Lab 5.1

1. Objectives

XML · Packages · Utilities · Javadoc · ShopV5.0 ·

2. Developing Shop V5.0

In IntelliJ, create a new project called ShopV5.0 and copy these files into it:

- Driver.java
- Store.java
- Product.java
- ScannerInput.java

We will add persistence to our app via two new items to the menu, one for Save and one for Load.
We will use an external component called XStream for our serialisation to XML.

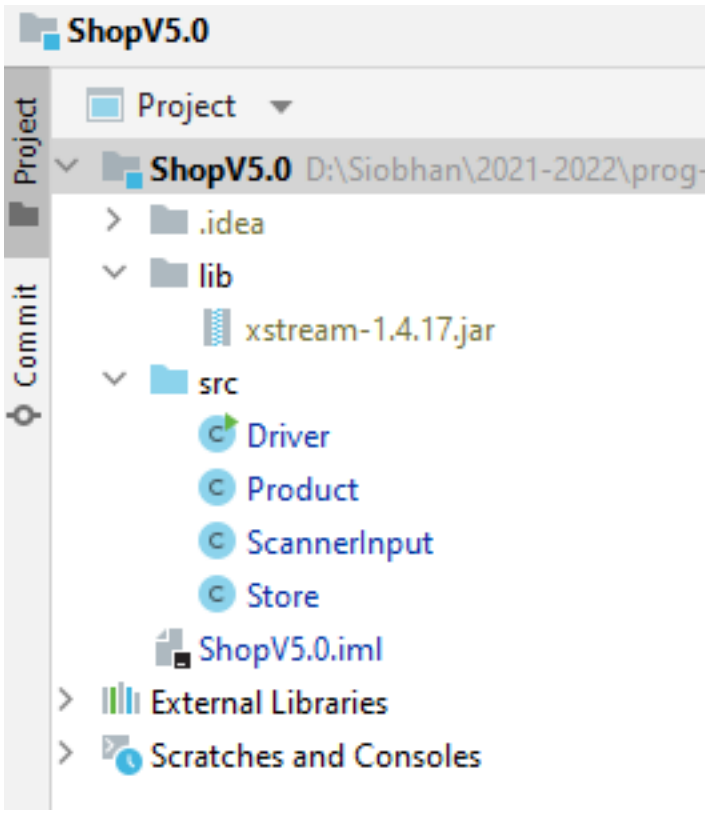
Setting up the Component for Serializing

Download the XStream, version 1.4.20 **jar** file from this website:

