# Systems Programming
## Module-2 (Packages. Data Structures)

## 1. Introduction

The project for the management of the warehouse (store) we have been working on has already completed its first phase and now we are moving on to the second phase.

- During the first phase (Module-1) the programming department worked on modelling the different elements involved in the management of the store and on designing the text menus that will serve to interact with the program.
- During the second phase (Module-2) the programming department will work on the logic of the application and on providing functionality to the menus created during the first phase. If the methods requested in module-2 have already been implemented in module-1, they must be reviewed to see if they meet the new specification.

## 2. Organization of the code of the store in packages

Before we go any further, as our code has grown quite a bit in the first phase, we are going to proceed to reorganize our classes to facilitate the addition of new classes and the maintenance of the code for the future.

- Organize your classes in folders (packages) as shown in Figure1-a. In Aula Global you can download a skeleton of some of the classes.

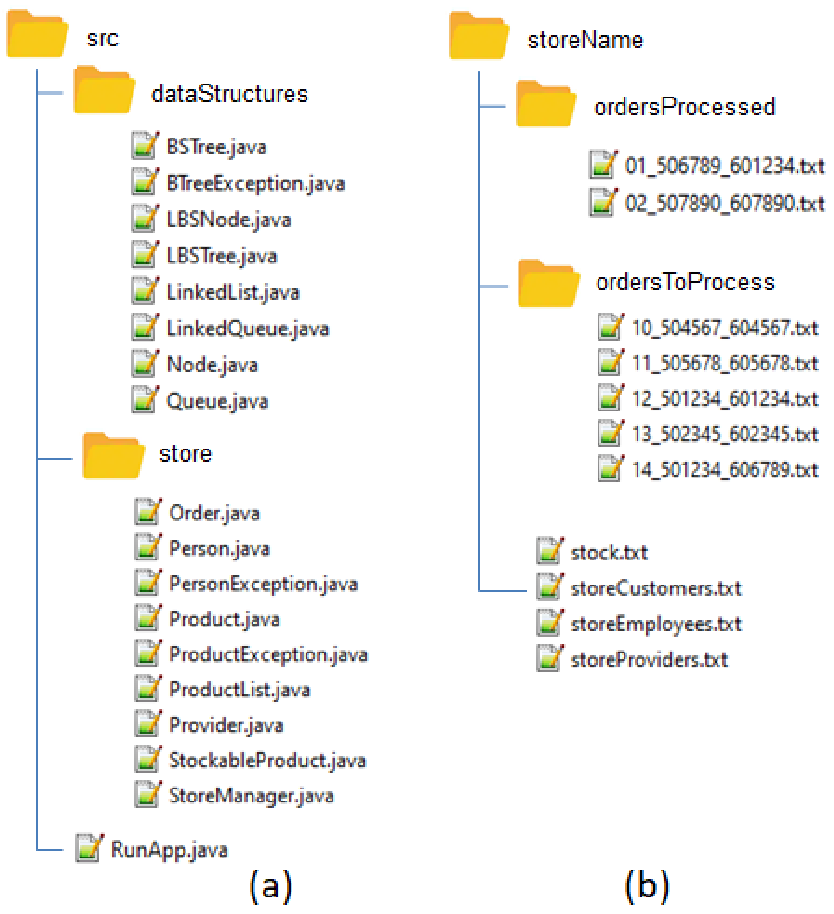


Figure-1: Project file structure.

- Replace the skeleton of the classes with those of your own project and make the necessary modifications to organize them into two packages. The classes in the `store` folder belong to the `store` package, those in the `dataStructures` folder belong to the `dataStructures` package and only the `RunApp` class does not belong to any package.

### 3. Organization of the information of the store in files

The `storeName` folder represents a specific store named "*storeName*". It contains two folders and several files (see Figure 1-b). All files (unless otherwise indicated) store information in the formats corresponding to the `toString()` methods indicated in module-1.

