

**OBJECT ORIENTED PRINCIPLES**

ASSIGNMENT3 DS: A JAVA APPLICATION FOR MANAGING A COMPUTER STORE.

**Declaration of Authorship**

I, Daniel Soden, declare that the work presented in this assignment titled ‘A Java Application for Managing a Computer Store’ is my own. I confirm that:

* This work was done fully by me as part of my degree in Software Development at MTU.
* Where I have consulted the published work and source code of others, this is always clearly attributed.
* In any situation I have looked for external sources, they have been quoted within this report.

On 01/12/2024

Signature: Daniel Soden

**Dependencies:**

Although Maven should for the most part ensure this code will work on any machine, with my usage of SQLite I think my video demonstration would be the best way to view the work I have put into this project. I had to do a lot of shuffling of packages to get SQLite to work for me initially so I hope it will still work for you too. In the future if I were to do this project again I would put it in a docker container so that then there would be no trouble at all with building the project and running the code.

* Maven (Used to manage packages, builds and directories rather than intellij’s standard method). This is also why my main is in org.example package.
* OpenJDk23 – I.E. Java 23.
* SQLite3 – Database used for program.
* [Note] For the SQLITE database in the zip file the computer products will have been inserted, however upon initialization from an independent install, this may need to be implemented manually.

**Java Application Description.**

This Java application provides an interactive, text menu-based, session for managing and purchasing computer products. The functionality of the application includes:

* Adding/removing users from the shop.
* Displaying the info of a user/computer.
* For users to purchase products from the Shops;
* For the user/store content to be loaded/stored to a SQLite3 database. – Resource for sqlite (<https://www.sqlitetutorial.net/sqlite-java/>) - Main usage of resource was for the boilerplate before any SQL code is made, after the boilerplate I declare all SQL code is my own. On top of this for demonstration purposes, I have filled in some SQL data manually via the SQLite CLI

**Technical Difficulty: OOP Concepts Demonstrated in the Java Application.**

1. **Primitive and Reference Variables.**
   * The class Order.java has the fields productID and customerID (an int and, therefore, primitive variable).
   * ShopImp.java uses a reference variable to Order o by using it to access the various values and methods provided by order, in particular, toString in order to create a receipt.
2. **Classes and Objects.**
   * The class Customer.java models a user of the Shop, and the method ShopImp.java::addCustomer creates a new Customer object.
3. **Encapsulation.**
   * The class Computer.java has a public field gbRam, and public methods getRam and setRam methods to access/update the field from other classes. This encapsulates the concept of a computer of which you can check the ram and change it when needed and then update this on the system
4. **Inheritance.**
   * The class Product.java inherits from Agent.java.
5. **Class Hierarchy.**
   * The class Computer.java inherits from Product.java which also inherits from Agent.java. Therefore, there is a class hierarchy, where computer is a child class of the parent Product and the Parent of product is Agent, making Computer the grandchild of Agent.
6. **Static Polymorphism (overloading).**
   * The class MyMain.java has two versions of the method scanInt each of them with a different signature in which one takes in a particular range and the other takes in any integer.
7. **Dynamic Polymorphism (overwriting).**
   * The class Computer.java overwrites the method getAmountOnOrder specified in the class Product. In the new implementation we retrieve the new amount on order through an SQL Query.
8. **Interface.**
   * The class Shop.java is an interface, modelling the management of a computer Shop (via methods as purchaseItem, saveReceipt, etc). The interface is implemented in the class ShopImp.java.
9. **User/Developer Isolation.**
   * By using abstract data types we can isolate our own developed methods from a user and how they are implemented to allow a user to use these methods without worrying about the logic of the program
     + Assuming the class Main.java was implemented by our first programmer. They can look at Shop.java, create an instance of type Shop to use all its functionality (the methods addCustomer, saveReceipt , etc.) all not knowing how all this functionality is implemented.
10. **Static Fields and Methods.**
    * The class Database.java has a static field url. Therefore, the field does not belong to a single object of the class, but to all objects of the class.This decision was made so I could access this field universally for database connectivity.
    * The class Computer.java, to retrieve all data related to a particular product via a given id. As a public static method, it can be called from any class and is used within the ShopImp class as it aids in preventing multiple SQLite connections causing an error.
11. **Final Fields, Methods and Classes.**
    * The class User.java has a final field name, as once it is defined, it cannot be modified.
    * The class Agent.java has a final method getId, so that no other class inheriting from Agent.java (for example, Customer.java or Product.java) can overwrite the method and compute the id in a different way.
12. **Exception Handling.**
    * The methods relating to the Sqlite connection all contain exception handling in that if it is unable to reach the Structured query language data, it will give back an sql, like exception.
    * I have also implemented type checking for various user inputs via the
13. **File Reading and Writing.**
    * Although not directly file reading, throughout the course of my application, espeically for listItems and listRecentOrders we are reading from the shop.db database file.
    * The method saveReceipt takes in an order O and from here is called within purchaseItem as an optional method to save the contents of their order to a txt file receipt**.**