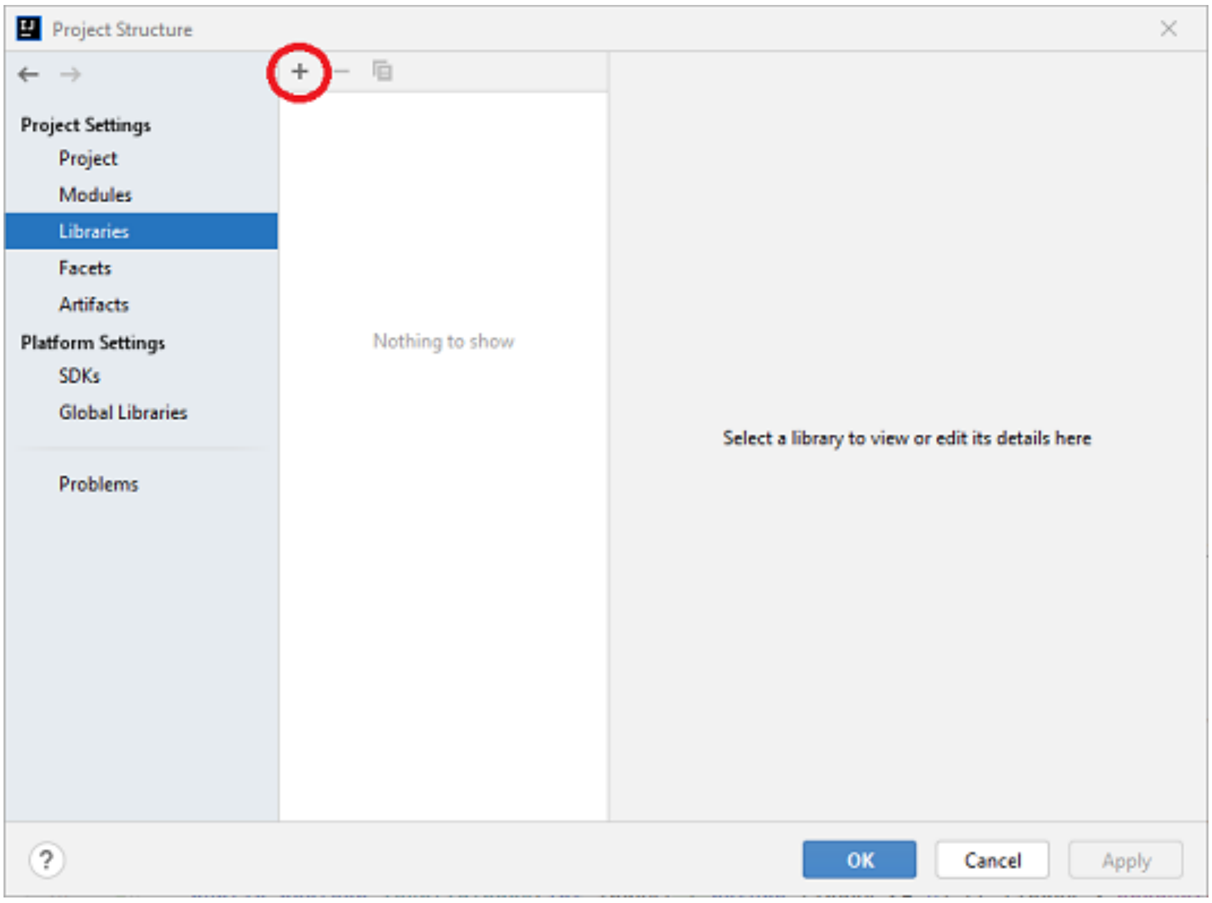
https://mvnrepository.com/artifact/com.thoughtworks.xstream/xstream/1.4.20

In IntelliJ, right click on the ShopV5.0 project folder and select **New**, followed by **Directory**. Call the new directory **lib**. Drag the xstream jar file already downloaded into the lib folder.

Your workspace should look like this:



From **File** menu, select **Project Structure**. Click on **Libraries**. To add a library to your build path, click on the + :



Select Java and locate the XStream compenent in your lib folder…click OK (a few times!).

Updating Store Class to serialize and deserialize products

Now that we have brought the XStream component into our project and added it to the build path (so the compiler finds it), we can now start using it.

In the Store class, create these two new methods that will use the XStream component to serialise to and from products.xml which is located in the root directory of your project:

```
    @SuppressWarnings("unchecked")
      public void load() throws Exception {
          //list of classes that you wish to include in the
serialisation, separated by a comma
          Class<?>[] classes = new Class[] { Product.class };

          //setting up the xstream object with default security and
the above classes
          XStream xstream = new XStream(new DomDriver());
          XStream.setupDefaultSecurity(xstream);
          xstream.allowTypes(classes);

          //doing the actual serialisation to an XML file
          ObjectInputStream is = xstream.createObjectInputStream(new
FileReader("products.xml"));
          products = (ArrayList<Product>) is.readObject();
          is.close();
      }

      public void save() throws Exception {
          XStream xstream = new XStream(new DomDriver());
          ObjectOutputStream out =
xstream.createObjectOutputStream(new FileWriter("products.xml"));
          out.writeObject(products);
          out.close();
      }
```

If the code doesn’t compile, you may need to import these packages:

```
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

Adding load and save functionality to the menu

In the Driver class, add options 9 and 10 to the menu. Note that we have added some nice formatting to the menu as visually it is getting quite large:

```
    private int mainMenu() {
        return ScannerInput.readNextInt("""
-----------------------------------------------------------------
                |              Shop Menu                 |
-----------------------------------------------------------------
                |   1) Add a product                     |
                |   2) List the Products                 |
                |   3) Update a product                  |
                |   4) Delete a product                  |
-----------------------------------------------------------------
                |   5) List the current products         |
                |   6) Display average product unit cost |
                |   7) Display cheapest product          |
                |   8) List products that are more expensive than
a given price  |
-----------------------------------------------------------------
                |   9)  Save products to products.xml    |
                |   10) Load products from products.xml  |
                |   0)  Exit                             |
                --------------------------------------------------
----------------
                ==>>  """);
    }
```

In the Driver class, update the switch statement to call two new methods for save and load:

```
        switch (option) {
                case 1 -> addProduct();
                case 2 -> printProducts();
                case 3 -> updateProduct();
                case 4 -> deleteProduct();
                case 5 -> printCurrentProducts();
                case 6 -> printAverageProductPrice();
                case 7 -> printCheapestProduct();
                case 8 -> printProductsAboveAPrice();
                case 9 -> saveProducts();
                case 10 -> loadProducts();
                default -> System.out.println("Invalid option
entered: " + option);
            }
```

and the code for saveProducts() is here:

```
    private void saveProducts() {
        try {
            store.save();
        } catch (Exception e) {
            System.err.println("Error writing to file: " + e);
        }
    }
```

and the code for loadProducts() is here:

```
    private void loadProducts() {
        try {
            store.load();
        } catch (Exception e) {
            System.err.println("Error reading from file: " + e);
        }
```

Testing the load and store

You should be in a position now to test.

Start your app and create two products e.g.

```
Please enter the product description: 24 inch monitor
Please enter the product code: 3423
Please enter the product cost: 129.99
Is this product in your current line (y/n): y

Please enter the product description: 14 inch monitor
Please enter the product code: 2322
Please enter the product cost: 109.99
Is this product in your current line (y/n): y
```

Now try menu option 9 to save your products.

You should now see a products.xml file appearing in the root folder of your project in IntelliJ.

Open this file and it should contain something similar to this:

```
<object-stream>
  <list>
    <Product>
      <productName>24 inch monitor</productName>
      <productCode>3423</productCode>
      <unitCost>129.99</unitCost>
      <inCurrentProductLine>true</inCurrentProductLine>
    </Product>
    <Product>
      <productName>14 inch monitor</productName>
      <productCode>2322</productCode>
      <unitCost>109.99</unitCost>
      <inCurrentProductLine>true</inCurrentProductLine>
    </Product>
  </list>
</object-stream>
```

Exit your application (menu option 0) and run it again.

Test option 10 and make sure that the saved products are loaded back into your products ArrayList correctly.

3. Shop V5.0 – Utilities

In this step, you will continue working on Shop V5.0.

ScannerInput

We are currently using the **ScannerInput** class to manage our user I/O. If you recall, we created this class to get past the bug that was in Scanner i.e. a read line is ignored after reading an int, float, double. In our ScannerInput class, we are using the try-catch construct to continually ask the user for correct input when asking for an int and a double.

Open this class now and familiarise youself with the code.

This type of class is called a **Utility**. Did you notice that we cut and paste the code into our SocialNetwork project too? We can reuse this utility class in any of our console projects.

Other Utilities

We can write and reuse other types of utilities too. It is a really good idea to build a repository of these utilities for project work.

For example, did you notice that the average product price typically has a lot more than two digits after the decimal point. Run your application and having entered a few new products, try option 6 to display the average price. There are a lot more than two decimal places.

We want to truncate this output to two decimal places. Let’s start by creating a new class called **Utilities** in your **src** folder.

In this class, write this utility method that will truncate (not round) the amount to two decimal places:

```
public class Utilities {
      public static double toTwoDecimalPlaces(double number){
          return (int) (number * 100 ) / 100.0;
      }

