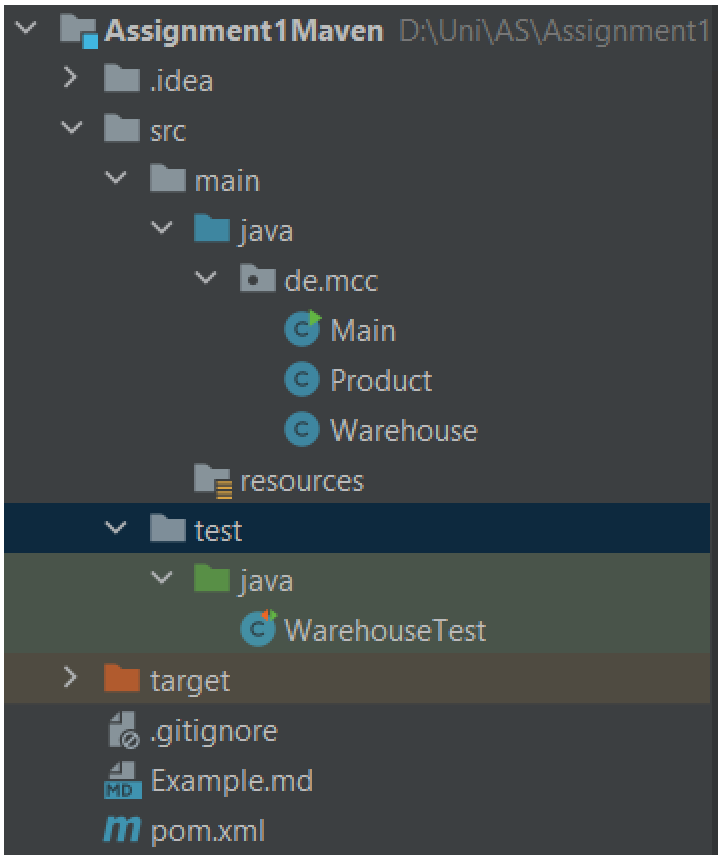
Anwendungssysteme Übungsaufgabe 1

Dimitar Dimov



```
Assignment1Maven D:\Uni\AS\Assignment1
  .idea
  src
    main
      java
        de.mcc
          Main
          Product
          Warehouse
      resources
    test
      java
        WarehouseTest
  target
  .gitignore
  Example.md
  pom.xml
```

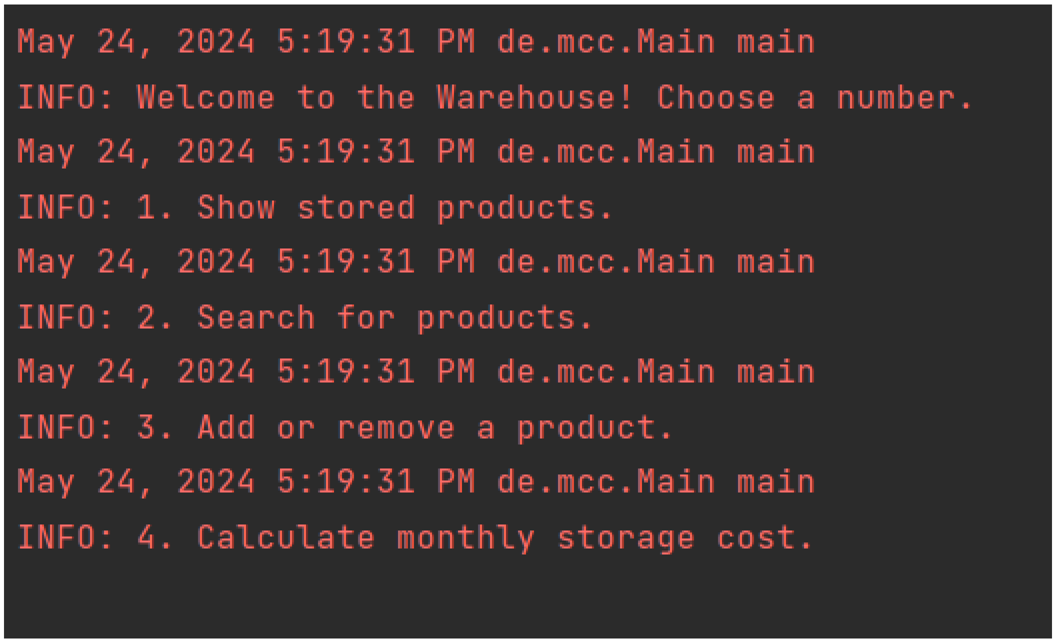
I tried keeping my structure as simple as possible. I have only 3 classes for the logic and 1 class for the tests.

The Main class is used to display the console and uses a switch case with the available options the user has to choose from.

The Product class is used to contain all the attributes needed to create a Product object, i.e. no logic in this class.

The Warehouse class does all the heavy lifting and included every single method that was listed as a requirement. It also initializes the ArrayList<Product> that has all the products in the warehouse.

The WarehouseTest class tests methods mainly from the Warehouse class (more on that in the coverage).

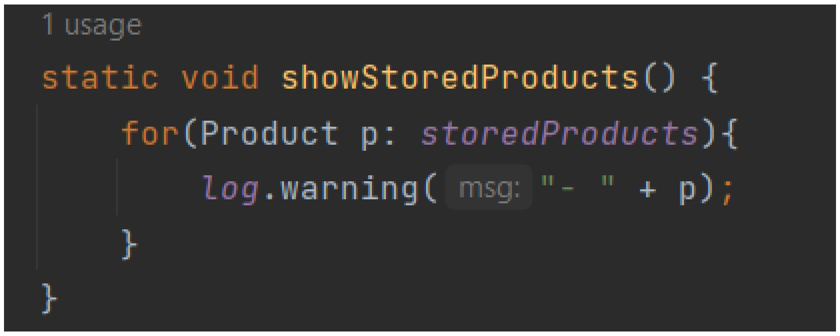


```
May 24, 2024 5:19:31 PM de.mcc.Main main
INFO: Welcome to the Warehouse! Choose a number.
May 24, 2024 5:19:31 PM de.mcc.Main main
INFO: 1. Show stored products.
May 24, 2024 5:19:31 PM de.mcc.Main main
INFO: 2. Search for products.
May 24, 2024 5:19:31 PM de.mcc.Main main
INFO: 3. Add or remove a product.
May 24, 2024 5:19:31 PM de.mcc.Main main
INFO: 4. Calculate monthly storage cost.
```

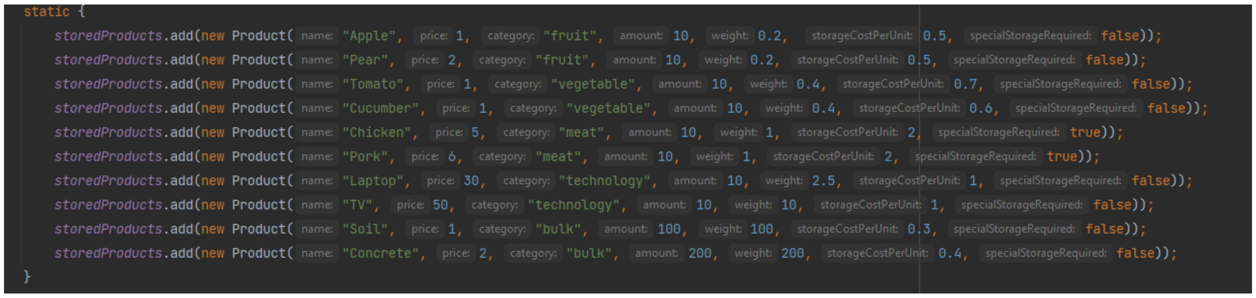
The console uses "WARNING" as the log level and the user has to pick a number from 1 to 4. If the user enters anything else, a default case runs and "Invalid option" is displayed.

1. DISPLAY ALL STORED PRODUCTS

This was done with a for-loop that goes through the "storedProducts" ArrayList. The list of products was put in a static block for easier access.