- Files:
    - *stock.txt* contains the list of products in the store.
    - *storeCustomers.txt* contains the list of persons who are customers of the store.
    - *storeProviders.txt* contains the list of companies that are providers of the store.
    - *storeEmployees.txt* contains the list of persons who are employees of the store.
- Folders:
    - *ordersProcessed* is the folder that contains the orders already processed.
    - *ordersToProcess* is the folder that contains the orders to be processed.
    - Both folders contain a set of files, each of which represents an order. Each order is characterized by:
        - The name of the file in the following format: *orderID_customerID_employeeID.txt* where: (1) `orderID`[1] is the order identifier; (2) `customerID` is the identifier of the customer who placed that order; and (3) `employeeID` is the identifier of the employee in charge of processing that order.
        - The content of the file, which is a set of products (`StockableProduct`) in the format indicated in the `toString()` in module-1[2]: `productName|productBrand|category|isCountable|measurementUnits|numUnits|costPerUnit|pricePerUnit|contactID|providerName|providerAddress|contactID|contactFirstName|contactLastName|contactEmail`

### 4. Modification of the basic classes. Methods `equals()` and `compareTo()`

In order to provide the functionality for the store you will have to modify some of the classes already implemented in module-1 to include the methods `compareTo()` of the `Comparable` interface, and `equals()`, according to the following indications:

**Modification of classes `Person`, `Provider` and `StockableProduct`**

- Add to the `StockableProduct` class an attribute of type `Provider` to indicate the provider of that product.
- Modify the classes `Person`, `Provider` and `StockableProduct` to implement the `Comparable` interface.
    - Persons (`Person`) are compared numerically based on their `ID`.
    - Providers (`Provider`) are compared numerically based on their tax identification number (`vat`). The `vat` can be represented as an integer value.
    - The products in the store (`StockableProduct`) are compared numerically according to the profit they bring to the store.
    - **NOTE:** Bear in mind that the `compareTo()` method of the `Comparable` interface receives as a parameter an `Object` type. Therefore, inside the method you will have to make an explicit casting to `Person`, `Provider` or `StockableProduct`. If that casting cannot be done, then a `ClassCastException` will be thrown. You do not need to program that exception as it already exists in the Java libraries, but you will have to catch it within the

[1] Review your `Order` class to ensure that you have an attribute to save `orderID`

[2] Review your `StockableProduct` class to make sure you have an attribute of type `Provider` to store the provider of each product. If you did not create this attribute in module-1 you will have to create it in section 4 of module-2 and then modify your `toString()` methods and read/write methods to take this attribute into account.

method itself and send a message indicating that the parameter of the `compareTo()` method should be a person (`Person`), a provider (`Provider`) or a product of the store (`StockableProduct`), respectively.

- Add a method `equals()` to the classes `Person`, `Provider` and `StockableProduct`. This method checks whether two persons, two providers, or two products are the same, respectively.
    - We consider that two people are the same if they have the same identifier (`id`).
    - We consider that two providers are the same if they have the same tax identification number (`vat`).
    - We consider that two products are the same if they have the same product identifier (`productID`).
    - **NOTE:** Any difference in the rest of the attributes will be considered a typographical error and will not be taken into account.

## 5. Modification of the product collections: `ProductList` and `Order`

**`ProductList` and `Order`. Methods to process information**

If you have not done it in module-1, add to the `ProductList` and `Order` classes the corresponding methods for information processing.

- Methods in class `ProductList`:
    - `public ProductList readFromFile(String file)`. It creates a `ProductList` object from the information stored in the file, with one product per line in the file.
    - `public void writeToFile(String file)`
    - `public void print()`
- Methods in class `Order`:
    - `public Order readFromFile(String file)`. It creates an `Order` object from the information stored in the file. The information of the `list` attribute is obtained from the content of the file (which has the same format as `ProductList`, with one `StockableProduct` per line). The information of the rest of the attributes is obtained from the name of the file, which has the following format `OrderID_CustomerID_EmployeeID.txt`.
    - `public void writeToFile()`. It writes in the file. It does not receive the file name because the file name is inferred from the attributes and must have the following format: `OrderID_CustomerID_EmployeeID.txt`.
    - `public void print()`
- **NOTE**: To program the file reading and writing methods you can proceed following the code provided in the class `Person`, but with the following modifications:
    - Instead of using `if((line=in.readLine())!= null)` which reads one line only, you can use `while((line=in.readLine())!= null)` to read all the lines in the file.
    - When you write in the file you only have to modify the `Order` class because besides writing the content of the file you have to build the name of the file from the attributes of the `Order` class as indicated above.

