# Assignment 3

Due May 27th 5pm on AUTOnline

## Product Recommender Algorithm

## Brief

For this assignment, you will individually develop an Online Shop in Java that implements a Product Recommender Algorithm (PRA) that recommends related products based on your purchases. The PRA works by analysing transaction histories of Online Shop customers, observing the frequency products are purchased together. Products with high purchase frequencies with your current purchases are recommended to you.

## Example

A three-digit integer number represents each product offered by the Online Shop. For example, the numbers 123, 187, 199, 200, 865 represent unique types of products sold by the Online Shop. A customer's purchase history is therefore a list of numbers. Here are the purchase histories from four different customers:

1. [187, 199, 200]
2. [187, 199, 123, 865]
3. [199, 123, 865]
4. [123, 199, 865, 187]

Suppose you purchase product 187. The PRA analyses each customer's transaction histories. If 187 is found in the history then it keeps track of the other products that customer purchased, creating an observation map from products to frequencies:

123 -> 2 (e.g. 123 was purchased with 187 by customers 2. and 4.)
199 -> 3 (e.g. 199 was purchased with 187 by customers 1., 2. and 3.)
200 -> 1 (e.g. 200 was purchased with 187 by customers 1.)
865 -> 2 (e.g. 865 was purchased with 187 by customers 2. and 4.)

The PRA will then return products 123, 865 and 199 as recommendations since the observation frequency is greater than/equal to 2. For larger product databases and more customers, we might have recommendations based on higher frequencies.

# Methodology

You will develop Java classes that implement the Online Shop. For full marks, your classes must adhere to correct OOP design principles.

All source code must be documented according to Java doc standards.

## Modelling Products

Design an abstract **Product** class that stores a product's code as an Integer and its price. The **Product** class implements the **Comparable** interface in order to compare products.
The Product class is extended by *at least five* concrete classes.
Each subclass must have

- two or more attributes,
- a constructor with input parameters for all attributes, including inherited attributes
- a toString method

## The product database and purchase history classes

Products are stored in a **ProductDatabase class**, which comprises a

- **HashMap** collection, mapping product codes (integers) to **Product** objects.
- constructor, initialising the **HashMap**
- a toString method, and
- functionality for adding products to the database and returning the products stored in the database in an **ArrayList**.

Note: You do not need to remove products from the database.

The **PurchaseHistory** class models the purchase history of a customer. It contains an **ArrayList** of product objects, supplied by the constructor. The class has a single get method for the **ArrayList** but no set or other updating methods.

## Payment System Functionality

The Payment System is modelled as an interface, comprising the methods:

- `public Double amountOwing();`
- `public void completeTransaction();`

The payment system is implemented by the Online Shop, which provides method bodies to compute the amount the customer owes. The complete transaction method simply clears the shopping cart.

## The Recommender System

The recommender system class has instance variables for the product database and a set containing customer purchases history objects. The constructor initializes both instance variables. This class has a single method with the signature

```
ArrayList<Product> praAlgorithm(ArrayList<Product> cart,int freq)
```

with the customer's shopping cart and a frequency as input, returning a list of recommended products. Consider the following input:

| Method Invocation | Output |
|---|---|
| praAlgorithm([187],2) | [123,199,865] |
| praAlgorithm([187],3) | [199] |
| praAlgorithm([200],2) | [] |
| praAlgorithm([200],1) | [199,187] |
| praAlgorithm([123],2) | [199,865,187] |
| praAlgorithm([123],3) | [199,865] |

Now look at the example and try to calculate the output yourself. Use this process to devise an algorithm to compute the recommendations.

## Online Shop

The Online Shop is modelled by a Java class that contains a product database and a recommender system. It maintains a shopping cart that contains products the customer has selected to purchase. The Online Shop implements a payment system to compute the amount owing and completing the customer's transaction (which also clears the cart).

## Online Shop Application (Program Interaction)

The Online Shop Application provides interaction between the customer and an Online Shop object. It provides menu interactions for

- Adding a product to the cart, which lists all products in the database, from which the customer can select from
- Viewing the products in the shopping cart
- Finalising purchases, which uses a payment system to compute the total amount owing and to process the transaction. When the payment is completed, the customer is given product suggestions based on what they have purchased.

*All product lists displayed by the Online Shop Application must be sorted by price, in ascending order.*

## How to get started

The first step to getting started on the assignment is creating the **Product** abstract class and deciding how the class should be extended using at least 5 subclasses. Think of interesting examples of products to sell. **They must be different from the ones shown in the console sample provided; otherwise you will get zero marks for that portion of the assignment.** The point is to think about developing your own classes.

Once you have finished this, you can instantiate products with product codes 123, 187, 199, 200 and 865 (from the example) and generate a sample product database. I suggest that you write a static method with the signature

```
ProductDatabase generateSampleDatabase()
```

in your Online Shop Application class, which initialises and returns a product database object with these five products.

