# Lesson 06 Demo 10

# Implementing Streams with User Defined Objects

**Objective:** To implement streams with user-defined objects to efficiently process and manipulate collections of custom data types in a functional programming style

**Tools Required:** Eclipse IDE

**Prerequisites:** None

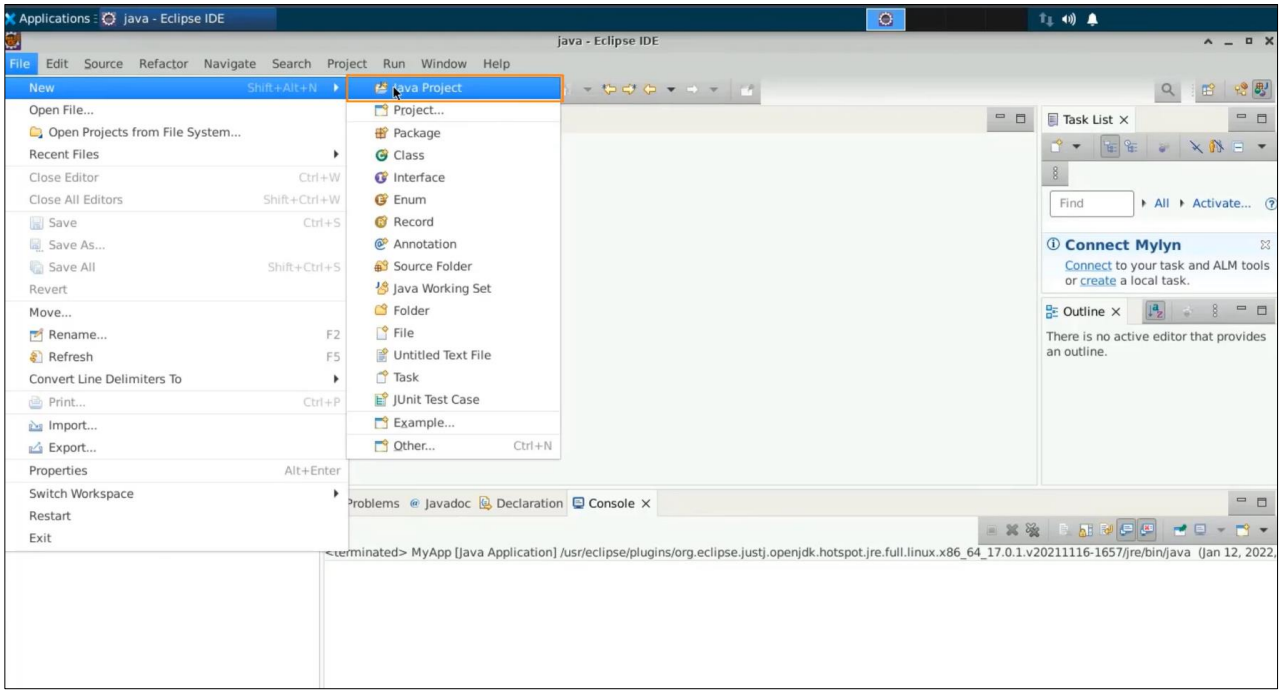
Steps to be followed:

1. Create a new Java project, followed by a class
2. Create constructors and define certain attributes for the class
3. Execute the code
4. Work with streams and re-execute the code
5. Create some mapped products with sample data
6. Write a new user object and run the code

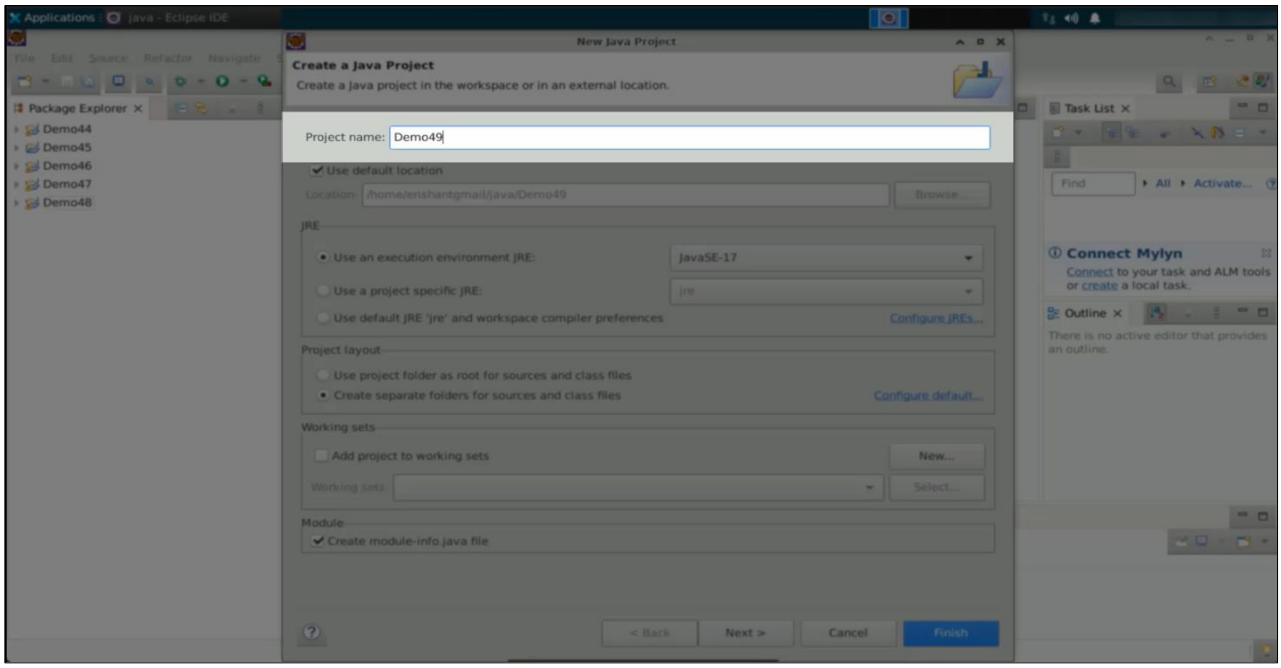
## Step 1: Create a new java project, followed by a class

1.1 Open the **Eclipse IDE**

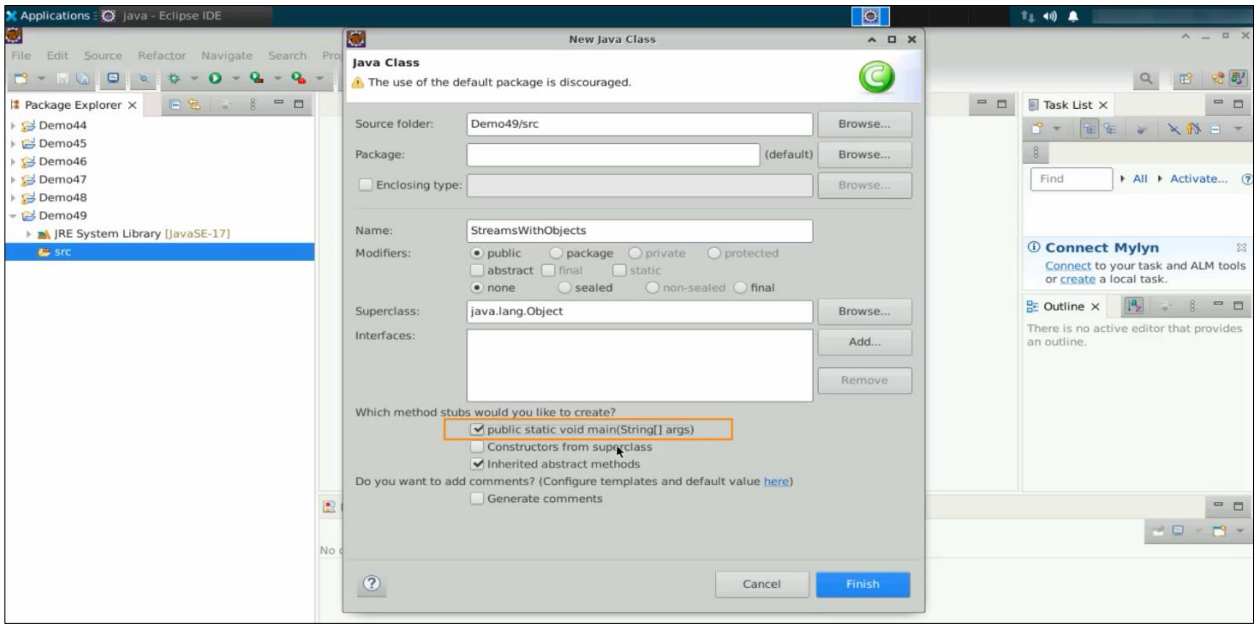
1.2 Select **File**, then **New,** and click **Java project**



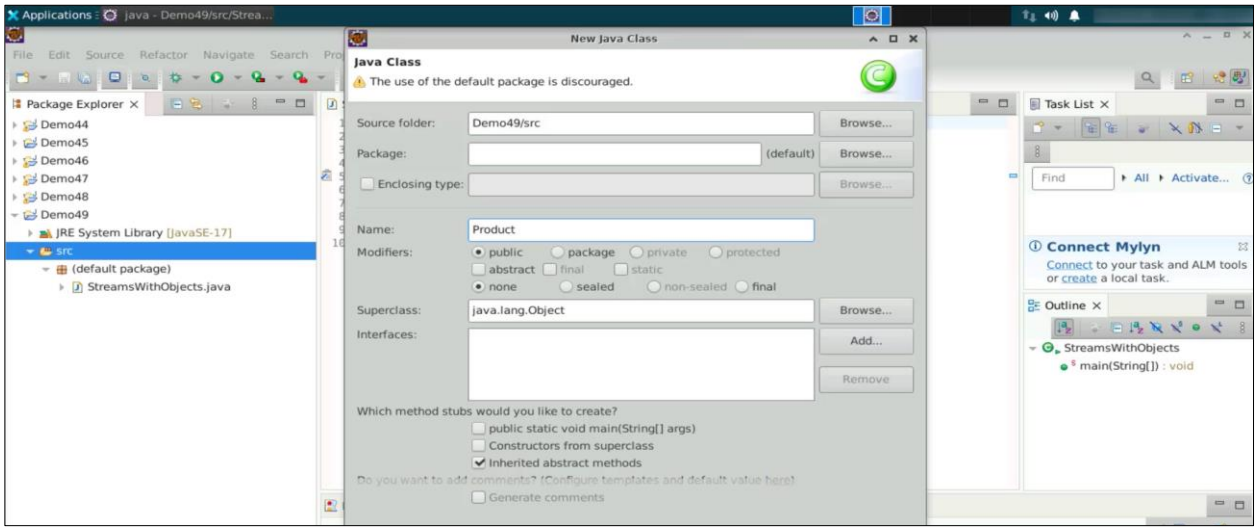

1.3 Name the project as **Demo49**, uncheck the option **Create a module-info.java file**, and click **Finish**



1.4 With the **Demo49** project selected in the Project Explorer, right-click and choose **New Class**. Name the class **StreamsWithObjects**, select the main method, and click **Finish**

