction data.

Object-oriented programming concepts used in the program are:

Classes and Objects: The program defines classes for Item and Transaction, which are used to create objects to represent items and transactions.

encapsulation: The Item and Transaction classes encapsulate their data and behaviour by defining private fields and public methods to access and modify them.

Constructors: The Item and Transaction classes define constructors to create objects and initialise their state.

Polymorphism: The program indirectly uses it by calling methods on objects of different classes using a common interface (i.e. the ArrayList class).

**Purpose of Dependencies**

The program reads and writes data to text files, so it needs the File, FileReader, FileWriter, BufferedReader, and BufferedWriter classes. It also needs to handle exceptions related to file I/O, so it needs FileNotFoundException and IOException..

It stores items and transactions in memory using ‘ArrayLists’, so it needs ‘ArrayList’. It also needs to store the date of each transaction, so it needs ‘LocalDate’. It uses ‘BigDecimal’ to represent the unit price and total price of each item. It uses Date to represent the date and time of each transaction, but this class has been replaced by LocalDate to correct method errors.

It also allows users to enter numeric input, so it needs ‘Scanner’. And lastly, it needs ‘InputMismatchException’ to handle invalid input.

# Discussion

One aspect of the code that went well is the use of object-oriented programming to create classes for Item and Transaction. These classes encapsulate the data and behaviour associated with items and transactions in order to make the code more modular and easier to maintain. Additionally, the application uses appropriate data structures (e.g., ArrayLists, Sets) to store and manipulate the data.

One limitation of this program is the lack of error checking and validation for user input and file input, which could lead to incorrect results or program crashes *(Burd, B. and Burd (2014)* if invalid data is entered. Moreover, the program does not currently support concurrent access by multiple users, which could be a limitation for larger stores with multiple employees.

Another improvement would be to modularize the code further and break it up into smaller functions. For example, the code for generating the daily transaction report could be split into multiple functions to improve readability and maintainability *(Burd, B. and Burd (2014)* . Furthermore, the code could benefit from better error handling and logging to help diagnose issues that may arise during runtime.

Overall, this program achieves the main objective of providing a store inventory management system with basic functionality. The use of object-oriented programming and appropriate data structures make the code modular and easy to maintain. However, there are limitations and areas for improvement, namely that which is associated with the ‘Transaction.txt’ file and class due to the need for additional features like printing the report based upon the values written and stored in ‘item.txt’, with the program being particularly problematic for configuring the amount of stock remaining after each transaction. Further Logical analysis and unit testing is therefore needed for future development purposes.

# References

*Clark, M., & Anstey, M. (2019). Java in easy steps (6th ed.). In Easy Steps Limited.* ***[see Appendix A]***

*Burd, B. and Burd (2014) Beginning Programming with Java for Dummies* ***[see 3. Discussion]***

*Samoylov, N. (2018) Introduction to Programming: Learn to Program in Java with Data Structures, Algorithms, and Logic.* ***[see 2. Implementation]***

*Cutajar, J. (2018) Beginning Java Data Structures and Algorithms: Sharpen Your Problem Solving Skills by Learning Core Computer Science Concepts in a Pain-Free Manner.* ***[see 2. Implementation]***

**Text files**

“items.txt”

*id,description,unitPrice,qtyInStock,totalPrice*

*1,Blue T-Shirt,19.99,50,999.50*

*2,Black Jeans,49.99,30,1499.70*

*3,White Sneakers,39.99,25,999.75*

*4,Red Hoodie,29.99,40,1199.60*

*5,Green Shorts,14.99,20,299.80*

“Transactions.txt”

*date,id,description,qtySold,amount,stockRemaining,transactionType*

*2023-04-01,1,Blue T-Shirt,10,199.90,40,Added*

*2023-04-02,2,Black Jeans,5,249.95,25,Updated*

*2023-04-03,3,White Sneakers,2,79.98,23,Removed*

*2023-04-04,4,Red Hoodie,15,449.85,25,Added*

*2023-04-05,5,Green Shorts,8,119.92,12,Updated*

**Input and output examples**

Menu:

**Input**

*Please choose an option:*

*1. Add a new item*

*2. Update an existing item*

*3. Remove an item*

*4. Search for an item*

*5. Generate daily transaction report*

*6. Exit*

*Option:*

Option 1 display:

**Input**

*Option: 1*

*Enter item ID: 7*

*Enter item description: g*

*Enter unit price: 23.44*

*Enter quantity in stock: 5*

**Output**

*Total price: 117.20*

*Item added.*

Option 2 display:

**Input**

*Option: 2*

*Enter ID of item to update: 4*

*Enter new description for item: f*

*Enter new unit price for item: 44.88*

*Enter new quantity in stock for item*: 3

**Output**

*Item updated: 4,f,44.88,3,134.64*

*Item file updated.*

Option 3 display:

**Input**

*Option: 3*

*Enter item ID to remove: 3*

*Item removed.*

**Output**

Item.txt [written to]

*1,Blue T-Shirt,19.99,50,999.50*

*2,Black Jeans,49.99,30,1499.70*

*4,Red Hoodie,29.99,40,1199.60*

*5,Green Shorts,14.99,20,299.80*

Option 4 display:

**Input**

Option: 4

Enter item ID: 4

**Output**

*ID Description Unit Price Qty in Stock Total Price*

*4 Red Hoodie 29.99 40 1199.60*

Option 5 display:

**Input**

Option: 5

**Output**

*Today's Transaction Report:*

*Date: 2023-04-26*

*date,id,description,qtySold,amount,stockRemaining,transactionType*

*2023-04-01,1,Blue T-Shirt,10,199.90,40,Added*

*2023-04-02,2,Black Jeans,5,249.95,25,Updated*

*2023-04-03,3,White Sneakers,2,79.98,23,Removed*

*2023-04-04,4,Red Hoodie,15,449.85,25,Added*

*2023-04-05,5,Green Shorts,8,119.92,12,Updated*

**Testing [unit tests; test cases; edge cases; input scenarios]**

***[See Program Test files under “.test uses” folder. Contains comments for each scenario ;***

***Pseudocode - under Tests labelled ‘Pseudocode.txt’]***

**Error Messages; Improvements**

**Store Class**

generateReport method

1. code added to compare the items sold in each transaction with the inventory to ensure that there is sufficient stock before selling. Here’s the implementation:

*// Check if there is sufficient stock for the transaction*

*for (Item item : items) {*

*if (item.getDescription().equalsIgnoreCase(description)) {*

*if (item.getStock() >= qtySold) {*

*// Reduce the stock of the item*

*item.setStock(item.getStock() - qtySold);*

*// Add the transaction to the transactions list*

*transactions.add(transaction);*

*} else {*

*System.out.println("Error: Not enough stock for transaction with id " + id);*

*}*

*break;*

*console output:*

*“Exception in thread "main" java.lang.Error: Unresolved compilation problems:*

*The method getStock() is undefined for the type Item*

*The method getStock() is undefined for the type Item*

*at store.generateReport(store.java:163)*

*at store.main(store.java:134)”*

1. To fix this error, the Item class now has a getStock() method that returns the current stock of the item. Here is the implementation:

*public class Item {*

*private int id;*

*private String name;*

*private double price;*

*private int stock;*

*// constructor and other methods*

*public int getStock() {*

*return stock;*

*}*

}

*console output:*

*“Exception in thread "main" java.lang.Error: Unresolved compilation problem:*

*The method setStock(int) is undefined for the type Item*

*at store.generateReport(store.java:165)*

*at store.main(store.java:134)”*

1. The error "The method setStock(int) is undefined for the type Item means that the Item class does not have a setStock() method defined. To fix this, the setStock(int) method is added to the Item class. Here is the implementation:

*public void setStock(int stock) {*

*this.stock = stock;*

*}*

searchItem method

1. Code added to: create a new scanner object to read user input from the console ; using the keyword variable to search for items in the items ArrayList ; updating the print statements to match the requested output format. Here’s the implementation:

*Scanner scanner = new Scanner(System.in);*

----

*String keyword = scanner.nextLine();*

----

*System.out.printf("%-5d %-20s %-10.2f %-15d %-15.2f\n", item.getId(), item.getDescription(), item.getUnitPrice(), item.getQtyInStock(), item.getTotalPrice());*