The text file **purchase-history.txt** is provided to you with this assignment.
It contains the purchase histories of the four customers shown in the example.
In your Online Shop Application class, write a static method which reads this text file, and has the following signature:

```
HashSet<PurchaseHistory> readPurchaseHistoryData(ProductDatabase pb,String filename)
        throws ProductNotFoundException,IOException,NumberFormatException
```

This method reads the text file and throws the following exceptions:

1) IOException, thrown if the file cannot be opened/read
2) NumberFormatException, thrown if integer parsing fails
3) ProductNotFoundException, your own exception, which is thrown if a product code does not appear in the database.

Your main method should **try** to read the purchase history data, handling exceptions in an appropriate way (potentially by terminating execution with a meaningful message, but not by an uncaught exception).

Note: Your application should elegantly handle exceptions occurring from bad user input.

## Marking Scheme

| Criteria: | Weight: | Grade A<br>Range<br>100 ≥ x ≥ 80% | Grade B<br>Range<br>80 > x ≥ 65% | Grade C<br>Range<br>65 > x ≥ 50% | Grade D<br>Range<br>50 > x ≥ 0% |
|---|---|---|---|---|---|
| **Product and subclasses** | **20%** | OOP paradigm consistently used for implementation of all product classes. Meaningful subclasses are defined. | Inconsistent use of OOP paradigm. Correct implementation of product class functionality | Inconsistent use of OOP paradigm or poor product subclasses. Correct implementation of product class functionality | Absent product classes/reused product classes or code does not compile |
| **Product Database & Purchase History** | **10%** | OOPS paradigm consistently used. Good use of collections in both classes, with correct functionality implementation | Inconsistent use of OOP paradigm. Correct use of collections with correct functionality implementation | Inconsistent use of OOP paradigm. Incorrect use of collections but with mostly correct functionality implementation | Absent product database or purchase history or code does not compile |
| **Online Shop** | **10%** | OOP paradigm consistently used for implementation of all Online Shop functionality. | Inconsistent use of OOP paradigm but correct implementation of Online Shop functionality | Some basic functionality of Online Shop/poor usage of collections | Absent functionality of Online Shop or code does not compile. |
| **Recommender System** | **10%** | OOP paradigm consistently used. Correct implementation of algorithm with good choice of collections. | Inconsistent use of OOP paradigm. Correct implementation of algorithm with good choice of collections. | Incorrect implementation of algorithm/poor choice of collections | Absent functionality of the recommender system or code does not compile |
| **Online Store App Runtime Demonstration:**<br>**-Uniqueness**<br>**-Purpose**<br>**-Context**<br>**-Interactive** | **20%** | The object-oriented program is unique, purposeful and provides an elegant implementation of the Online Shop Application.<br><br>Program is interactive.<br>All exceptions are handled.<br>Purchase history text file is read<br>Sample product database generated. | The object-oriented program is unique, purposeful. Reasonable implementation of the Online Shop Application.<br>Program is interactive.<br>Some errors reading text file or generating product databases. | The object-oriented program features an incomplete demonstration of the Online Shop Application<br><br>Program is not interactive | Absent functionality of Online Shop Application or code does not run after compiling |
| **Code Quality:**<br>**-Whitespace**<br>**-Naming**<br>**-Reuse**<br>**-Modularity**<br>**-Encapsulation** | **15%** | Whitespace is comprehensively consistent. All naming is sensible and meaningful. Code reuse is present. Code is modular. Code is well encapsulated. | Whitespace is comprehensively consistent. Majority of naming is sensible. Code is modular. Code is encapsulated. | Whitespace is comprehensively consistent. Code has some modularity. Code has some encapsulation. | Whitespace is inconsistent and hence code is difficult to read. |
| **Documentation Standards:**<br>**-Algorithms Commented**<br>**-Javadoc** | **15%** | Entire codebase has comprehensive Javadoc commenting. Algorithms are well commented. | Majority of the codebase features Javadoc commenting. Majority of algorithms are commented. | Some Javadoc comments present. Some algorithms are commented. | No Javadoc comments present. |

## Authenticity

All work submitted must be unique and your own!

We use automated methods to detect academic integrity breaches.

## Submission Instructions

Submit the following documents as an archive **.zip** file of your documents on AUTOnline before the deadline:

- Your source code (**.java** files).
- Sample console output demonstrating your program in use (**.txt** file)

Zip structure and file naming requirements. Please ensure your submission matches the following:

**lastname-firstname-studentid.zip**
- ***.java**
- **studentid-console-sample.txt**

Replace the underlined text with your personal details.

## Late submissions will receive a grade of 0

An extension will only be considered with a Special Consideration Form approved by the School Registrar. These forms are available at the School of Computer and Mathematical Science located in the WT Level 1 Foyer.

You will receive your marked assignment via AUTonline. Please look over your entire assignment to make sure that it has been marked correctly. If you have any concerns, you must raise them with the lecturer. You have **one week** to raise any concerns regarding your mark. After that time, your mark cannot be changed.

*Do not email the lecturer because you do not like your mark. Only go if you feel something has been marked incorrectly*.