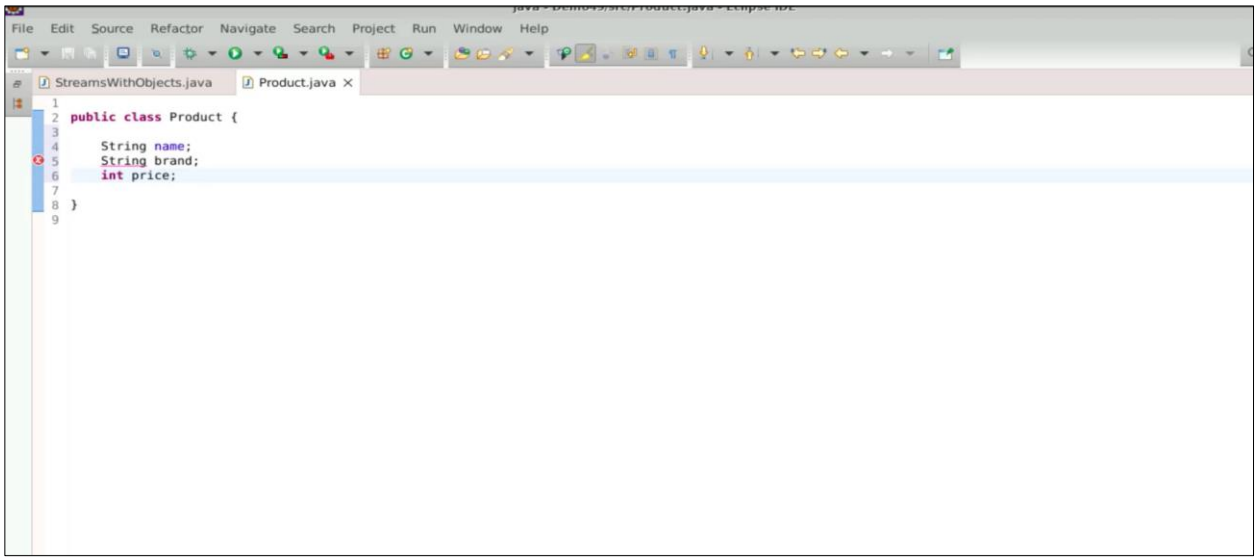



1.5 Now, define a Product class. Create a new class and name it as **Product**.

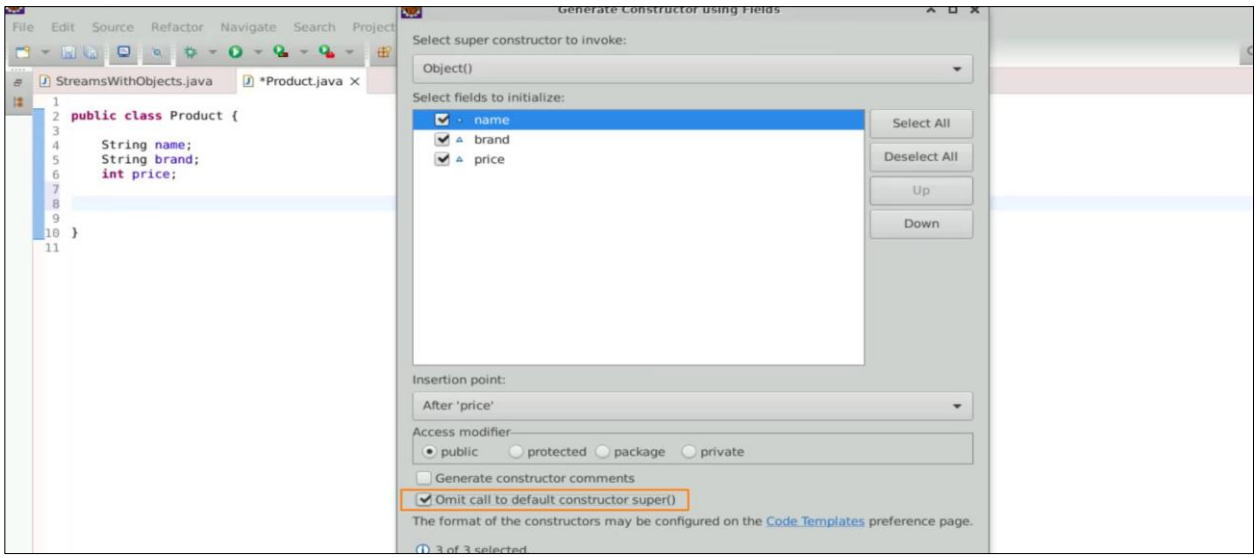


## Step 2: Create constructors and define certain attributes for the class

2.1 In the **Product** class, define certain attributes. Mention the attribute's **name**, **brand,** and **price**



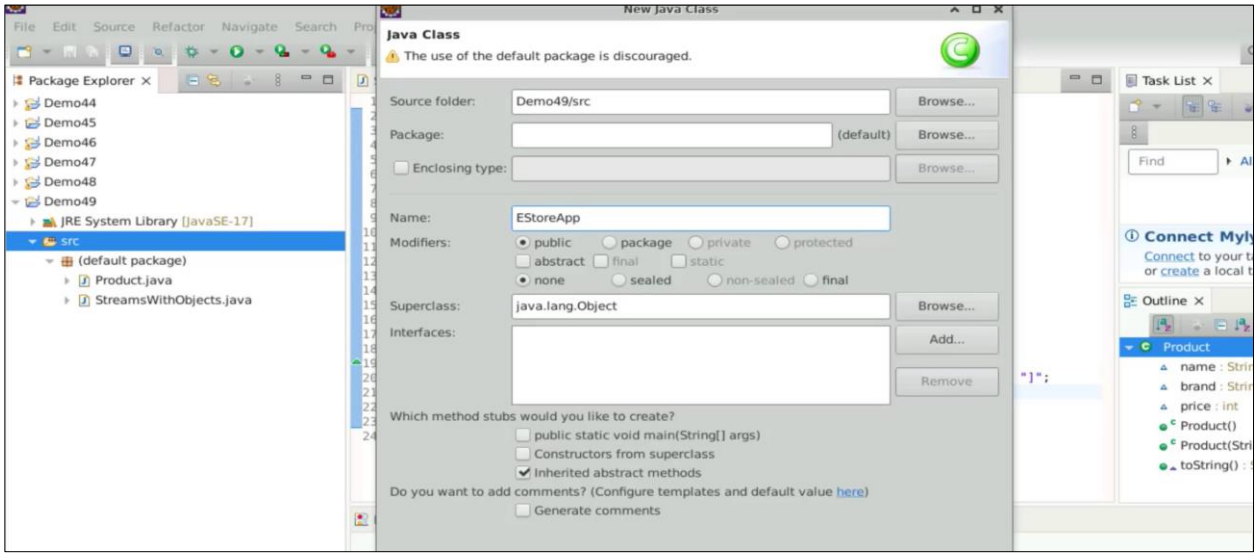

2.2 Now, create the constructors. First generate a constructor that takes parameters for the fields (name, brand, and price). Then, generate a default constructor with no parameters.



2.3 Finally, override the **toString** method in the **Product** class to display the data inside the product object. This will provide a basic representation of the **Product** object. Since we are not using packages, the data can be accessed from anywhere even if it is set as default access

```
public class Product {

    String name;
    String brand;
    int price;

    public Product() {

    }

    public Product(String name, String brand, int price) {
        this.name = name;
        this.brand = brand;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product [name=" + name + ", brand=" + brand + ", price=" + price + "]";
    }

}
```

2.4 Now, create another class called **EStoreApp** to represent the E-Store application



2.5 In the **EStoreApp** constructor, initialize an **ArrayList** of type **Product**. Inside the constructor, we will add new **Product** objects. For example, add an iPhone with the brand **Apple** and the price of **70000**

```java
import java.util.ArrayList;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));

    }

}
```

2.6 Similarly, add a few more products

```java
import java.util.ArrayList;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Water Bottle", "Hidrate", 5000));
        products.add(new Product("Alphabounce", "Adidas", 6000));

    }

}
```

2.7 Now, create a method called **showProductsInEStore** to display the product details. The method will iterate over the products and print the name, brand, and price with proper formatting

```java
import java.util.ArrayList;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Water Bottle", "Hidrate", 5000));
        products.add(new Product("Alphabounce", "Adidas", 6000));
    }

    void showProductInEStore() {

    }

}
```

2.8 In the **showProductsInEStore** method, display the product details using the **showProduct** method of the **Product** class. We will add a delimiter for better representation and include an empty line after displaying each product

```java
public class Product {

    String name;
    String brand;
    int price;

    public Product() {

    }

    public Product(String name, String brand, int price) {
        this.name = name;
        this.brand = brand;
        this.price = price;
    }

    void showProduct() {
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.println(name+"\t"+brand+"\t"+price);
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.println();
    }

    @Override
    public String toString() {
        return "Product [name=" + name + ", brand=" + brand + ", price=" + price + "]";
    }

}
```

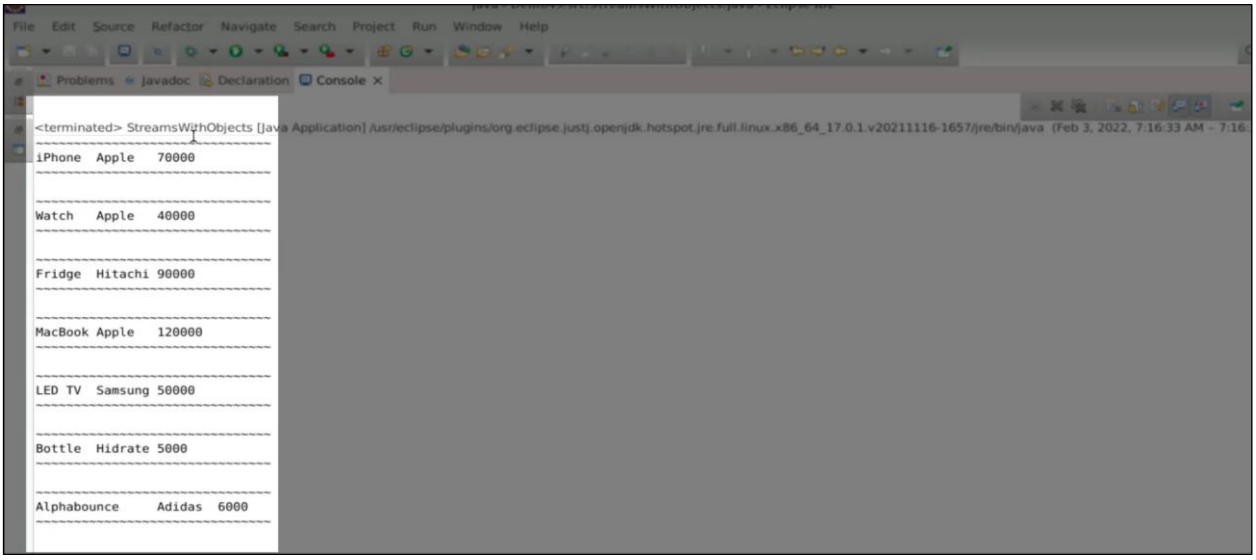
2.9 Now, back in the **EStoreApp** class, we will iterate over the products using a lambda expression. For each product, we will call the **showProduct** method to display its details

```java
import java.util.ArrayList;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Alphabounce", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

}
```

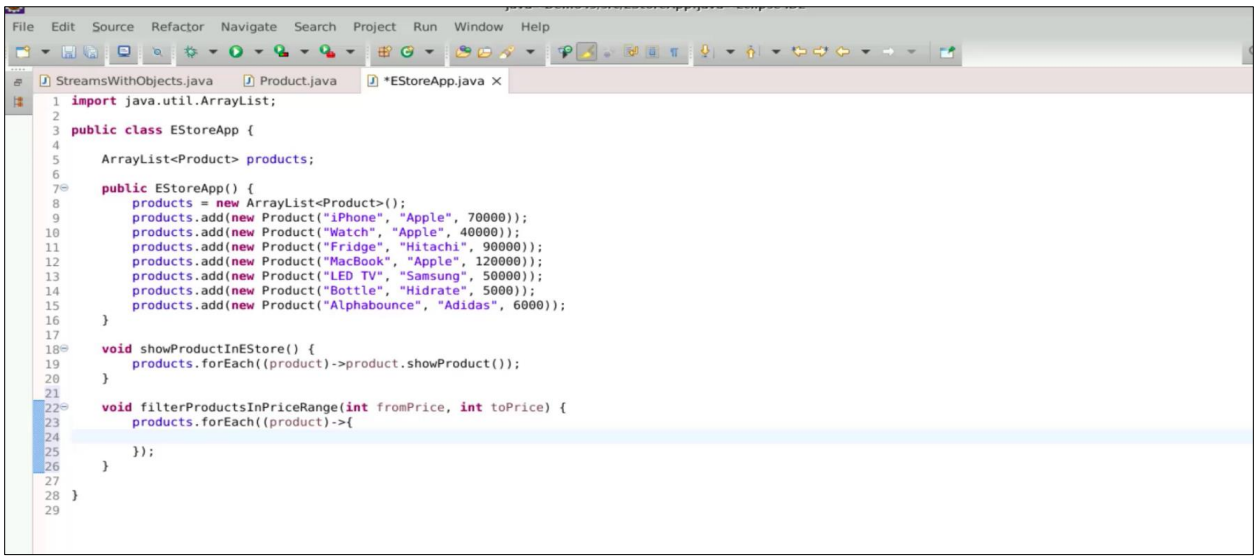
2.10 In the main method of the **StreamsWithObjects** class, we will create an instance of the **EStoreApp** class by using the constructor. This will automatically execute the constructor and initialize the product list. Then, we will call the **showProductsInEStore** method of the **EStoreApp** object to display the products

```java
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        app.showProductInEStore();

    }

}
```