**`ProductList`. Methods for searching, inserting, deleting and modifying products.**

To implement these methods we can assume, for simplicity, that there are not two products with the same identifier in the list (we will check this when inserting items in the list).

- `public int indexOf(int productID)` searches the product whose identifier matches the one received as parameter and returns the position in the product is found. In case there is no product with that identifier it returns -1.
- `public StockableProduct search(int productID)` searches the product whose identifier matches the one received as parameter and returns it as the result. If there is no product with that identifier, it returns `null`.
- `public void insert(StockableProduct other)` adds the product received as parameter to the product list and updates the value of the attributes total cost, total price and total benefit. **NOTE**: If the product already exists instead of adding it as a new product, it increases the number of units of the existing product.

- `public StockableProduct remove(int productID, int numUnits)` removes the number of units of the indicated product from the list and updates the value of the attributes that store the total cost, total price and total benefit.
  - If the product does not exist, return `null`.
  - If after subtracting the number of units indicated as parameter the product is left with zero units, it is permanently removed from the list.
- `public void modify(int productID)` modifies the product whose identifier is received as parameter. To do this you will need to ask the user for the new values. **NOTE:** you can use the `readFromStdio()` method of the `StockableProduct` class to do this.

**Review the data structures provided: `LinkedList<E>`, `LinkedQueue<E>`, `LSBTree<E>`**

- You can download the data structures that you must learn to program during this course from Aula Global.
- In the exam we may ask you to program any of the methods of these data structures with small variations. We will be working with them during the lectures and laboratory sessions in this course.
- In the project you will not have to worry about how to program this data structures but about how to use them.

## 6. Class `StoreManager`

**Attributes**

Add to the store manager the following set of attributes, which will allow storing relevant information for its operation.

- A queue with orders (`LinkedQueue<E>`), which stores the orders to be processed (`ordersToProcess`).
- A linked list (`LinkedList<E>`), which stores the orders that have already been processed (`ordersProcessed`).
- A binary search tree (`LBSTree<E>`) of persons (`Person`), which stores the customers of the store (`storeCustomers`).
- A binary search tree (`LBSTree<E>`) of providers (`Providers`), which stores the providers of the store (`storeProviders`).
- A binary search tree (`LBSTree<E>`) of persons (`Person`), which stores the employees that process the orders in the store (`storeEmployees`).
  - NOTE: Do not worry about the implementation of these data structures, we will see them in lectures and laboratory sessions. Just check their method signatures to insert, extract or search elements, so that you can use these methods in case you need it.
- An array `Strings[] storeDataInfo`, which saves the name of the store and the path (relative to the project) of the files and/or folders where the information is stored, taking into account the following positions in the array: 0-name of the store, 1-stock.txt, 2-ordersToProcess, 3-ordersProcessed, 4-storeCustomers.txt, 5-storeProviders.txt, 6-storeEmployees.txt.

**Set/get methods and constructors**

- Create the `get/set` methods to access and change the attributes you have just created.
- Create an additional `set` method for each of the data structures. This set method receives the name of a file or folder as a parameter. Each of these `set` methods has two main functions: (1) saves the name it receives as parameter in the array `storeDataInfo` and (2) (if it receives a file or folder as parameter) reads it and stores its information in the appropriate data structure.
  - Example-1: the `setOrdersToProcess(String folder)` method receives as parameter the *path* where the orders to process are, (1) stores the path in `storeDataInfo[2]` and (2) reads all the files of this folder and stores their content in the queue called `ordersToProcess`.
  - Example-2: the `setStoreCustomers(String file)` method receives as parameter the *path* of the file where the clients of the store are, (1) stores this path in `storeDataInfo[4]` and (2) reads the content of the file and stores it in the tree `storeDataCustomers` ordered by its `ID`.
- `public void set(String[] storeData)`. Create a `set` method that receives as parameter an array of strings which contains the name of the store and the name of the files/folders with all its

information. This method will call the above mentioned methods to save the *path* to the files that contain the data and load their content in the different data structuresof the store.