  }
```

Remember your order of evaluation…brackets are done first, then division. Try to figure out what's happening in this method.

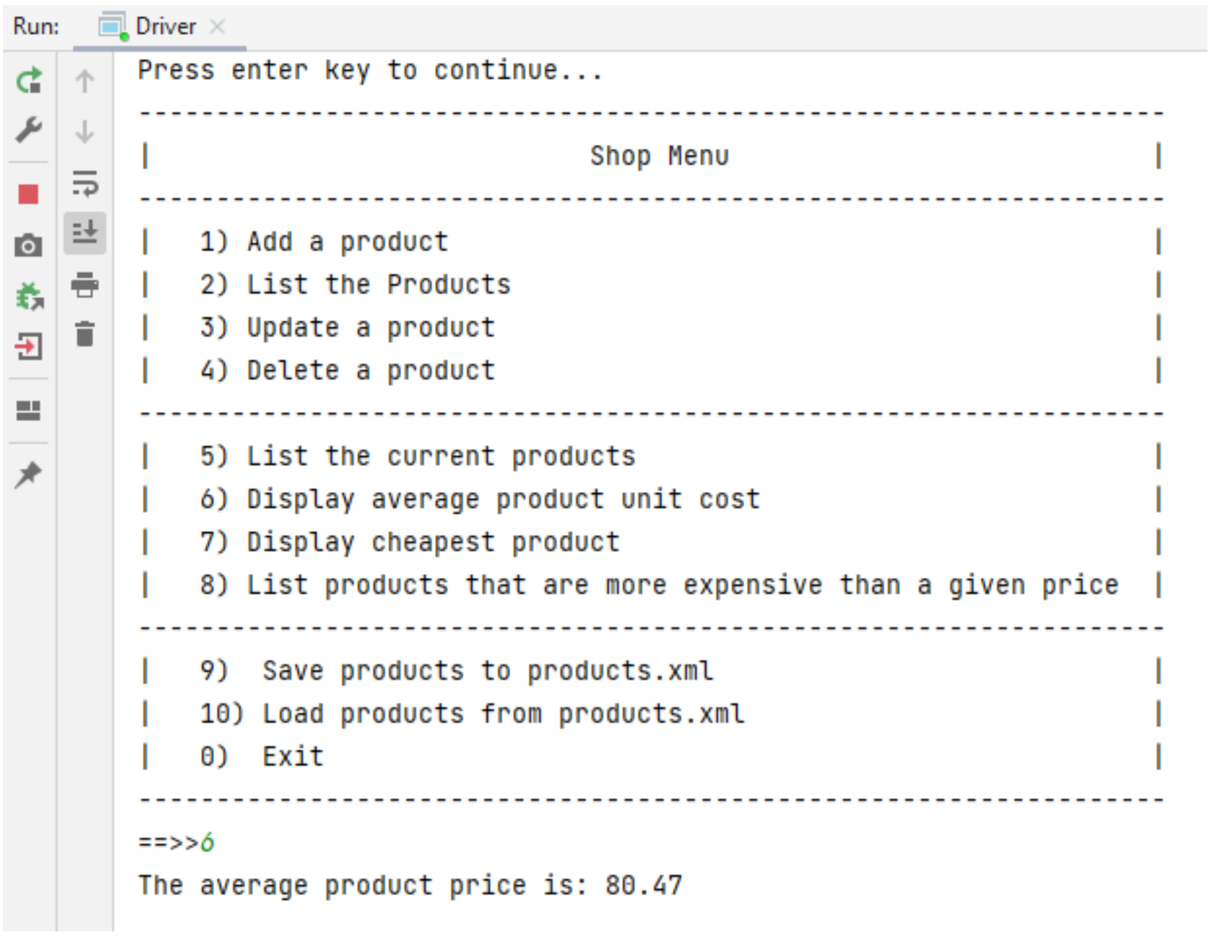
Now we will return to our Store class and the averageProductPrice method which is currently coded like this:

```
    public double averageProductPrice() {
          if (!products.isEmpty()) {
              double totalPrice = 0;
              for (Product product : products) {
                  totalPrice += product.getUnitCost();
              }
              return totalPrice / products.size();
          } else {
              return -1;
          }
      }
```

Before returning the result of the calculation, we want to truncate it to two decimal places using our new utility method, as shown here:

```
    public double averageProductPrice() {
          if (!products.isEmpty()) {
              double totalPrice = 0;
              for (Product product : products) {
                  totalPrice += product.getUnitCost();
              }
              return Utilities.toTwoDecimalPlaces(totalPrice /
products.size());
          } else {
              return -1;
          }
      }
```

When you have the changes made, test the code to make sure the truncating is working when calculating the average price.



