Lesson 06 Demo 08

Implementing Comparator with Lambda Expressions

Objective: To implement a comparator with lambda expressions in Java to simplify sorting logic and make the code more concise

Tools Required: Eclipse IDE

Prerequisites: None

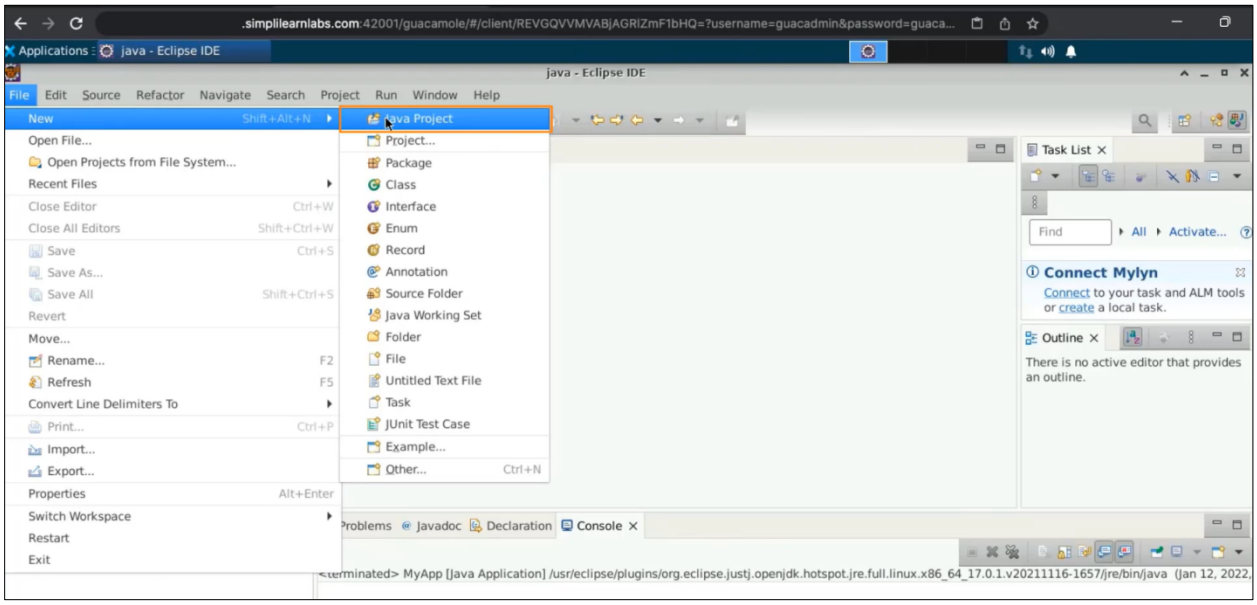
Steps to be followed:

1. Create a new java project
2. Generate constructors with the fields and create the default constructor
3. Create a class and a method that returns an ArrayList
4. Add product objects, with sample data
5. Use the sort method that takes a list as input
6. Create a comparator object and execute the code
7. Write an anonymous class, or a Lambda expression