*Console output:*

*Exception in thread "main" java.lang.Error: Unresolved compilation problem:*

*Type mismatch: cannot convert from BigDecimal to double*

*at Item.getUnitPrice(Item.java:40)*

*at store.searchItem(store.java:223)*

*at store.main(store.java:131)*

1. Error is fixed by changing the method getUnitPrice() in the Item class to return a double instead of a BigDecimal. Here’s the implementation:

*public double getUnitPrice() {*

*return unitPrice.doubleValue();*

*}*

**----**

*public double getTotalPrice() {*

*return unitPrice.multiply(new BigDecimal(qtyInStock)).doubleValue();*

*}*

removeItem method

1. The main change is the addition of a Scanner object that is used to read user input from the console. The Scanner is created at the beginning of the method and used to read the user's input for the item ID to remove. The Scanner object is then used to consume the newline character that is left in the input buffer after reading the integer value.

*Error with “items.txt” file display as item is removed*

1. Needed to add a ‘toString()’ method to override in the Item class which is properly defined. To properly format the output of the ‘Item’ objects, a string returns the fields in a comma-separated format, with the unit price and total price rounded to 2 decimal places. Contents of the ‘items’ ArrayList are now printed to match the format of the “items.txt” file. Here’s the implementation:

*public class Item {*

*// ... class definition ...*

*@Override*

*public String toString() {*

*return String.format("%d,%s,%.2f,%d,%.2f",*

*id, description, unitPrice, qtyInStock, totalPrice);*

*}*

}

updateItem method:

1. updates to the items list are not being saved to the file. The following code is added to the end of the ‘updateItem’ method to write the updated list to the file:

*try {*

*PrintWriter writer = new PrintWriter(new FileWriter("items.txt"));*

*for (Item i : items) {*

*writer.println(i.getItemCode() + "," + i.getItemName() + "," + i.getUnitPrice() + "," + i.getQtyInStock() + "," + i.getTotalPrice());*

*}*

*writer.close();*

*} catch (IOException e) {*

*System.out.println("Error writing to file: " + e.getMessage());*

*}*

This code creates a new PrintWriter object that writes to the "items.txt" file, and then loops through the items list to write each item to the file in the correct format. Finally, it closes the writer to ensure that the changes are saved to the file.

1. Error occurs because a ‘double’ value is passed as an argument to the ‘setUnitPrice’ method which expects a ‘BigDecimal’ value as shown:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

*The method setUnitPrice(BigDecimal) in the type Item is not applicable for the arguments (double)*

*at store.updateItem(store.java:285)*

*at store.main(store.java:127)*

to fix this error, a ‘BigDecimal’ value is instead passed to the ‘setUnitPrice’ method, as shown:

*item.setUnitPrice(new BigDecimal(Double.toString(unitPrice)));*

addItem method:

1. adding some error handling to handle invalid user input, such as entering a negative quantity or a negative unit price. As shown:

*} catch (IllegalArgumentException e) {*

*System.out.println(e.getMessage());*

*qtyInStock = -1; // Reset to -1 to prompt user to enter a valid quantity*

*}*

*}*

1. the saveItemsToFile() function is not being called after adding the new item, which means that the changes are not being saved to the fileUpdated version:

*items.add(item);*

*System.out.println("Item added.");*

*// Save the changes to the file*

*saveItemsToFile(items, "items.txt");*

*System.out.printf("Total price: %.2f\n", totalPrice);*

**Transaction Class:**

1. Modify the constructor to initialise the class variables and make them private. This can be achieved by using the "this" keyword to reference the class variables in the constructor.
2. Replace the "Date" class with the "LocalDate" class, which is a “more modern and recommended class” *(Clark, M., & Anstey, M. 2019*) for working with dates.
3. Add getters and setters for all the class variables to provide access to them outside the class.
4. Update the method signatures to use the correct return types and parameter types.
5. Remove unused methods that do not provide any value to the class.

# A screenshot of a computer Description automatically generatedAppendix B

A screenshot of a computer

Description automatically generated

**Flowcharts**

[based on Pseudocode]

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated