In the class named ProductInfo, there are important parameters such as name and id of the product.

*public class* ProductInfo *implements* Comparable<ProductInfo>,  
 Cloneable {

It is seen that some field values are Bigdecimal to use the compareto method.

*private* BigDecimal m\_cost = BigDecimal.**ZERO**;  
*private* BigDecimal m\_price = BigDecimal.**ZERO**;

***@Override****public int* **compareTo**(ProductInfo other)  
{  
 *return* m\_price.**compareTo**(other.m\_price);  
}

With cloneable we return different object references and distinguish them with equals and hashcode! so the data in line 1 is the same as number 1 in the other line. We made different copies in the scenario and provided this with equals and hash codes!

***@Override****public* Object **clone**()  
{  
 *return new* **ProductInfo**()  
 .**setId**(m\_id)  
 .**setName**(m\_name)  
 .**setStock**(m\_stock)  
 .**setCost**(m\_cost)  
 .**setPrice**(m\_price);  
}

***@Override****public int* **hashCode**()  
{  
 *return* m\_id;  
}  
  
***@Override****public boolean* **equals**(Object obj)  
{  
 *if* (!(obj *instanceof* ProductInfo))  
 *return false*;  
  
 *var* other = (ProductInfo)obj;  
  
 *return* other.m\_id == m\_id;  
}

products.**stream**().***distinct***().***forEach***((val)->strings.***add***(val.**toString**()));

ProductFactory class reads csv files line by line and assigns the data in each line to various elements of the ProductInfo class. For this, we first created a ProductInfo list as a constant. Also, we chose this way to use a class that did not derive from Collection with StreamSupport.

*public final* List<ProductInfo> PRODUCTS = *new* **ArrayList**<>();

*public* Iterable<ProductInfo> **getProductsAsIterable**()  
{  
 *return* PRODUCTS;  
}

With ProductFactory, we have assigned the information in a line in a csv file to an object's variables!

*private static* String **join**(String [] strings, *int* startIndex, *int* endIndex, String delimiter)  
{  
 StringBuilder sb = *new* **StringBuilder**();  
 *for* (*int* i = startIndex; i < endIndex; ++i)  
 sb.**append**(strings[i]).**append**(delimiter);  
  
 *return* sb.**substring**(0, sb.**length**() - delimiter.**length**());  
}

*private static* ProductInfo **getProduct**(String line)  
{  
 *var* productsInfo = line.**split**("[,]");*//regex  
 var* id = Integer.parseInt(productsInfo[0]);  
 *var* name = join(productsInfo, 1, productsInfo.length - 3, ",");  
 *var* stock = Integer.parseInt(productsInfo[productsInfo.length - 3]);  
 *var* cost = *new* **BigDecimal**(productsInfo[productsInfo.length - 2]);  
 *var* price = *new* **BigDecimal**(productsInfo[productsInfo.length - 1]);  
  
 *return new* **ProductInfo**().**setId**(id).**setName**(name).**setPrice**(price).**setCost**(cost).**setStock**(stock);  
}

Since classes can perform various operations through methods, the effect of wrapper classes has been shown for the representation of basic types in structures such as Collection, their use in streams, and the withdrawal of basic types in type conversions.

*var* productsInfo = line.**split**("[,]");*//regex  
var* id = Integer.parseInt(productsInfo[0]);

With wrapper classes, a base type value can also be logically held by an Object reference. This process

can be boxed. In the same way, value can be obtained by opening boxes. So the general purpose is of the basic is to keep the values in "heap"

The enumeration is a public static final class that acts as a kind of labeling, so they behave singular. The human object has a marital status state.

*public enum* MaritalStatus {  
 **SINGLE**, **MARRIED**, **DIVORCED**, **WIDOW**}

*private static final* MaritalStatus[] **ms\_status** = MaritalStatus.values();  
*private static final* Random **m\_random** = *new* **Random**();  
*public final* List<Person> PEOPLE = *new* **ArrayList**<>();

A Person object can be created with the Person Factory method. Various stream operations can be performed, such as grouping.

*private static* Person **getPerson**(String line)  
{  
 *var* peopleInfo = line.**split**("[,]");  
  
 *return new* **Person**()  
 .**setId**(Integer.parseInt(peopleInfo[0]))  
 .**setName**(peopleInfo[1])  
 .**setGender**(peopleInfo[2])  
 .**setBirthDate**(peopleInfo[3])  
 .**setMaritalStatus**(**ms\_status**[**m\_random**.**nextInt**(**ms\_status**.length)]);  
}

*var* factory = PersonFactory.loadFromTextFile(Path.of("people.csv"));  
*var* people = factory.PEOPLE;  
  
Map<MaritalStatus, List<Person>> map = people.**stream**()  
 .***collect***(Collectors.groupingBy(Person::**getMaritalStatus**));  
  
*if* (map.***containsKey***(MaritalStatus.**SINGLE**)) {  
 Console.writeLine("single ones:");  
 map.***get***(MaritalStatus.**SINGLE**).**forEach**(Console::writeLine);  
}

Within the project, there are functional interfaces, optional classes and third party libraries written by the csd association.

Jar file belonging to Kaplan-socket class is added with dependency and used in a multi-client client server architecture.

*private void* **sendStream**(*int* count, *int* length, BufferedWriter bw) *throws* IOException {  
  
  
*// var text = StringUtil.getRandomTextEN(random, length);  
 var* text = Sender.distinctProducts("products.csv",count+length);  
  
  
 *//Console.writeLine("%s ", text);* bw.**write**(text + "\r\n");  
 bw.**flush**();  
  
 }

A simple security test is performed on the client that wants to receive service from the server, and then it is expected to process within 20 seconds. The socket is closed for clients that exceed this period.

In this part, it is aimed that the server client architecture should work with callback and custom libraries as much as possible.

subscribeRunnable(() -> **generateStreamCallback**(clientSocket), clientSocket,  
 ex -> Console.Error.writeLine("generatePasswords:%s", ex.**getMessage**()), () -> m\_clients.***remove***(clientSocket));

one of them is the subscribeRunnable method of the exceptionutil class. With this method of the exceptionutil class, subscriblerunnable, the desired job is received with an action callback.

The structure targeted here is an action callback, the socket method parameter to be closed, how the error will be handled with the Consumer, and finally the socket will be deleted with the runnable.

subscribeRunnable(() -> **generateStreamCallback**(clientSocket), clientSocket,  
 ex -> Console.Error.writeLine("generatePasswords:%s", ex.**getMessage**()), () -> m\_clients.***remove***(clientSocket));  
}

***@FunctionalInterface****public interface* IActionCallback {  
 *void* **run**() *throws* Exception;  
}

**generateStreamCallback**(Socket clientSocket) *throws* IOException

If the run method in IActionCallback didn't throw Exception, generateStreamcallback had to be put in try catctch block !!!