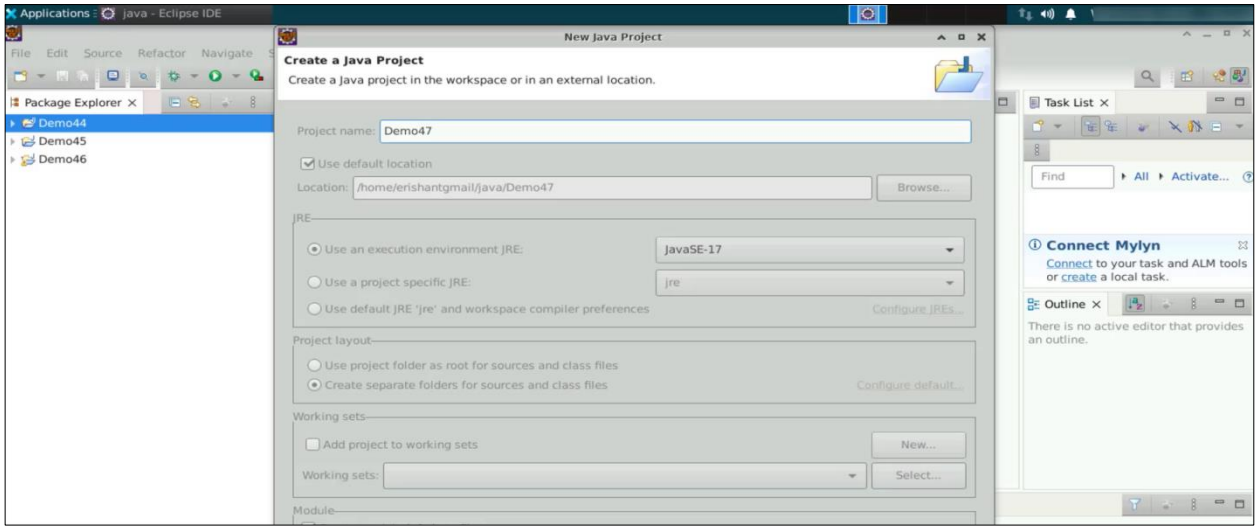
Step 1: Create a new java project

1.1 Open the **Eclipse IDE**

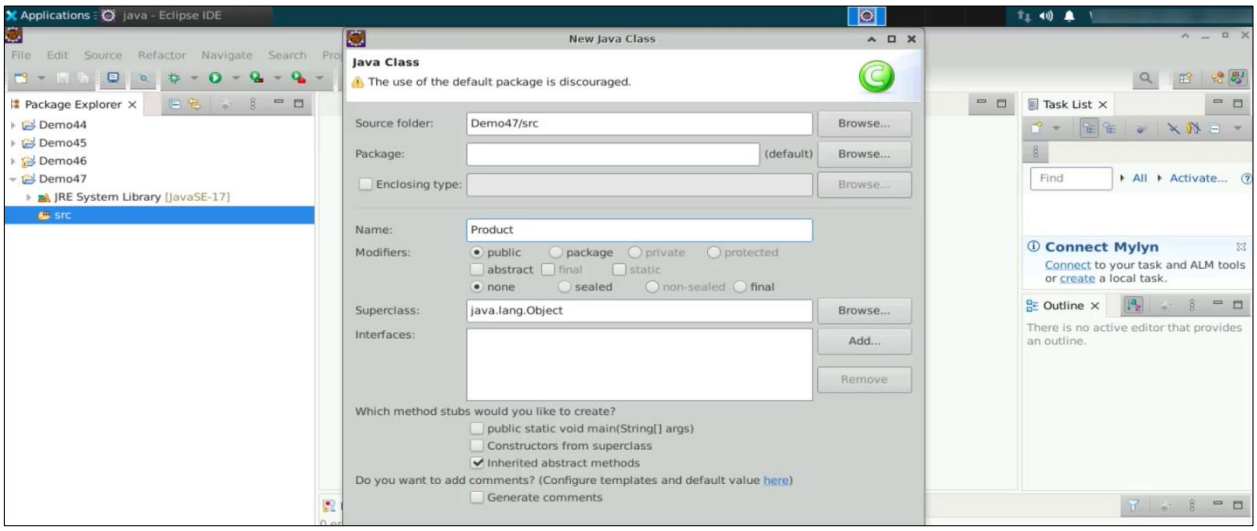
1.2 Select **File**, then **New,** and click **Java project**



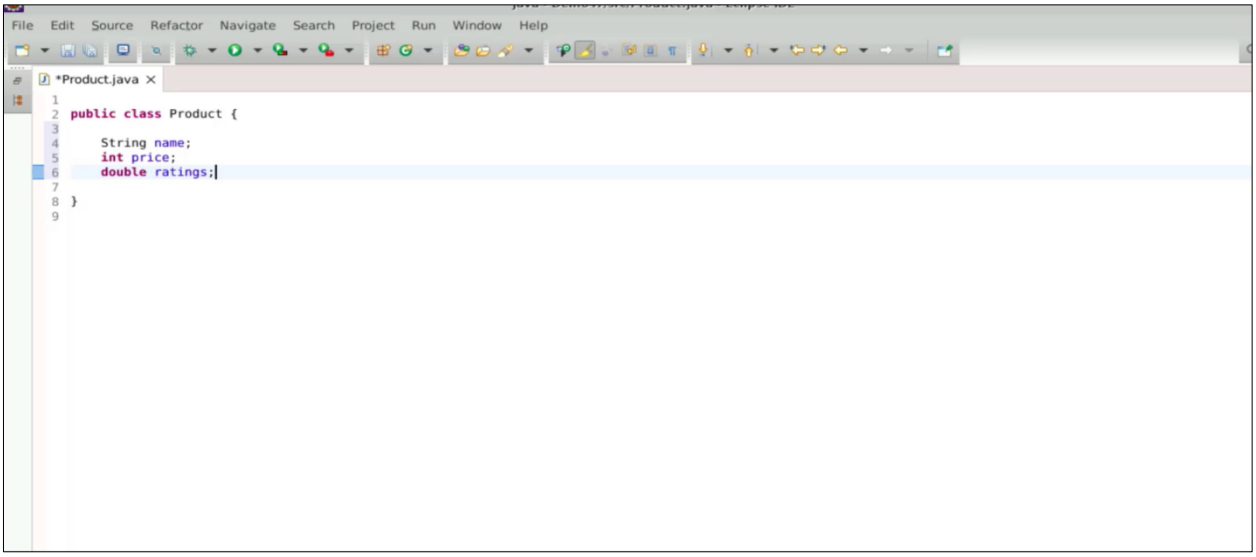
1.3 Enter the project name **Demo47,** uncheck **Create a module-info.java file**, and click **Finish**



1.4 Right-click on the **Demo47** project in the **src** folder and create a new class. Name this class **Product**. Select the main method and click **Finish**

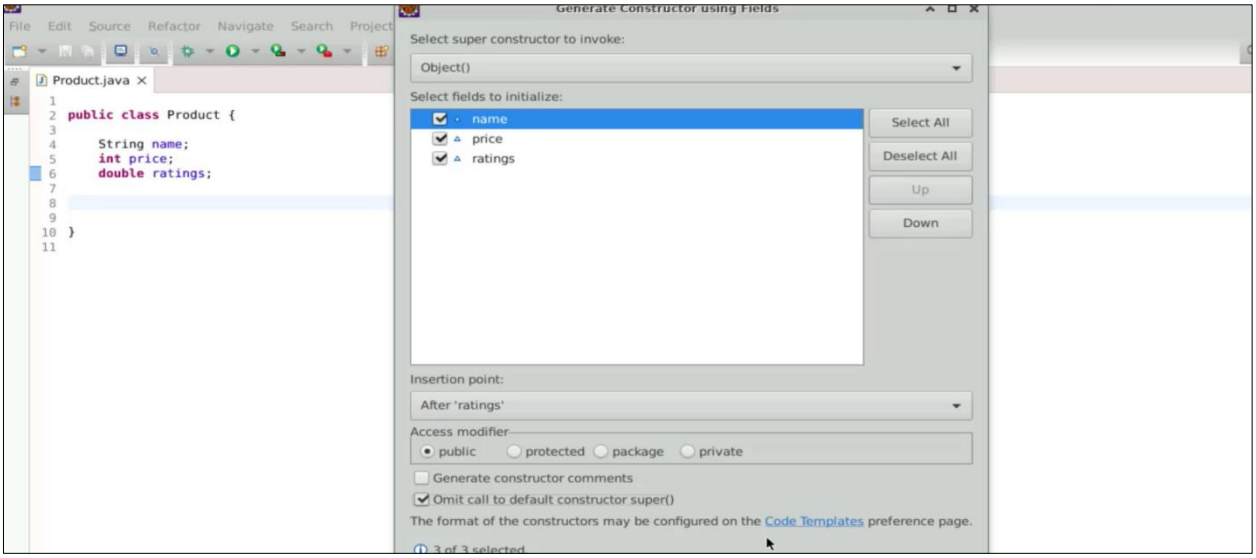


1.5 In the Product class, you will create variables for the product **name**, **price**, and **ratings**

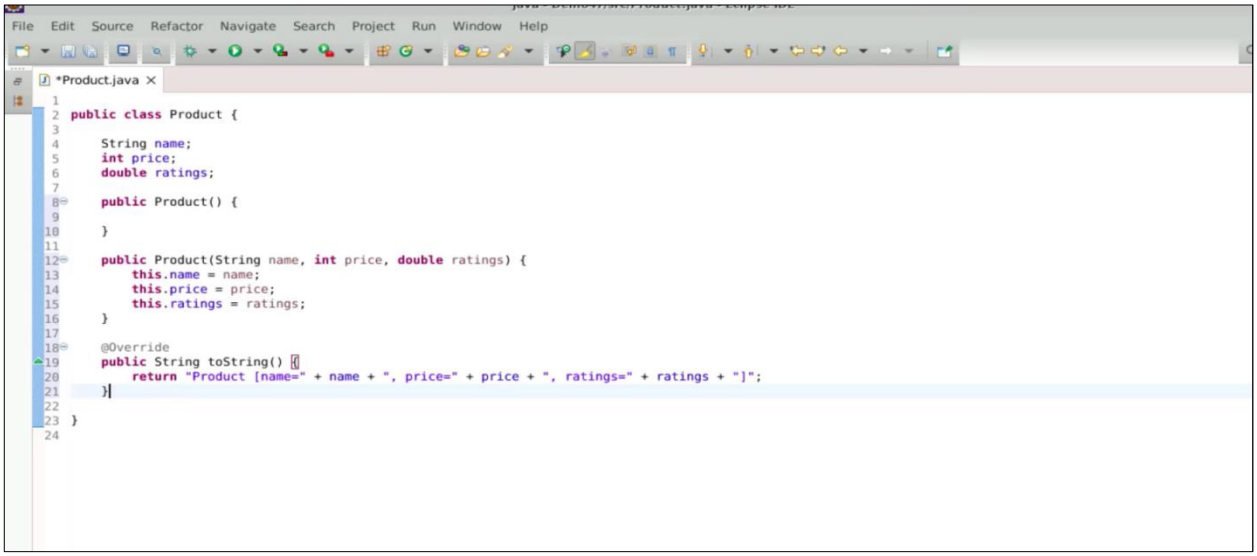


Step 2: Generate constructors with the fields and create the default constructor

2.1 Now, generate constructors with the fields and create the default constructor for the Product class

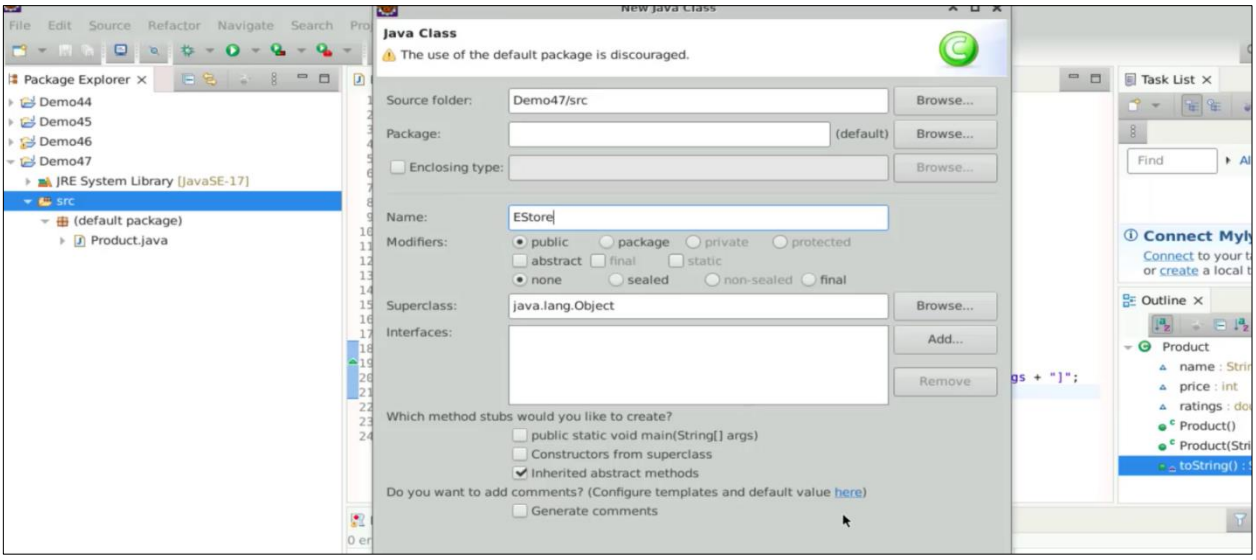


2.2 After completing the above step, navigate to the source code and select **Generate toString()** method. The **toString()** method is used to display the data stored in your Product object. This method defines the structure of your Product object

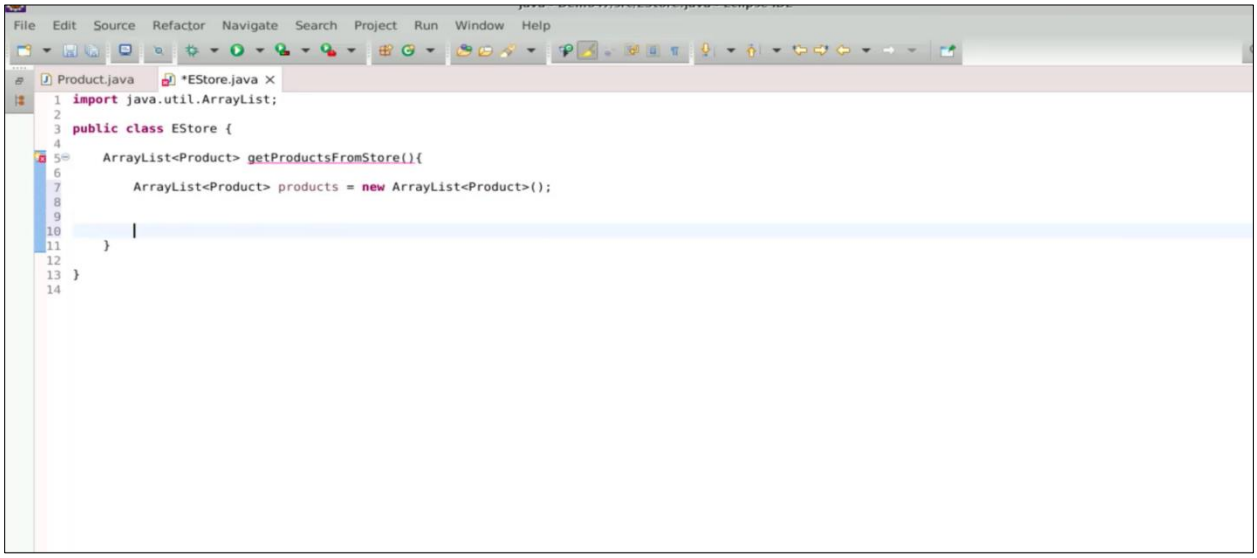


Step 3: Create a class and a method that returns an ArrayList

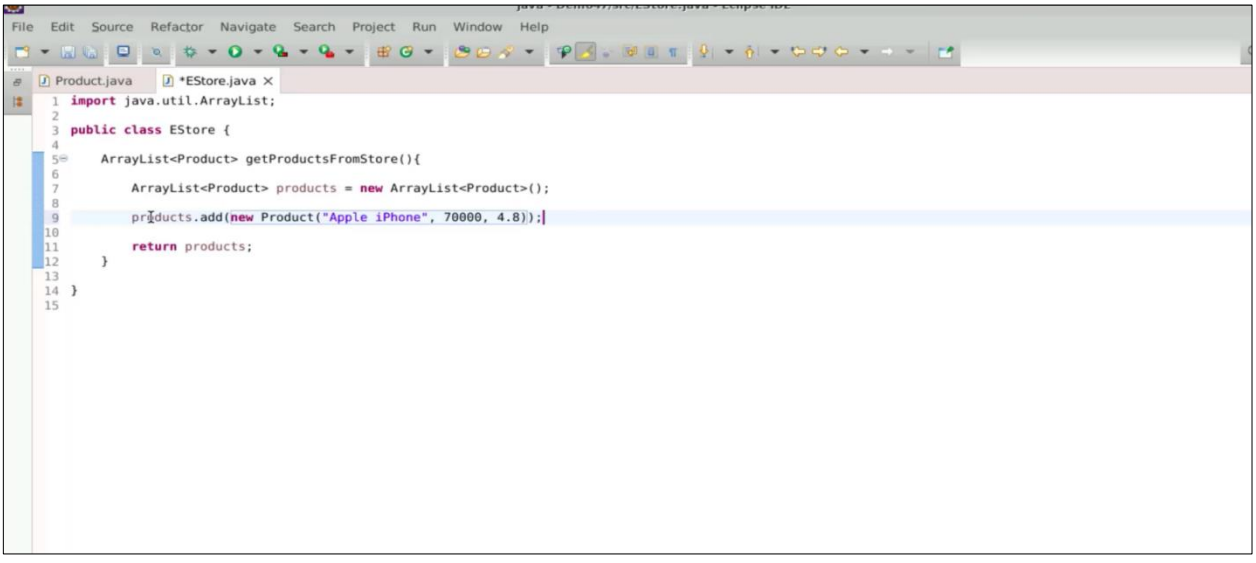
3.1 Create another class with name **EStore**