2.11 The products will be displayed using a loop, showing the details of each product. At this point, you can provide users with options to filter the products according to there preferences




2.12 In the **EStoreApp** class, we have a method called **filterProductsInPriceRange** which takes two integer inputs: **fromPrice** and **toPrice**. Here, you can implement a lambda expression to filter the products based on the price range



2.13 Inside the lambda expression, we can define the filtering condition. For example, if the Price of the product is greater than or equal to **fromPrice** and less than or equal to **toPrice**, we can call the **showProduct** method to display that product

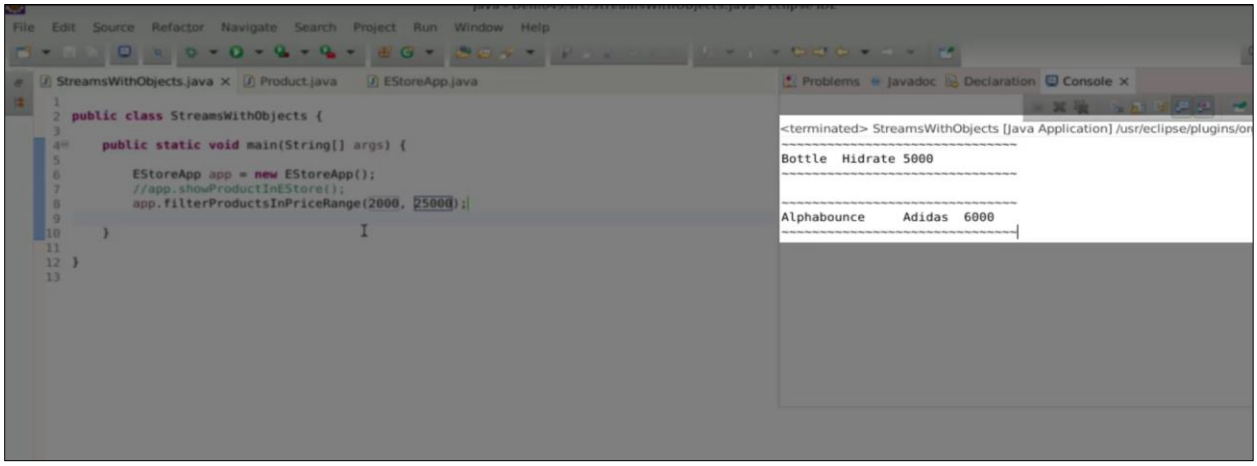
```java
import java.util.ArrayList;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Alphabounce", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });
    }

}
```

2.14 Returning to the main method, we have the option to filter the products in the price range using the **filterProductsInPriceRange** method. For instance, if we want to filter products in the price range of **2000** to **25000**, we can call this method with the appropriate arguments

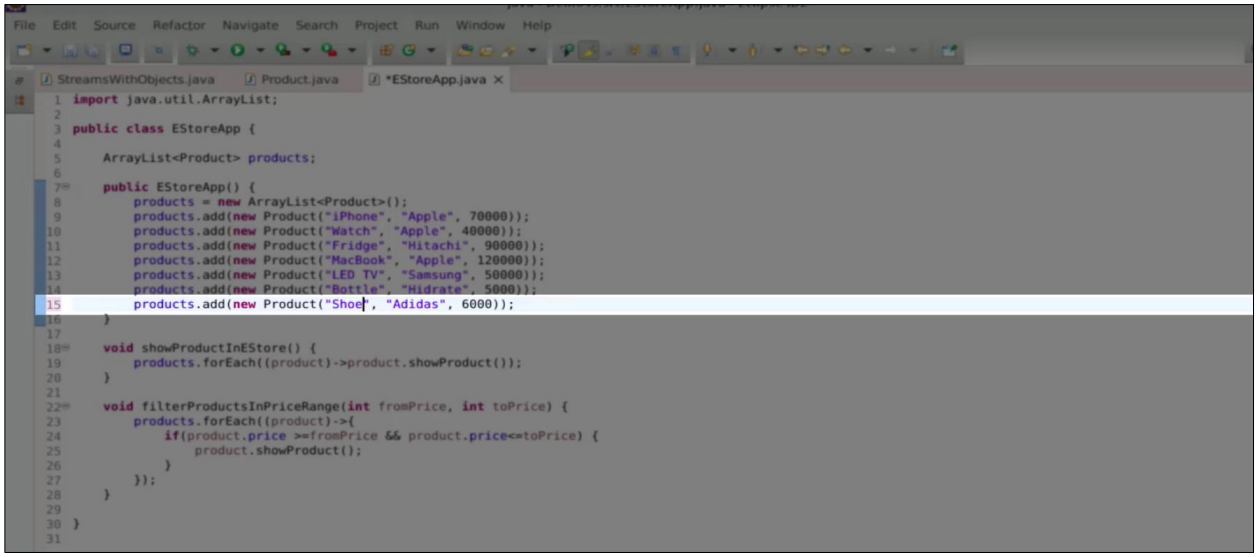
```java
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        //app.showProductInEStore();
        app.filterProductsInPriceRange(2000, 25000);

    }

}
```

## Step 3: Execute the code with example data

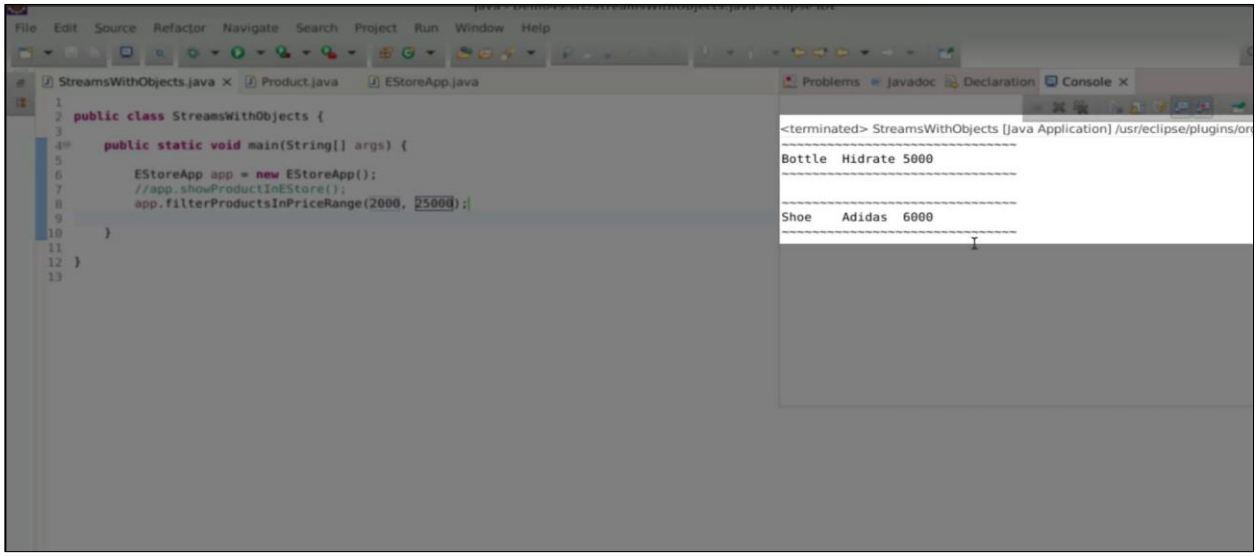
3.1 Run the code, you will observe that only two products are displayed instead of all the products. This is due to the applied filtration



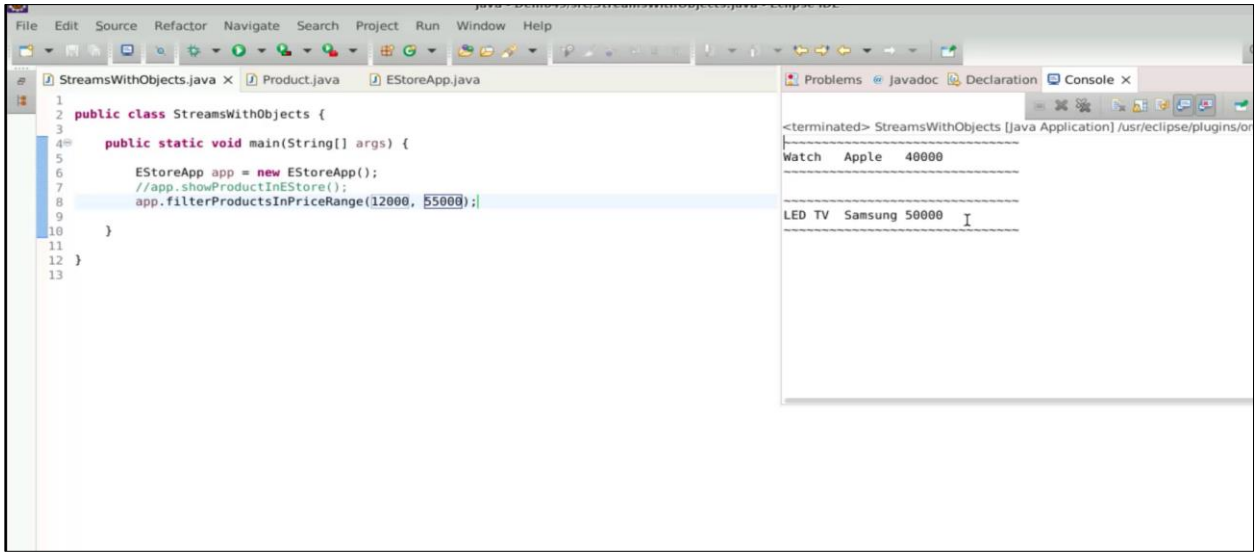

3.2 In the **EStoreApp**, if you examine the products, you will notice that all the products have prices greater than 25000. However, after filtering, only the water bottle and the shoe fall within the specified price range. If you modify the code to filter based on the shoe, you can rerun the code to see the updated results



3.3 The displayed products correspond to the filtration criteria defined




3.4 Now, modify the filtration criteria again, setting it from **12,000** to **55,000**. After rerunning the code, you will once again see the two products that satisfy this new price range



3.5 If you choose a price range between **2000** and **55000**, you will have four products displayed. This approach utilizes a simple if condition to apply the filter

```
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        //app.showProductInEStore();
        app.filterProductsInPriceRange(2000, 55000);

    }

}
```

```
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/or
Watch   Apple   40000

LED TV  Samsung 50000

Bottle  Hidrate 5000

Shoe    Adidas  6000
```

## Step 4: Work with streams and re-execute the code with example data

4.1 Now, you can obtain a stream object of type Product by calling **products.stream()**. This creates a stream from the products list. Then, apply a **filter** function on the stream, passing the filtering criteria. In this case, the criteria is that the product price falls within a specified range. Below is the updated code snippet for the filter function:

```
import java.util.ArrayList;
import java.util.stream.Stream;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        Stream<Product> stream = products.stream();
        stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
```

4.2 The filter function will return another stream called filteredProducts. You can replace the previous instance of filterProducts with this updated variable. Next, you can use the forEach method on the filteredProducts stream to iterate over each product and display its details using the showProduct method. This is an implementation utilizing streams on user-defined objects

```java
import java.util.ArrayList;
import java.util.stream.Stream;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());
    }

}
```

4.3 Upon running this code, you will obtain the same output as before, but this implementation is achieved using streams

```java
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        //app.showProductInEStore();
        app.filterProductsInPriceRange(2000, 55000);

    }

}
```

```
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/or
Watch   Apple   40000

LED TV  Samsung 50000

Bottle  Hidrate 5000

Shoe    Adidas  6000
```

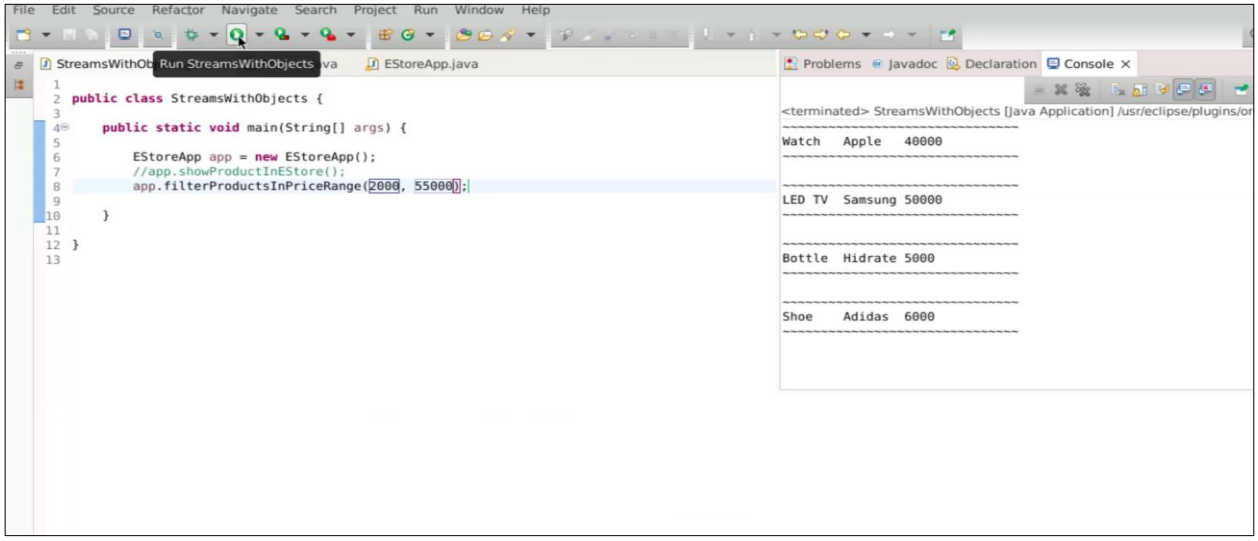
4.4 Now, improve the code further. Instead of assigning the stream to a variable and then using it, chain the operations together. Starting with **products.stream()**, you can apply the filter function directly on the stream, followed by the **forEach** method to display the product details.

```java
import java.util.ArrayList;
import java.util.stream.Stream;

public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        /*Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());*/

        products.stream().filter((product)->product.price >=fromPrice && product.price<=toPrice).forEach((product)->product.showProduct());
    }

}
```

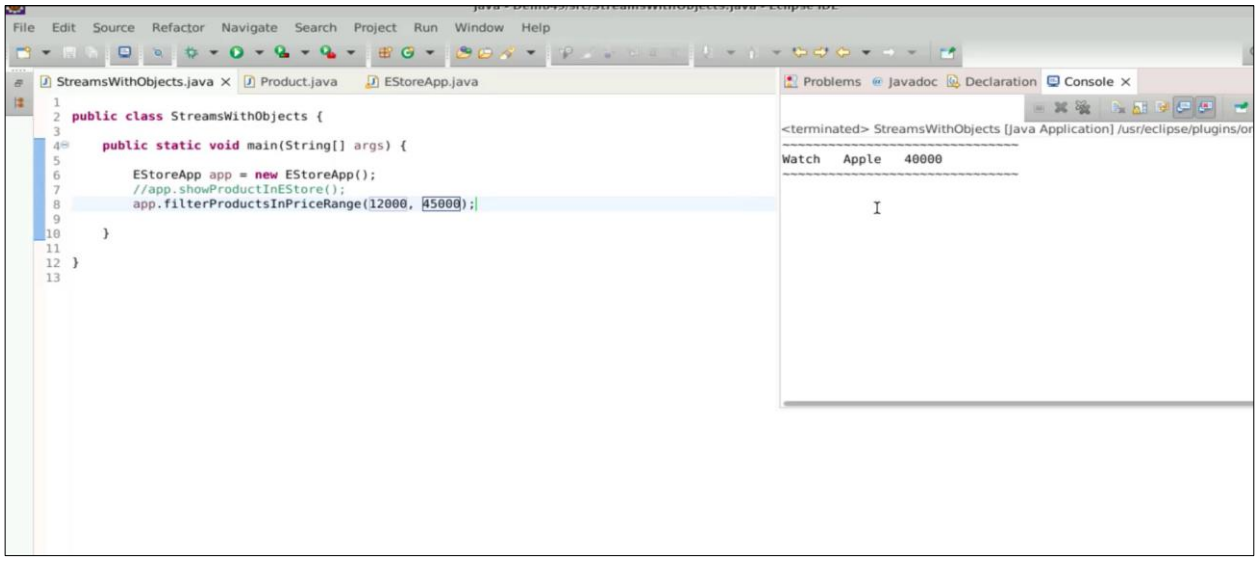
4.5 You can break the code at the points **products.stream()** and **forEach** for a more concise implementation

```java
public class EStoreApp {

    ArrayList<Product> products;

    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        /*Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());*/

        products.stream()
            .filter((product)->product.price >=fromPrice && product.price<=toPrice)
            .forEach((product)->product.showProduct());
    }

}
```