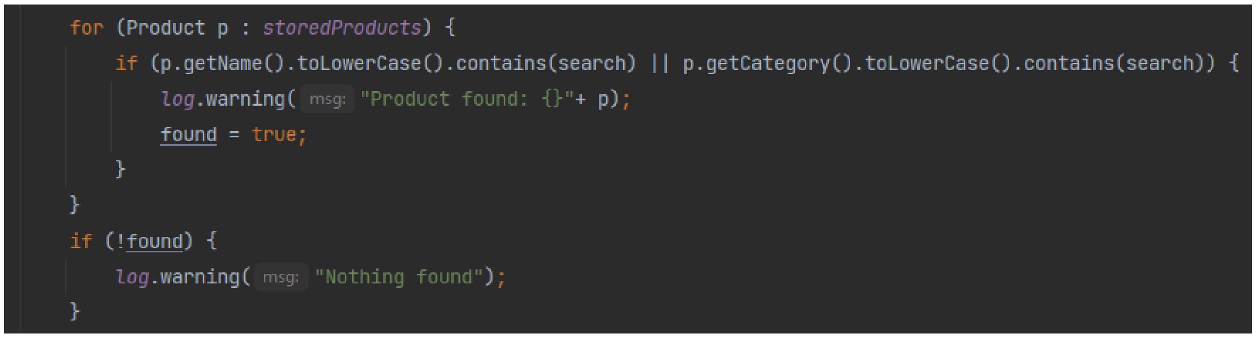
```java
1 usage
static void showStoredProducts() {
    for(Product p: storedProducts){
        log.warning( msg: "- " + p);
    }
}
```



```java
static {
    storedProducts.add(new Product( name: "Apple",  price: 1,  category: "fruit",  amount: 10,  weight: 0.2,   storageCostPerUnit: 0.5,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Pear",  price: 2,  category: "fruit",  amount: 10,  weight: 0.2,  storageCostPerUnit: 0.5,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Tomato",  price: 1,  category: "vegetable",  amount: 10,  weight: 0.4,  storageCostPerUnit: 0.7,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Cucumber",  price: 1,  category: "vegetable",  amount: 10,  weight: 0.4,  storageCostPerUnit: 0.6,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Chicken",  price: 5,  category: "meat",  amount: 10,  weight: 1,  storageCostPerUnit: 2,  specialStorageRequired: true));
    storedProducts.add(new Product( name: "Pork",  price: 6,  category: "meat",  amount: 10,  weight: 1,  storageCostPerUnit: 2,  specialStorageRequired: true));
    storedProducts.add(new Product( name: "Laptop",  price: 30,  category: "technology",  amount: 10,  weight: 2.5,  storageCostPerUnit: 1,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "TV",  price: 50,  category: "technology",  amount: 10,  weight: 10,  storageCostPerUnit: 1,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Soil",  price: 1,  category: "bulk",  amount: 100,  weight: 100,  storageCostPerUnit: 0.3,  specialStorageRequired: false));
    storedProducts.add(new Product( name: "Concrete",  price: 2,  category: "bulk",  amount: 200,  weight: 200,  storageCostPerUnit: 0.4,  specialStorageRequired: false));
}
```

2. SEARCH THROUGH PRODUCTS AND PRODUCT CATEGORIES

The method initializes a scanner and prompts the user to write a string that isn't case sensitive. Then a for-loop and an if-statement check if anything was found in the storedProducts list.

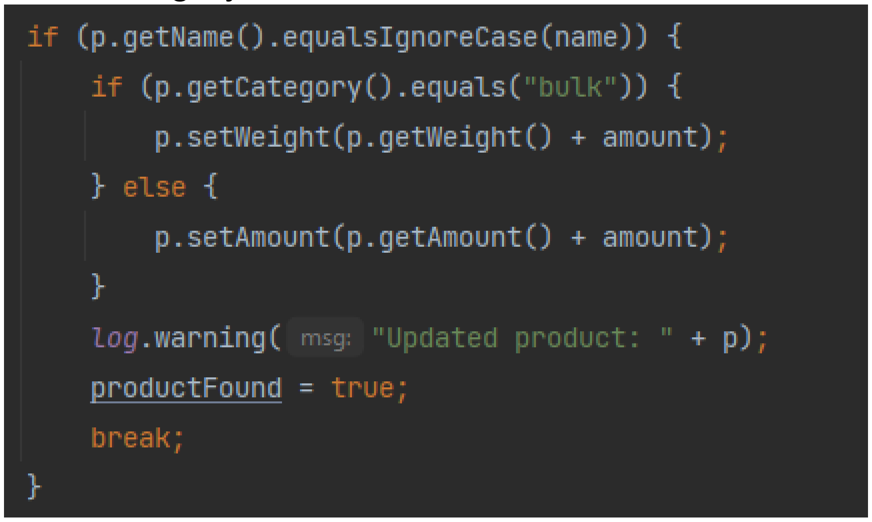


```
for (Product p : storedProducts) {
    if (p.getName().toLowerCase().contains(search) || p.getCategory().toLowerCase().contains(search)) {
        log.warning( msg: "Product found: {}"+ p);
        found = true;
    }
}
if (!found) {
    log.warning( msg: "Nothing found");
}
```

3. ADD PRODUCTS AND CHANGE THE AMOUNT IN THE SYSTEM

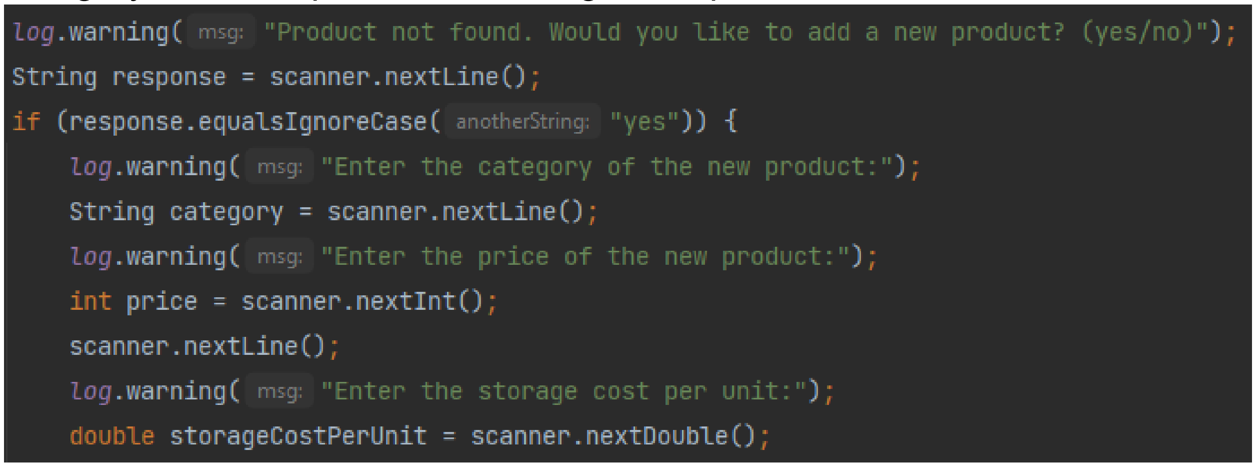
This method was changed constantly and turned out to be very long. It asks the user to enter the name, then the amount, which could be positive or negative so that you can add or remove products, creates local boolean called "productFound", and the goes through a for-loop.

- In the category of "bulk", we set the weight of a product and in every other category we set the amount



```
if (p.getName().equalsIgnoreCase(name)) {
    if (p.getCategory().equals("bulk")) {
        p.setWeight(p.getWeight() + amount);
    } else {
        p.setAmount(p.getAmount() + amount);
    }
    log.warning( msg: "Updated product: " + p);
    productFound = true;
    break;
}
```

- If a product isn't found, the user is asked if he/she wants to add a new type of product. If "yes" (another case insensitive String), a series of question regarding the product start to appear. Since the user already wrote the name of the product, the first question is the category, then the price, then storage cost per unit etc.