3.2 Within the **EStore** class, create a method that returns an **ArrayList**. This **ArrayList** will be of type **Product**. Name this method **getProductsFromStore**. This method will retrieve an **ArrayList** of Product objects. Initialize this ArrayList as a new instance

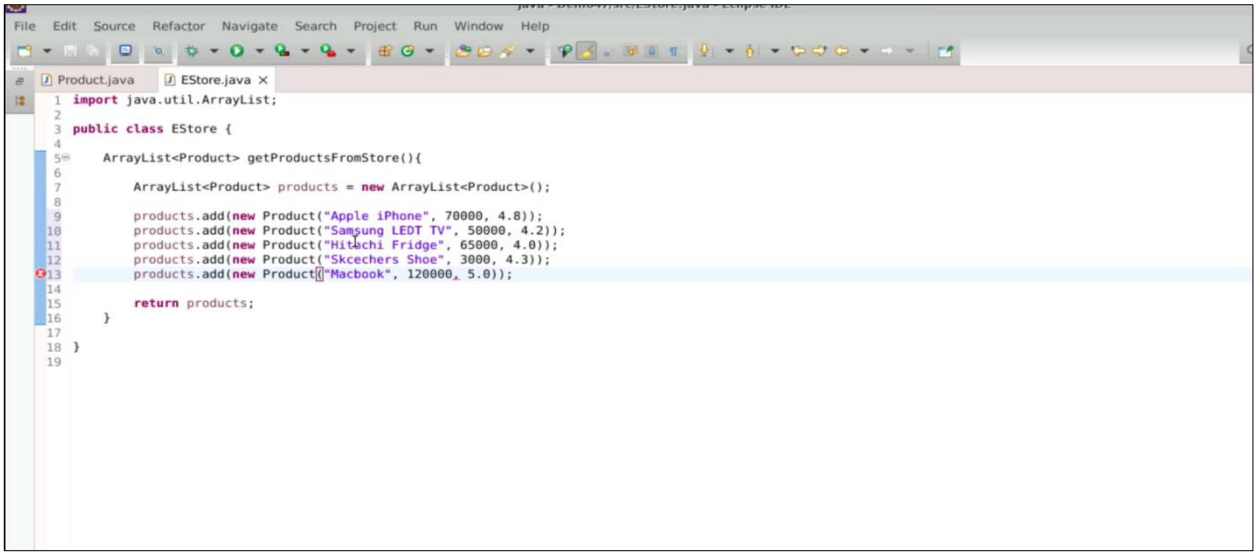


3.3 This ArrayList will be returned by the **getProductsFromStore** method. To populate the ArrayList, add a few Product objects directly. For example, you can use the syntax **products.add(new Product("Apple iPhone", 70000, 4.8));** This adds a new Product object with the name **Apple iPhone**," price of **70000,** and ratings of **4.8**.