- `public StoreManager(String name, String stock, String ordersToProcess, String ordersProcessed, String storeCustomers, String storeProviders, String storeEmployees)`: Create a constructor that receives as parameter the name of the store (`name`) and 6 Strings indicating the name of the files (`stock`, `storeCustomers`, `storeProviders`, `storeEmployees`) and folders (`ordersToProcess`, `ordersProcessed`) that contain the rest of the necessary information to create a store manager. The constructor must: (1) call the corresponding `set` method to assign each parameter and (2) calculate the attributes that it does not receive as parameter such as the cost of the products it has in stock (`stockCost`) and the benefit that would be obtained its sale (`stockBenefit`). **NOTE**: Bear in mind that to calculate the benefit from the sale of the stock you will need to know not only the cost of the products but also their price.

**Methods `toString()`,`print()` and `readFromStdio()`**

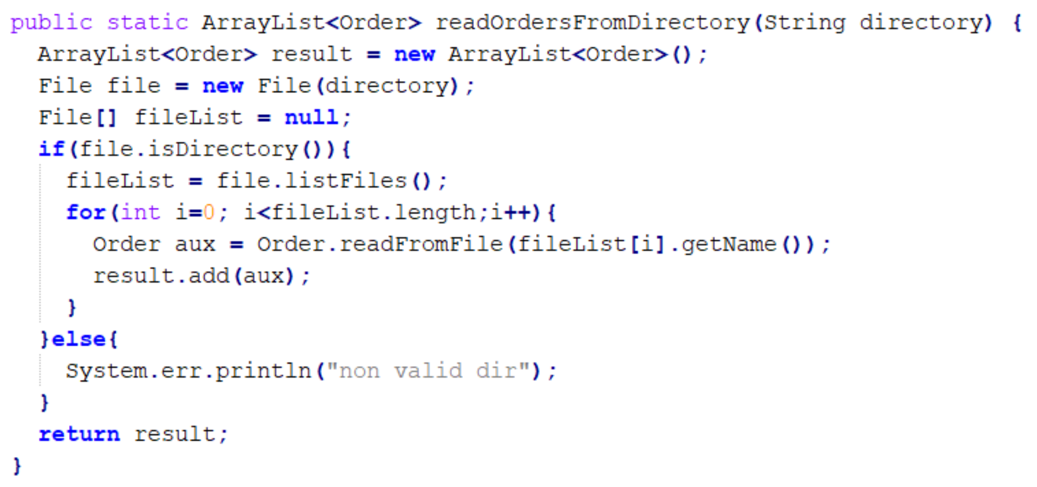
- `public String toString()` which prints the name of the store and the name of the files or folders that store the rest of the relevant information: `name|stock|ordersToProcess|ordersProcessed|storeCustomers|storeProviders|storeEmployees`
- `public void print()` which prints the information of the store in three lines: (1) the first line contains the name of the store and its relevant files/folders in the format indicated by the `toString()` method, (2) the second line indicates the cost of the stock, and (3) the third line indicates the benefit obtained after its sale. **NOTE:** Remember that this information appears when the main menu is printed.
- `public StoreManager readFromStdio()` which serves to read the information from the store from the standard input. To do this, you must ask the user for the name of the store and the name of the files and folders that contain the information to build it. You must request this information in the same order as indicated in the constructor.

**Auxiliary methods for processing information from the files of the store**

- `public ArrayList<Person> readPersonsFromFile(String file)`. To see the format of the file you can check the files *storeCustomers.txt* and *storeEmployees.txt* as example).
- `public ArrayList<Provider> readProvidersFromFile (String file).`
- Note (Reading methods):
    - These methods (1) create an `ArrayList<E>` of objects of type `Person` or `Provider`, (2) fill it from the information contained in the file they receive as parameter and (3) return it as result. The file they receive as parameter contains the information of one object per line.
    - To read several objects from a file you can follow the same strategy that we indicated in `ProductList`: when you read the file instead of using `if((line=in.readLine())!= null)` which only reads one line (as we did in module-1) you can use `while((line=in.readLine())!= null)` to read all the lines in the file.