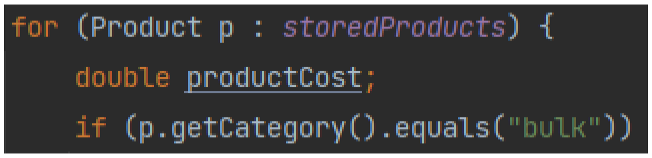


```
log.warning( msg: "Product not found. Would you like to add a new product? (yes/no)");
String response = scanner.nextLine();
if (response.equalsIgnoreCase( anotherString: "yes")) {
    log.warning( msg: "Enter the category of the new product:");
    String category = scanner.nextLine();
    log.warning( msg: "Enter the price of the new product:");
    int price = scanner.nextInt();
    scanner.nextLine();
    log.warning( msg: "Enter the storage cost per unit:");
    double storageCostPerUnit = scanner.nextDouble();
```

  This method turned out a bit long, because after adding products that can be sold by weight instead of amount I had to update it, but it is fairly simple. It only uses a set of if-statements and scanner methods, and finally it the console prints out either "New product added" or "Operation canceled".

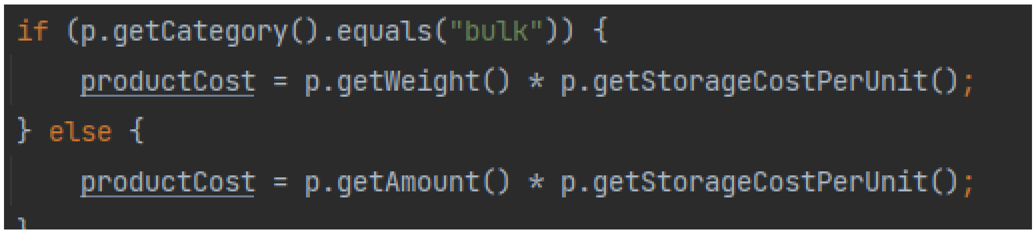
4. CALCULATE THE COST TO STORE ALL PRODUCTS

First, we define a double called "totalCost" for the total amount and initialize it to be 0.0. Afterwards, we sort through the "storedProducts" again and check if the category of products is "bulk"



```
for (Product p : storedProducts) {
    double productCost;
    if (p.getCategory().equals("bulk"))
```

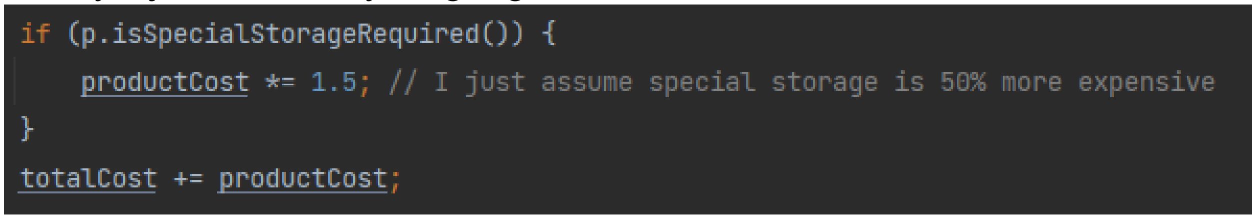
If that is the case, we multiply the weight with the storage cost per unit. If not, we multiple the amount with the storage cost per unit



```
if (p.getCategory().equals("bulk")) {
    productCost = p.getWeight() * p.getStorageCostPerUnit();
} else {
    productCost = p.getAmount() * p.getStorageCostPerUnit();
```

If the product is listed as needing special storage in the “storedProducts” ArrayList, I multiply the product cost by 1.5.