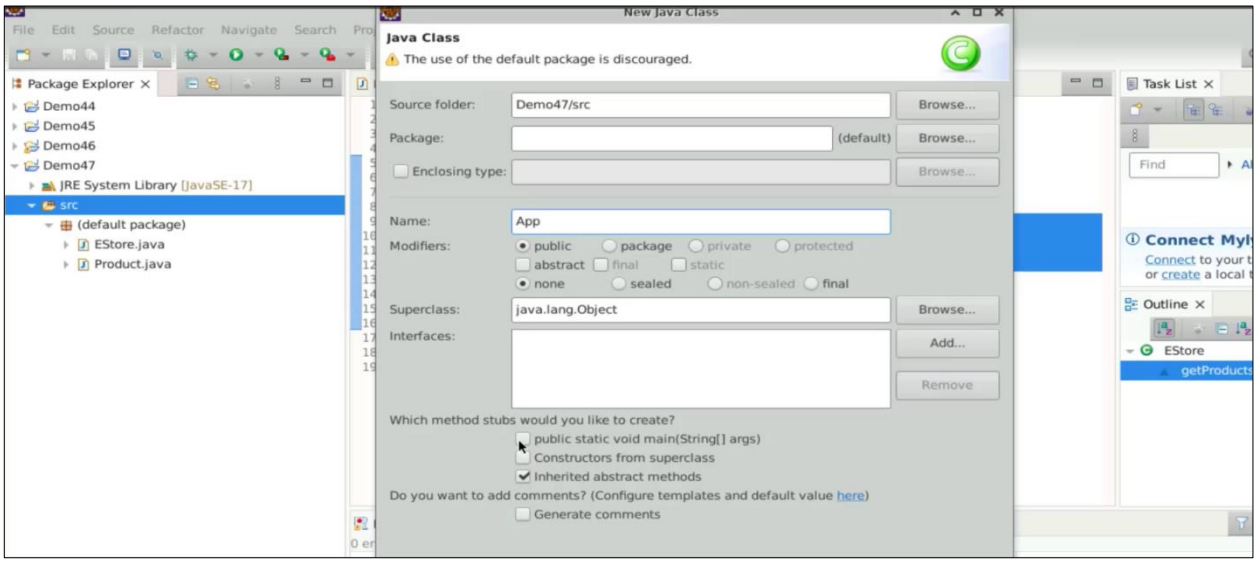


Step 4: Add product objects, with sample data

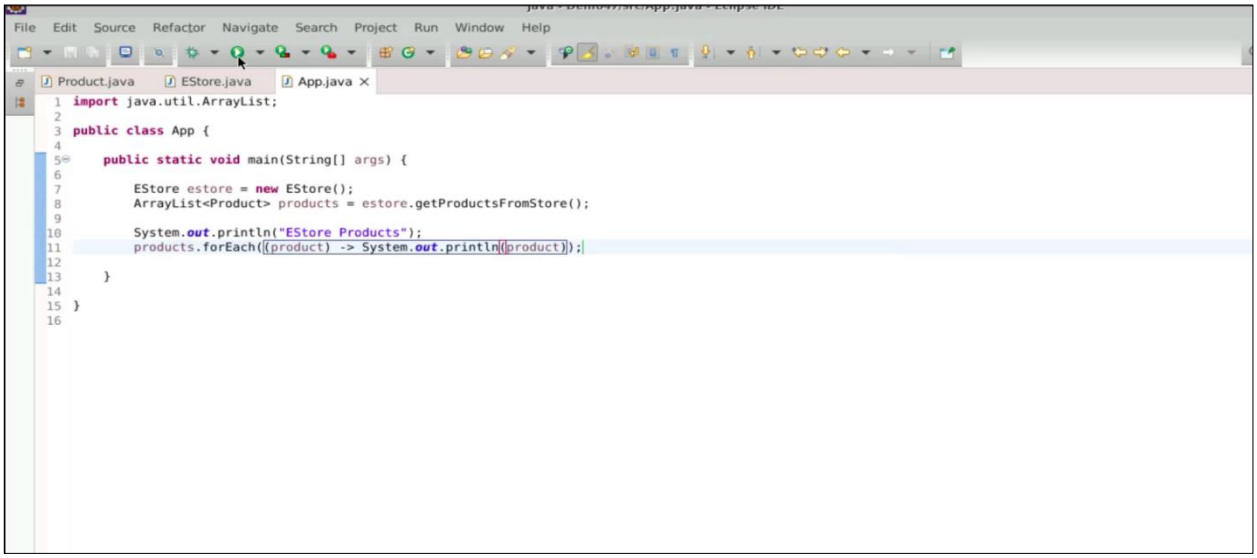
4.1 Similarly, you will add a few more product objects. For example, add the following products: **Samsung LED TV**, **Hitachi Fridge**, **Skechers**, **and a MacBook**. You can modify the prices and ratings for these products as needed