```
Press enter key to continue...
-------------------------------------------------------------------
|                          Shop Menu                              |
-------------------------------------------------------------------
|   1) Add a product                                              |
|   2) List the Products                                          |
|   3) Update a product                                           |
|   4) Delete a product                                           |
-------------------------------------------------------------------
|   5) List the current products                                  |
|   6) Display average product unit cost                          |
|   7) Display cheapest product                                   |
|   8) List products that are more expensive than a given price   |
-------------------------------------------------------------------
|   9)  Save products to products.xml                             |
|   10) Load products from products.xml                           |
|   0)  Exit                                                      |
-------------------------------------------------------------------
==>>6
The average product price is: 80.47
```

Another Utility method

Let's add another method to the Utility class.

In our Driver, we have the following code repeated in both the update and the add:

```
    //Ask the user to type in either a Y or an N.  This is then
    //converted to either a True or a False (i.e. a boolean value).
    char currentProduct = ScannerInput.readNextChar("Is this product
in your current line (y/n): ");
    boolean inCurrentProductLine = false;
    if ((currentProduct == 'y') || (currentProduct == 'Y'))
         inCurrentProductLine = true;
```

We are going to write a utility method that will convert a **Y** or a **y** response to true and if any other character is entered, a **false** will be returned.

In your Utilities class, add the following method:

```
    public static boolean YNtoBoolean(char charToConvert){
           return ((charToConvert == 'y') || (charToConvert == 'Y'));
      }
```

Now return to your Driver class and in the addProduct method, use your new utility method:

```
      //gather the product data from the user and create a new
product object - add it to the collection.
      private void addProduct(){

          String productName = ScannerInput.readNextLine("Enter the
Product Name:  ");
          int productCode = ScannerInput.readNextInt("Enter the
Product Code:  ");
          double unitCost = ScannerInput.readNextDouble("Enter the
Unit Cost:  ");

          //Ask the user to type in either a Y or an N.  This is
then
          //converted to either a True or a False (i.e. a boolean
value).
          char currentProduct = ScannerInput.readNextChar("Is this
product in your current line (y/n): ");
          boolean inCurrentProductLine =
Utilities.YNtoBoolean(currentProduct);

          boolean isAdded = store.add(new Product(productName,
productCode, unitCost, inCurrentProductLine));
          if (isAdded){
              System.out.println("Product Added Successfully");
          }
          else{
              System.out.println("No Product Added");
          }
      }
```

and also use it in the updateProduct method too (code snippet):

```
  ...
      //Ask the user to type in either a Y or an N.  This is then
      //converted to either a True or a False (i.e. a boolean
value).
      char currentProduct = ScannerInput.readNextChar("Is this
product in your current line (y/n): ");
      boolean inCurrentProductLine =
Utilities.YNtoBoolean(currentProduct);

      //pass the index of the product and the new product details to
Store for updating and check for success.
      if (store.updateProduct(indexToUpdate, new
Product(productName, productCode, unitCost, inCurrentProductLine))){
          System.out.println("Update Successful");
```