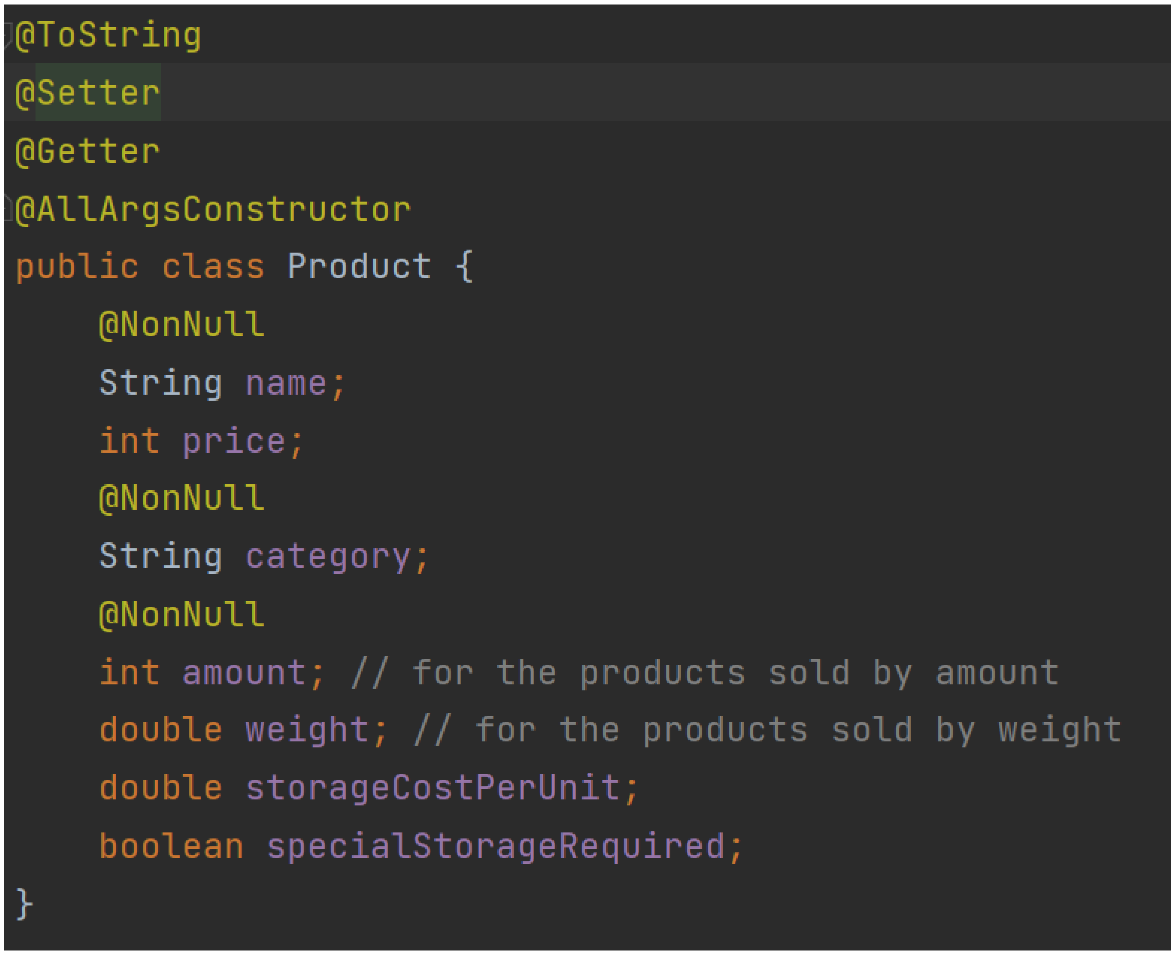
Finally, I just add everything together



```java
if (p.isSpecialStorageRequired()) {
    productCost *= 1.5; // I just assume special storage is 50% more expensive
}
totalCost += productCost;
```

5. WEIGHT AND AMOUNT OF PRODUCTS

This was just added in the Product class, which uses Lombok logging for setters, getters, a custom toString and the constructor



```java
@ToString
@Setter
@Getter
@AllArgsConstructor
public class Product {
    @NonNull
    String name;
    int price;
    @NonNull
    String category;
    @NonNull
    int amount; // for the products sold by amount
    double weight; // for the products sold by weight
    double storageCostPerUnit;
    boolean specialStorageRequired;
}
```