**Auxiliary method for loading information from the order folders**

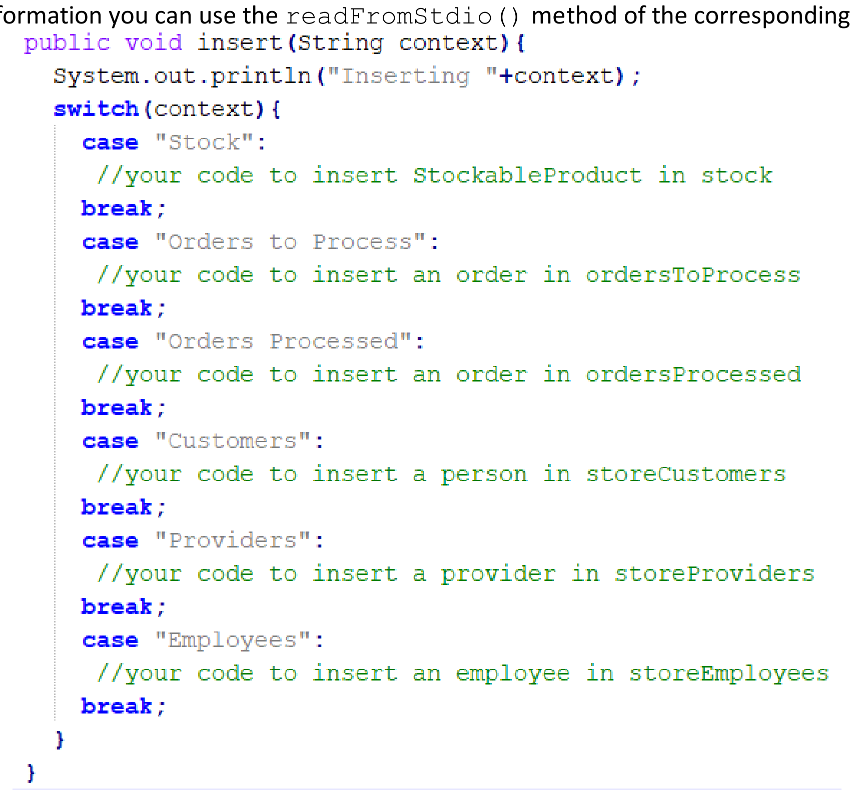
- `public static ArrayList<Order> readOrdersFromFolder(String folder)` is a method that reads a set of orders from the folder it receives as parameter. For the method to work, the files contained in the folder must have the appropriate format both in name (*orderID_customerID_employeeID*) and in content (one order per line in the format indicated in the `Order` class). You can review the example files. Below you can see a reference implementation. You can add a `try/catch` statement to this reference implementation in case your `readFromFile(String file)` method of the class `Order` throws an exception.



```java
public static ArrayList<Order> readOrdersFromDirectory(String directory) {
  ArrayList<Order> result = new ArrayList<Order>();
  File file = new File(directory);
  File[] fileList = null;
  if(file.isDirectory()){
    fileList = file.listFiles();
    for(int i=0; i<fileList.length;i++){
      Order aux = Order.readFromFile(fileList[i].getName());
      result.add(aux);
    }
  }else{
    System.err.println("non valid dir");
  }
  return result;
}
```

**Methods to operate with the data of the store (search, insert, delete and modify elements)**
Our store has several data structures to store the information. On any of these data structures we can apply the operations of search, insertion, extraction (removal) or modification. To identify the data structure we are referring to we will use a variable called `context` that can take several values: “*Stock*”, “*Orders to Process*”, “*Orders Processed*”, “Customers”, “*Providers*” or “*Employees*”. This way we only have to implement one method for each operation and we will know on which data structure to apply this method depending on the context that the method receives as a parameter. The code in the following figure can serve as a template for the insert method that performs the insert operation. You can use similar structures for the other operations.