4.6 Run the code again you get the same output as before




4.7 Now, modify the price range from **12,000** to **45,000**. After running the code, you will observe that only one product satisfies the new filtering criteria. The filter function allows you to work with various criteria and obtain the desired products accordingly



4.8 Next, list all the products, add an empty line, and include a small delimiter. Then, add another empty line and label it as **Filtered products**. This will enhance the representation of the products and their filtering results

```
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();
        System.out.println();
        System.out.println("==============================");
        System.out.println();
        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

    }

}
```

## Step 5: Create some mapped products with sample data

5.1 After running the code, you will see the filtered products within the specified price range

```
public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

        System.out.println();
        System.out.println("==============================");
        System.out.println();

    }

}
```

```
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/or
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shoe    Adidas  6000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

==============================
FILTERED PRODUCTS
Watch   Apple   40000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

==============================
```

5.2 Now, create a method called **showProductsToPrimeMember()**. For prime members, you can provide a flat discount of **500** units on all products. Declare an integer variable named **flatDiscount** and set it to **500**, representing the flat discount amount for prime members.

```java
    public EStoreApp() {
        products = new ArrayList<Product>();
        products.add(new Product("iPhone", "Apple", 70000));
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        /*Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());*/

        products.stream()
            .filter((product)->product.price >=fromPrice && product.price<=toPrice)
            .forEach((product)->product.showProduct());
    }

    void showProductsToPrimeMember() {
        int flatDiscount = 500;

    }

}
```

5.3 Now, obtain the stream object of products using **products.stream()**. Then, apply the **map** function on the stream, where the input is a product, and the output is the updated price of product after deducting the flat discount.

```java
        products.add(new Product("Watch", "Apple", 40000));
        products.add(new Product("Fridge", "Hitachi", 90000));
        products.add(new Product("MacBook", "Apple", 120000));
        products.add(new Product("LED TV", "Samsung", 50000));
        products.add(new Product("Bottle", "Hidrate", 5000));
        products.add(new Product("Shoe", "Adidas", 6000));
    }

    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        /*Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());*/

        products.stream()
            .filter((product)->product.price >=fromPrice && product.price<=toPrice)
            .forEach((product)->product.showProduct());
    }

    void showProductsToPrimeMember() {
        int flatDiscount = 500;
        Stream<Product> stream = products.stream();
        stream.map((product)->{
            product.price -= flatDiscount;
            return product;
        });
    }

}
```

5.4 To simplify the code, modify the **map** function to return the product itself, but with the price updated as **product.getPrice() - flatDiscount**. Assign this stream to **mappedProducts** and then use **forEach** to iterate over the mapped products and display their details using the **showProduct** method.

```
File Edit Source Refactor Navigate Search Project Run Window Help
StreamsWithObjects.java  Product.java  EStoreApp.java
11          products.add(new Product("Watch", "Apple", 40000));
12          products.add(new Product("Fridge", "Hitachi", 90000));
13          products.add(new Product("MacBook", "Apple", 120000));
14          products.add(new Product("LED TV", "Samsung", 50000));
15          products.add(new Product("Bottle", "Hidrate", 5000));
16          products.add(new Product("Shoe", "Adidas", 6000));
17      }
18
19      void showProductInEStore() {
20          products.forEach((product)->product.showProduct());
21      }
22
23      void filterProductsInPriceRange(int fromPrice, int toPrice) {
24          /*products.forEach((product)->{
25              if(product.price >=fromPrice && product.price<=toPrice) {
26                  product.showProduct();
27              }
28          });*/
29
30          /*Stream<Product> stream = products.stream();
31          Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
32          filteredProducts.forEach((product) -> product.showProduct());*/
33
34          products.stream()
35              .filter((product)->product.price >=fromPrice && product.price<=toPrice)
36              .forEach((product)->product.showProduct());
37      }
38
39      void showProductsToPrimeMember() {
40          int flatDiscount = 500;
41          Stream<Product> stream = products.stream();
42          Stream<Product> mappedProducts = stream.map((product)->{
43              product.price -= flatDiscount;
44              return product;
45          });
46          mappedProducts.forEach((product)->product.showProduct());
47      }
48
```

5.5 Execute the **showProductsToPrimeMember()** method using **app.showProductsToPrimeMember()**. Running the code will show the products with a discount of 500 for each item.

```
File Edit Source Refactor Navigate Search Project Run Window Help
StreamsWithObjects.java  Product.java  EStoreApp.java
1
2  public class StreamsWithObjects {
3
4      public static void main(String[] args) {
5
6          EStoreApp app = new EStoreApp();
7          System.out.println("ALL PRODUCTS");
8          app.showProductInEStore();
9
10         System.out.println();
11         System.out.println("==============================");
12         System.out.println();
13
14         System.out.println("FILTERED PRODUCTS");
15         app.filterProductsInPriceRange(12000, 45000);
16
17         System.out.println();
18         System.out.println("==============================");
19         System.out.println();
20
21         System.out.println("MAPPED PRODUCTS");
22         app.showProductsToPrimeMember();
23
24     }
25
26 }
27
```

```
Problems  Javadoc  Declaration  Console
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/or
MacBook Apple   119500
LED TV  Samsung 49500
Bottle  Hidrate 4500
Shoe    Adidas  5500
```

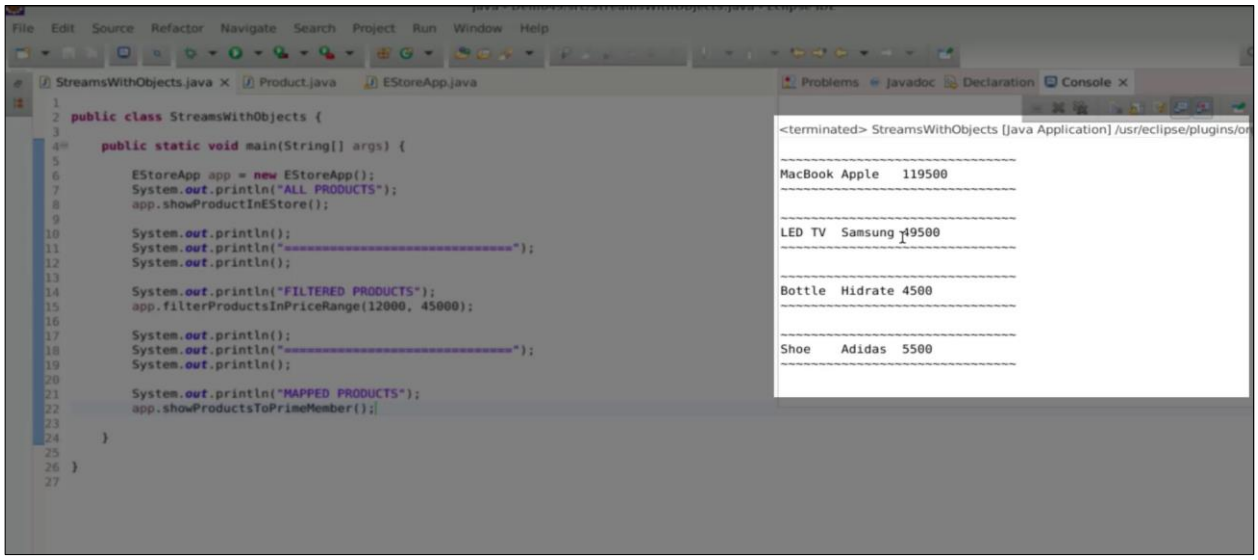
5.6 You will observe that the amount of 500 has been subtracted in the price of each product, indicating the discount for prime members

```
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.1.v20211116-1657/jre/bin/java (Feb 3, 2022, 7:29:54 AM – 7:29:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MacBook Apple   120000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
LED TV  Samsung 50000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Bottle  Hidrate 5000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shoe    Adidas  6000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


==============================
FILTERED PRODUCTS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Watch   Apple   40000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


==============================
MAPPED PRODUCTS
```