```
        }
...
```
Now run your app again and test that both the add and update work as expected for the Y and N options.

4. Shop V5.0 – Packages

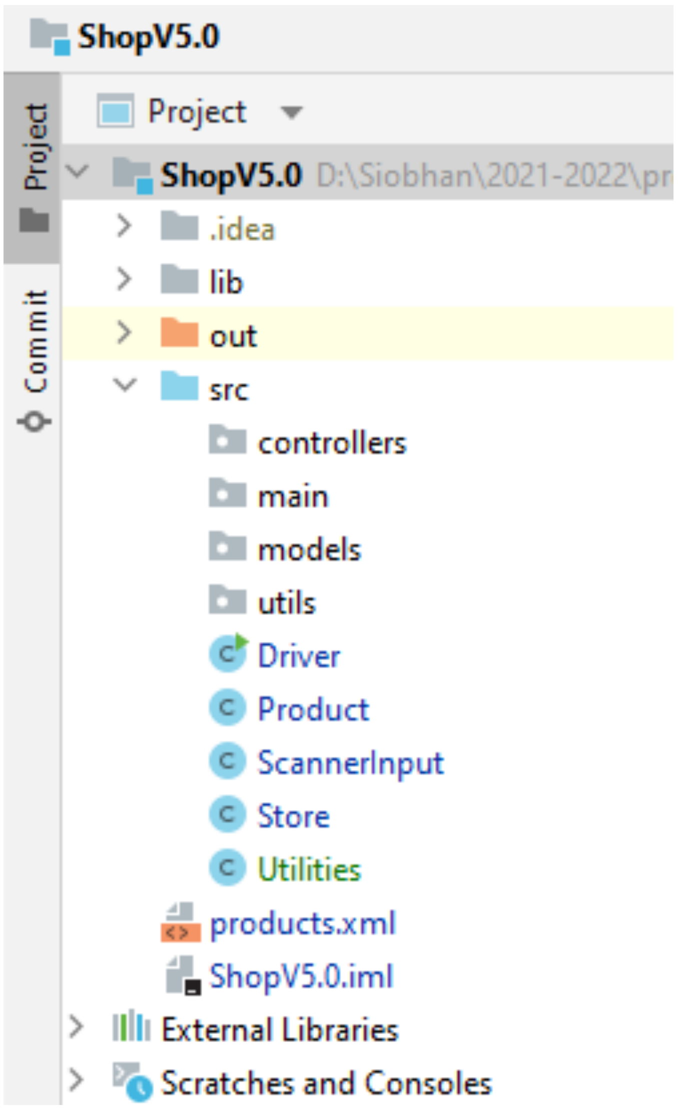
In this step, you will continue working on Shop V5.0 but we will start adding **packages** to our folder structure. This will help us manage our projects as they grow larger by categorising code into different areas.

Adding a package

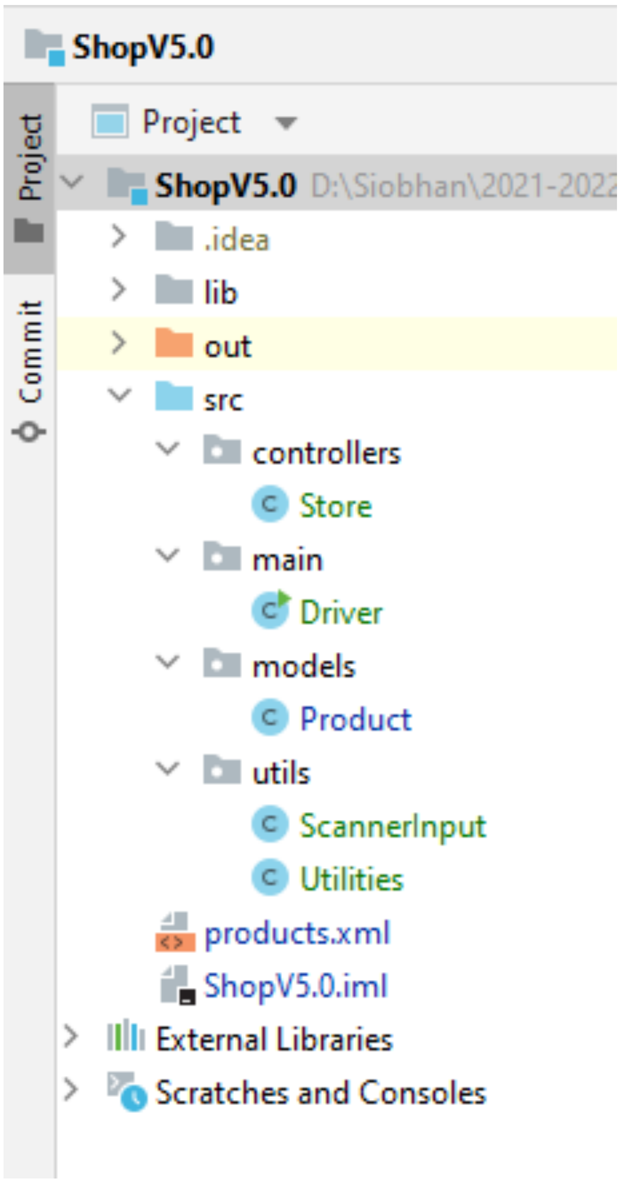
Right click on your src folder and select **New, Package**. Give the package the name **controllers**.

Repeat the same process to add three more packages, one called **utils**, one called **models** and the last called **main**.

Your folder structure should look something like this:



Now move the java code into the respective packages so you have the below structure. You will be prompted to **refactor** - yes, you do want to do this. Your folders should look like this now:



During the refactoring, code changes will have happened in your project. Review your code now and see can you spot some of them e.g.

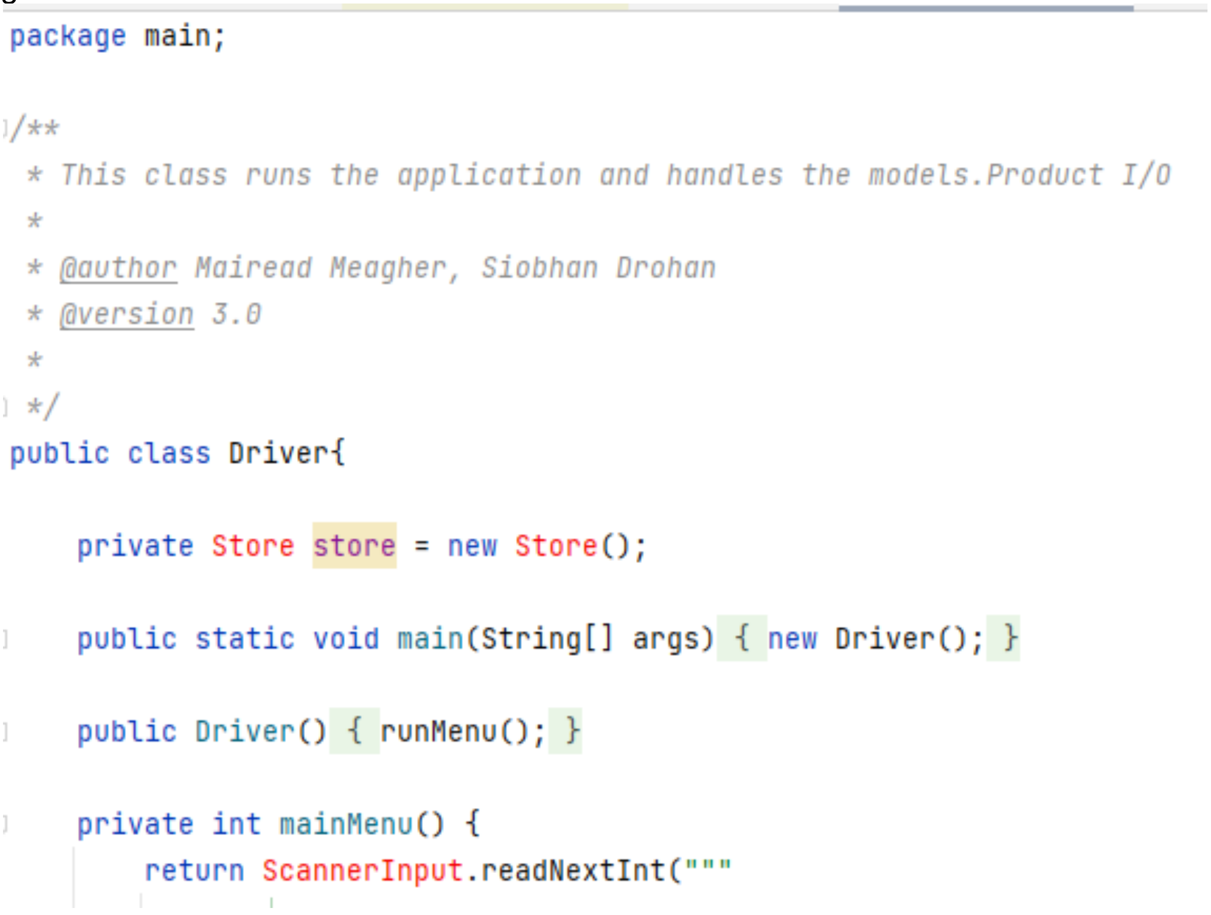
Store.java:

```
package controllers;

import models.Product;
import utils.Utilities;
```

Driver.java:

```
package main;

import controllers.Store;
import models.Product;
import utils.ScannerInput;
import utils.Utilities;
```

If there are new syntax errors in your code, it's probably to do with import issues e.g.:



```
package main;

/**
 * This class runs the application and handles the models.Product I/O
 *
 * @author Mairead Meagher, Siobhan Drohan
 * @version 3.0
 *
 */
public class Driver{

    private Store store = new Store();

    public static void main(String[] args) { new Driver(); }

    public Driver() { runMenu(); }

    private int mainMenu() {
        return ScannerInput.readNextInt("""
```

If this happens to you, click on the errored code (e.g. Store) and press the key combination ALT+Enter. The import should automatically be included at the top of your class OR you will see a context sensitive menu appearing offering options for fixing the error with "import class" as the first option.

Should you encounter errors in this step, the solution for V5.0 is available in the solutions tab.

5. Javadoc

As you code away, good practice is to Javadoc your public classes and methods. A Javadoc comment is enclosed in /** */ and typically uses annotations (@) within it.

You will notice, in the code we gave at the start of this lab, that there are a lot of Javadoc comments included.

For example:

```
package models;
```

```
/**
 * A scaled down version of a models.Product class
 *
 * @author Mairead Meagher, Siobhan Drohan
 * @version 3.0
 */
public class Product {

    private String productName = "";
    private int productCode = -1;
    private double unitCost = 0;
    private boolean inCurrentProductLine = false;

    /**
     * Constructor for objects of class models.Product
     * @param productName Name of the product
     * @param productCode Code of the product
     * @param unitCost Unit cost of the product
     */
    public Product(String productName, int productCode, double
unitCost, boolean inCurrentProductLine) {
        this.productName = productName;
        this.productCode = productCode;
        this.unitCost = unitCost;
        this.inCurrentProductLine = inCurrentProductLine;
    }

    //-------
    //getters
    //-------
    /**
     * Returns the Product Name
     * @return productName
     */
    public String getProductName(){
        return productName;
    }

    /**
     * Returns the Unit Cost
     * @return unitCost
     */
    public double getUnitCost(){
        return unitCost;
    }

    /**
     * Returns the Product Code
     * @return productCode
```

```
         */
        public int getProductCode() {
            return productCode;
        }

        /**
         * Returns a boolean indicating if the product is in the
current product line
         * @return inCurrentProductLine
         */
        public boolean isInCurrentProductLine() {
            return inCurrentProductLine;
        }

        //-------
        //setters
        //-------
        /**
         * Updates the Product Code to the value passed as a parameter
         * @param productCode The new Product Code
         */
        public void setProductCode(int productCode) {
            this.productCode = productCode;
        }

        /**
         * Updates the Product Name to the value passed as a parameter
         * @param productName The new Product Name
         */
        public void setProductName(String productName) {
            this.productName = productName;
        }

        /**
         * Updates the Unit Cost to the value passed as a parameter
         * @param unitCost The new unit cost for the product
         */
        public void setUnitCost(double unitCost) {
            this.unitCost = unitCost;
        }

        /**
         * Updates the boolean indicating whether the product is in
the current product line or not.
         * @param inCurrentProductLine Indicator that determines if
the product is in the current product line or not.
         */
        public void setInCurrentProductLine(boolean
inCurrentProductLine) {
```

```
            this.inCurrentProductLine = inCurrentProductLine;
        }

        /**
         * Builds a String representing a user friendly representation
of the object state
         * @return Details of the specific product
         */
        public String toString()
        {
            return "models.Product description: " + productName
                    + ", product code: " + productCode
                    + ", unit cost: " + unitCost
                    + ", currently in product line: " +
inCurrentProductLine;
        }

    }
```

Class comments

A class comment is placed directly above the class definition and typically describes the single responsibility of the class. If you look at the class comment for this class, you will notice that we are using two annotations; one to indicate the authors and the other to indicate the version number.

```
    /**
     * A scaled down version of a models.Product class
     *
     * @author Mairead Meagher, Siobhan Drohan
     * @version 3.0
     */
```

Method comments

Method comments are placed directly above the associated public method (note, you don't Javadoc private methods) and typically contain @param and/or @return annotations. See their use in the getter and setter below:

```
        /**
         * Returns a boolean indicating if the product is in the
current product line
         * @return inCurrentProductLine
         */
        public boolean isInCurrentProductLine() {
            return inCurrentProductLine;
        }
```

```
      /**
       * Updates the boolean indicating whether the product is in
the current product line or not.
       * @param inCurrentProductLine Indicator that determines if
the product is in the current product line or not.
       */
      public void setInCurrentProductLine(boolean
inCurrentProductLine) {
          this.inCurrentProductLine = inCurrentProductLine;
      }
```

It is good practice to write Javadoc comments as soon as you have finished testing a class/method and are happy that it is working as expected. In future labs, we will show you how to generate a static website from your Javadoc comments.