**14. Abstract Methods**

* In the class Product an abstract method getAmountOnOrder is created in an abstract way and from there it is implemented in the Computer class.

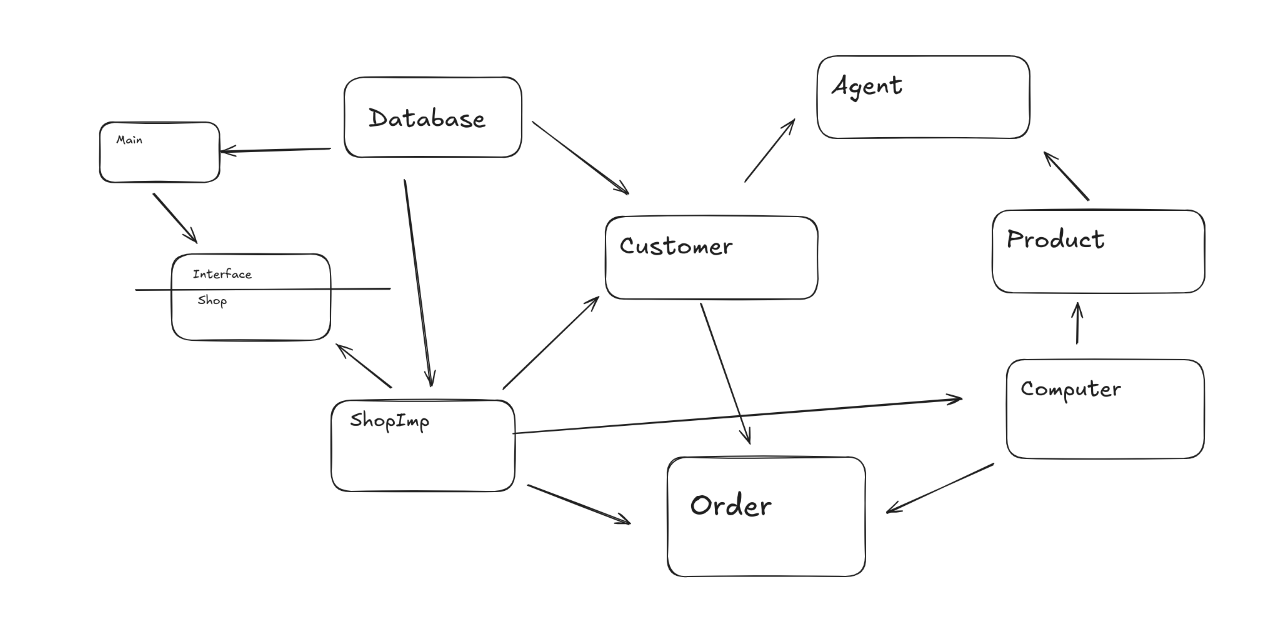
15. Java Generics

* Although not exactly strictly a java generic, my usage of an ArrayList with a datatype of Object, allows for a generic usage of various datatypes, wheter these objects are primitive or non primitive datatypes. I use this so that I can extract the integers doubles, and strings that come from the products.

16. Data Structures

* In the same breath we also used an Array datastrcuture to retrieve the products cost,name and id by indexing into this array. This was done to expand the scope of this data rather than keep it to an SQL query.

**UML Design: Java Application.**

**Testing the Java Application.**

My application is completely based off text inputs as seen by the scan(<GivenDatatype>) Methods in the main, ensuring the user gives both the correct length and that they give a variable of some sort. However despite the extra time I’ve had, I keep running into issues with how scanners work, it seems to be something outside of my own control as I have consulted many sources and still haven’t found a solution.

Overall I am quite happy with the Project I have created given my limited knowledge on certain aspects of OOP and my first time ever connecting a Relational Database Management system into a standard Cli application. There are definetly more features I would add but for now I am quite satisfied with my work.

**Video Link :** <https://youtu.be/u1786H5KegM>