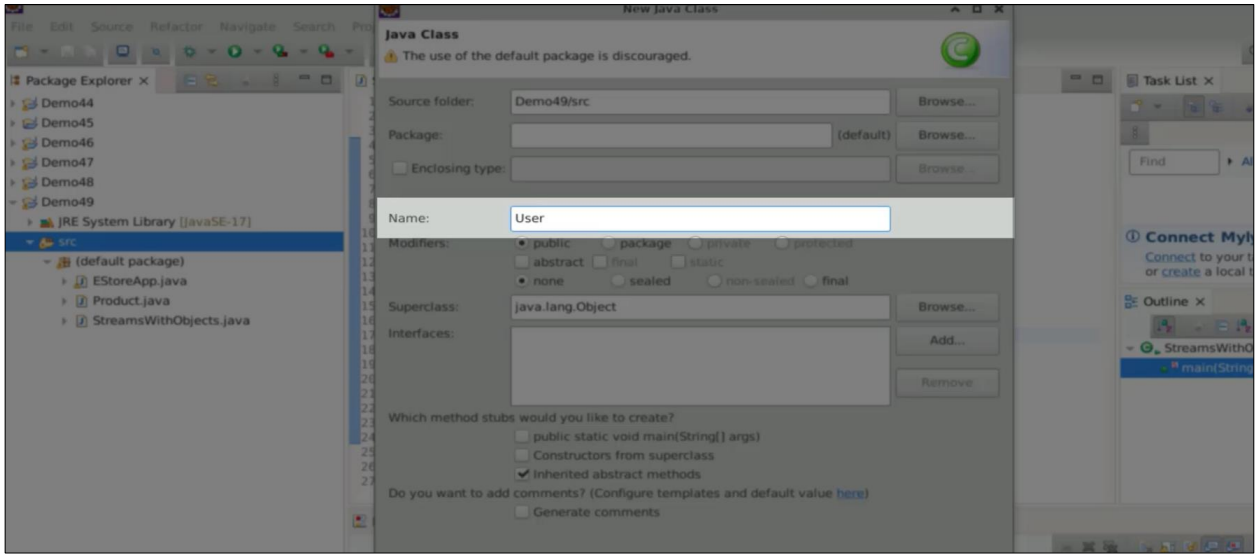
5.7 Alternatively, you can write the entire code in a single line. Start with **products.stream(),** then copy and paste the **map** code. After that, **append .forEach(product -> product.showProduct())**. This condensed approach allows for a more concise and efficient syntax, especially as you become more familiar with these techniques

```
    void showProductInEStore() {
        products.forEach((product)->product.showProduct());
    }

    void filterProductsInPriceRange(int fromPrice, int toPrice) {
        /*products.forEach((product)->{
            if(product.price >=fromPrice && product.price<=toPrice) {
                product.showProduct();
            }
        });*/

        /*Stream<Product> stream = products.stream();
        Stream<Product> filteredProducts = stream.filter((product)->product.price >=fromPrice && product.price<=toPrice);
        filteredProducts.forEach((product) -> product.showProduct());*/

        products.stream()
            .filter((product)->product.price >=fromPrice && product.price<=toPrice)
            .forEach((product)->product.showProduct());
    }

    void showProductsToPrimeMember() {
        int flatDiscount = 500;
        /*Stream<Product> stream = products.stream();
        Stream<Product> mappedProducts = stream.map((product)->{
            product.price -= flatDiscount;
            return product;
        });
        mappedProducts.forEach((product)->product.showProduct());*/

        products.stream()
            .map((product)->{
                product.price -= flatDiscount;
                return product;
            })
            .forEach((product)->product.showProduct());
    }

}
```

5.8 When executing the code in the **EStoreApp**, you will obtain the same result as before




5.9 Now, let us consider a scenario where a user has a shopping cart with multiple products. In the **EStore** package, create a new class called **User**



5.10 Inside the **User** class, include a **String** variable for the user **name** and an **ArrayList** of type **Product** named **cart** to store the selected user products. Create a constructor for the **User** class that initializes these fields

```java
import java.util.ArrayList;

public class User {

    String name;
    ArrayList<Product> cart;

    User(){

    }

    public User(String name, ArrayList<Product> cart) {
        this.name = name;
        this.cart = cart;
    }


}
```

5.11 Next, define a method named **showCart()** with a **void** return type. Within this method, print the contents of the cart using **System.out.println(cart)**. Then, use **cart.forEach(product -> product.showProduct())** to iterate over each product in the cart and display its details using the **showProduct** method.

```java
import java.util.ArrayList;

public class User {

    String name;
    ArrayList<Product> cart;

    User(){

    }

    public User(String name, ArrayList<Product> cart) {
        this.name = name;
        this.cart = cart;
    }

    void showCart() {
        System.out.println("CART");
        cart.forEach((product)->product.showProduct());
    }

}
```

## Step 6: Write a new user object and run the code with an example dataset

6.1 Now, create a new User object named John by calling the constructor with the name **John**. For the shopping cart, you can assign it as **app.getProductList().subList(2, 7)**, which represents products from index **2** to index **6**. Make sure to change the data type to **List<Product>** for convenience

```java
import java.util.ArrayList;
import java.util.List;

public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("MAPPED PRODUCTS");
        app.showProductsToPrimeMember();


        List<Product> shoppingCart = app.products.subList(2, 6);

        User user = new User("John", null);

    }
```

6.2 So now, link this shopping cart to the john user object. To make things more convenient, you can use the polymorphic type List instead of **ArrayList**. Now, you can call **user.showCart()** on the John object to display the products in his cart.

```java
import java.util.ArrayList;
import java.util.List;

public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("MAPPED PRODUCTS");
        app.showProductsToPrimeMember();


        List<Product> shoppingCart = app.products.subList(2, 6);
        User user = new User("John", shoppingCart);
        user.showCart();

    }
```

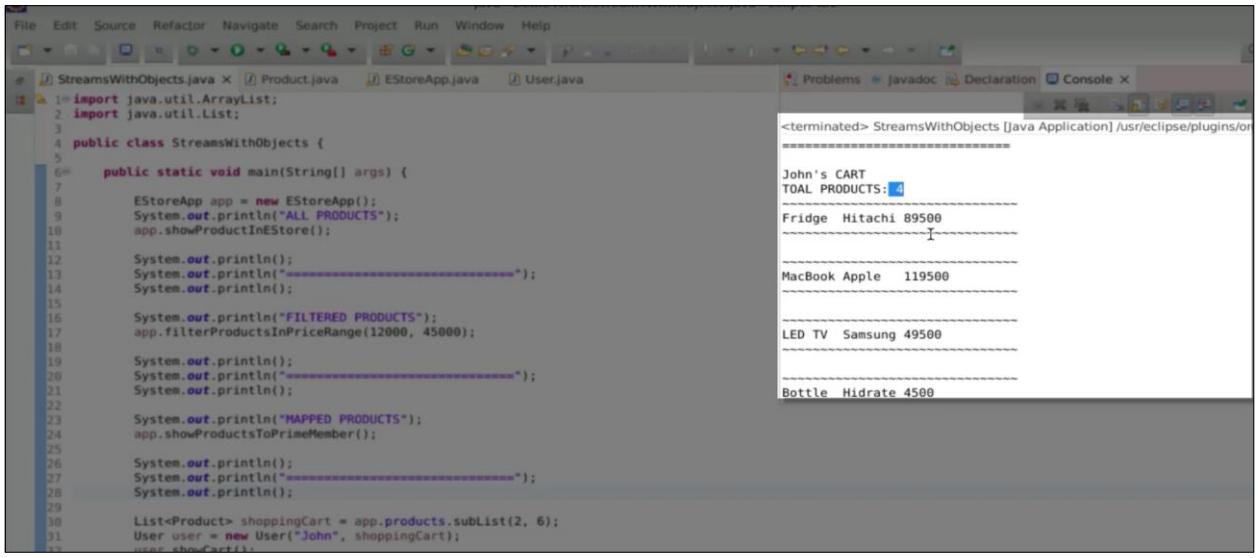
6.3 Add a delimiter to separate the outputs

```java
import java.util.ArrayList;
import java.util.List;

public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("MAPPED PRODUCTS");
        app.showProductsToPrimeMember();

        System.out.println();
        System.out.println("==============================");
        System.out.println();os

        List<Product> shoppingCart = app.products.subList(2, 6);
        User user = new User("John", shoppingCart);
```