- `public void insert(String context)`: inserts elements in the different data structures of the store. Depending on the context, the element to be inserted, and the structure in which it is inserted may be different (you can see the details in the code comments in the figure). To insert an element in a context, create an empty object of the type of the element to be inserted and then ask the user for the information to fill it in before inserting it in the corresponding data structure. **NOTE**: To request the user information you can use the `readFromStdio()` method of the corresponding object.



```java
public void insert(String context){
  System.out.println("Inserting "+context);
  switch(context){
    case "Stock":
     //your code to insert StockableProduct in stock
    break;
    case "Orders to Process":
     //your code to insert an order in ordersToProcess
    break;
    case "Orders Processed":
     //your code to insert an order in ordersProcessed
    break;
    case "Customers":
     //your code to insert a person in storeCustomers
    break;
    case "Providers":
     //your code to insert a provider in storeProviders
    break;
    case "Employees":
     //your code to insert an employee in storeEmployees
    break;
  }
}
```

- `public Object search(String context)`: searches elements in the structure indicated by the `context`. You can use a structure similar to the insert method taking into account the following considerations:

    - To search an element, you have to ask the user for the unique identifier of the element to be searched. The unique identifier depends on the context, for example, `ID` in the case of objects of type `Person`, `productID` in the case of products of the store and `orderID` in the case of orders.
    - If there no element matches the identifier, then this method returns `null`.
- `public Object remove(String context)`: removes (extracts) elements from the structure indicated by the `context`. You can use a structure similar to the insert method taking into account the following considerations:
    - To remove an element, you have to ask the user for the unique identifier of the element to be removed. If no element matches the identifier, the method returns `null`.
    - In case you are removing a `StockableProduct` you must also ask the user for the number of product units to be deleted.
    - The type of object returned by this method is different depending on the context. For example, it returns a `StockableProduct` if you are removing a product from stock; it returns an `Order` if you are removing orders from `ordersToProcess` or `ordersProcessed`.
    - **NOTE**: Deleting customers, providers or employees from the store is not allowed. Therefore, an error message will be printed in the last three cases: `System.err.println(context+": removing [Customers/Providers/Employees] from Store is not allowed" );` and `null` will be returned.
- `public void modify(String context)`. modifies elements in the structure indicated by the `context`. You can use a structure similar to the insert method taking into account the following considerations:
    - To modify an item, you will have to ask the user for the identifier of the item to be modified and the information you need to modify it.
    - NOTE: Modifying customers, providers or employees that already exist in the store is not allowed. Therefore, an error message will be printed in the last three cases: `System.err.println(context+": modifying [Customers/Providers/Employees] from Store is not allowed" );`

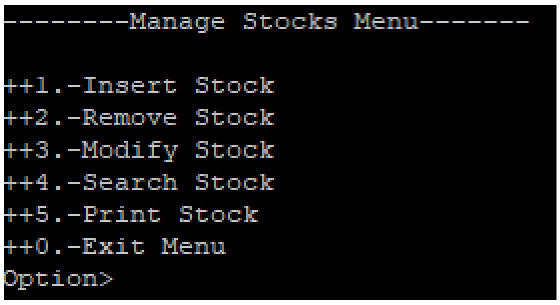
**Method for processing orders**

- `public void processOrders()`. This method processes all pending orders stored in `ordersToProcess`, does the necessary operations to process the orders and finally store them as `ordersProcessed`.
- For each `Order` this method must go through each of the products in the order:
    - Checking if the product is in stock and if there are enough units of it to serve the order.
    - Checking if the provider is among the providers of the store and if the employee is in the list of employees.
    - If any of the above mentioned conditions is not met, a general exception must be thrown (`Exception`). If there is no problem and sufficient units are available:
        - The requested product units are subtracted from the stock of the store. Then, if the stock of the product is zero, the product must be removed.
        - The customer is searched in the binary search tree of customers if it does not exist, then is is added,
        - The values of the cost of the store and its benefit are updated.

## 7. Menu Options

In the second level context menu (submenu), add the option *5. - Print*, which makes use of the `print()` methods you have created, so you can see the effect of the other menu options on the different structures of the store. It is important to note that when each menu option is selected, the requested operation must be executed.

The call to the `public void processOrders()` method of the `StoreManager` class, which processes all the pending orders stored in *ordersToProcess*, is made in option 3 of the menu (*manage orders to process*) including an option in the submenu that is "*Process pending orders*" to call the mentioned method.



```
--------Manage Stocks Menu-------

++1.-Insert Stock
++2.-Remove Stock
++3.-Modify Stock
++4.-Search Stock
++5.-Print Stock
++0.-Exit Menu
Option>
```