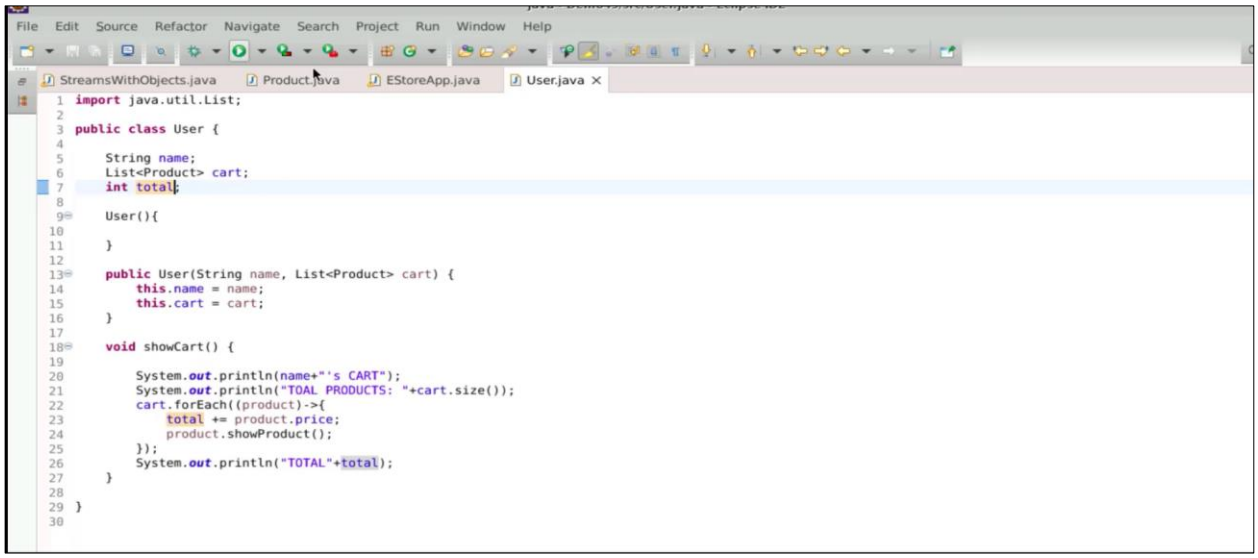
6.4 Inside the **showCart** method of the User class, display the username and the name of the cart. You can add a line saying **Total products** and set it as **cart.size()**, which represents the total number of products in the cart.

```java
import java.util.List;

public class User {

    String name;
    List<Product> cart;

    User(){

    }

    public User(String name, List<Product> cart) {
        this.name = name;
        this.cart = cart;
    }

    void showCart() {
        System.out.println(name+"'s CART");
        System.out.println("TOAL PRODUCTS: "+cart.size());
        cart.forEach((product)->product.showProduct());
    }

}
```

6.5 Return to the main method and run the code. You will see that John's cart contains **4** products, and since the products were mapped with a flat discount, they are displayed at a reduced price.




6.6 Next, create a variable named **total** and initialize it to 0. While iterating over the products in **showCart(),** increment the total by **product.getPrice()**. Define the total variable within the **showCart()** method.



6.7 Run the code again. Now, when displaying the cart, you will also see the total value of the cart. It shows that there are four products worth the total amount displayed.

```
import java.util.ArrayList;
import java.util.List;

public class StreamsWithObjects {

    public static void main(String[] args) {

        EStoreApp app = new EStoreApp();
        System.out.println("ALL PRODUCTS");
        app.showProductInEStore();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("FILTERED PRODUCTS");
        app.filterProductsInPriceRange(12000, 45000);

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        System.out.println("MAPPED PRODUCTS");
        app.showProductsToPrimeMember();

        System.out.println();
        System.out.println("==============================");
        System.out.println();

        List<Product> shoppingCart = app.products.subList(2, 6);
        User user = new User("John", shoppingCart);
```

```
<terminated> StreamsWithObjects [Java Application] /usr/eclipse/plugins/or
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Fridge  Hitachi 89500
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MacBook Apple  119500
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
LED TV  Samsung 49500
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Bottle  Hidrate 4500
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TOTAL 263000
```

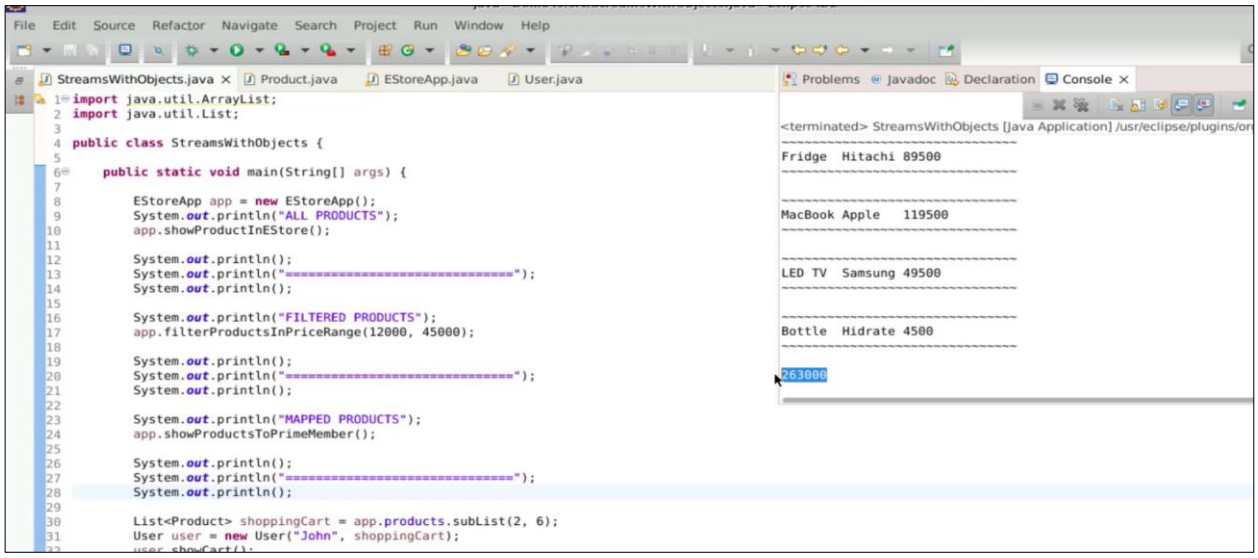
6.8 In the User class, instead of using forEach to display the products, you can utilize the stream. Write **cart.stream()** and on this stream, use the reduce function. The reduce function takes two products as input, **p1** and **p2**, and returns **p1.getPrice() + p2.getPrice()**

```
import java.util.List;

public class User {

    String name;
    List<Product> cart;
    int total;

    User(){

    }

    public User(String name, List<Product> cart) {
        this.name = name;
        this.cart = cart;
    }

    void showCart() {

        System.out.println(name+"'s CART");
        System.out.println("TOAL PRODUCTS: "+cart.size());
        /*cart.forEach((product)->{
            total += product.price;
            product.showProduct();
        });*/

        cart.forEach((product)->product.showProduct());

        cart.stream().reduce((p1, p2) -> p1.price+p2.price);

        System.out.println("TOTAL "+total);
```

6.9 Create a new product named p and set its price as **p1.getPrice() + p2.getPrice()**. Finally, return this product. You can try printing this directly, but since it returns an optional, use **get()** to retrieve the product and display its price.

```java
import java.util.List;

public class User {

    String name;
    List<Product> cart;
    int total;

    User(){

    }

    public User(String name, List<Product> cart) {
        this.name = name;
        this.cart = cart;
    }

    void showCart() {

        System.out.println(name+"'s CART");
        System.out.println("TOAL PRODUCTS: "+cart.size());
        /*cart.forEach((product)->{
            total += product.price;
            product.showProduct();
        });*/

        cart.forEach((product)->product.showProduct());

        System.out.println(cart.stream().reduce((p1, p2) -> {
            Product p = new Product();
            p.price += p1.price+p2.price;
            return p;
        }).get().price);

        //System.out.println("TOTAL "+total);
    }

}
```

6.10 Go to the **StreamWithObjects** class and run the code. Now you will see the same total amount displayed. By using the reduce function, you can reduce the entire list of product prices into a single price.




By following the above steps, you have successfully implemented the streams with user-defined objects to efficiently process and manipulate collections of custom data types in a functional programming style.