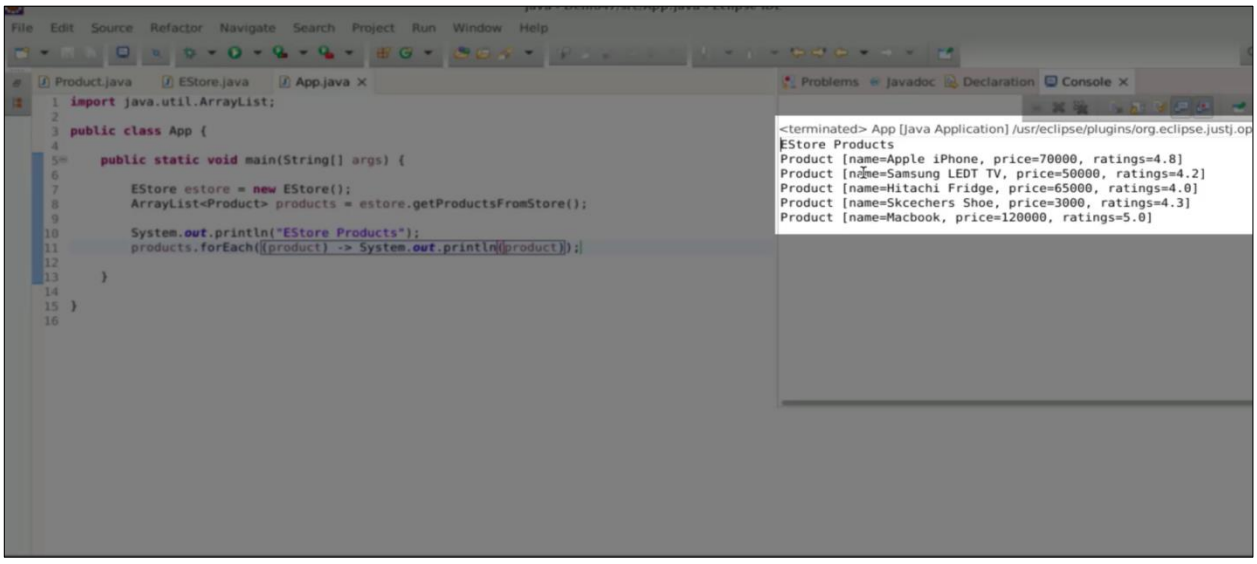
4.2 Create another new Java file as a new class, name this as **App** with the main method



4.3 Within the **App** class, retrieve the ArrayList of products from your EStore. Create a new instance of the **EStore** class, Then, use the **getProductsFromStore** method to obtain the ArrayList of products

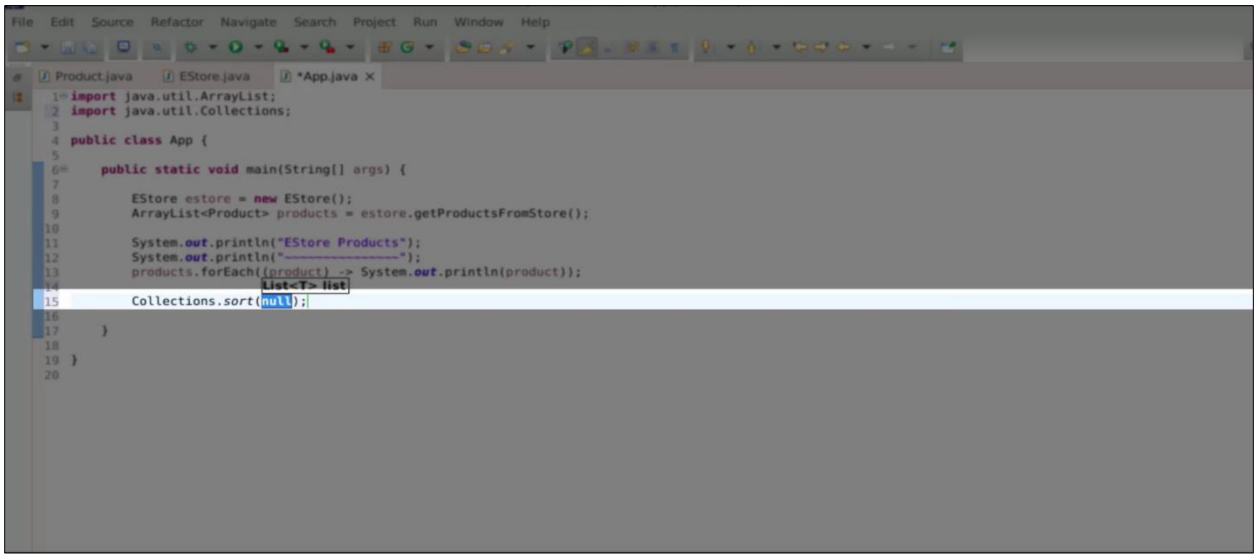


4.4 Iterate through the product ArrayList and print each product using a for-each loop with a Lambda expression. These are the products available in the **EStore,** displayed exactly as you added them. These are the default products listed in the **EStore**.

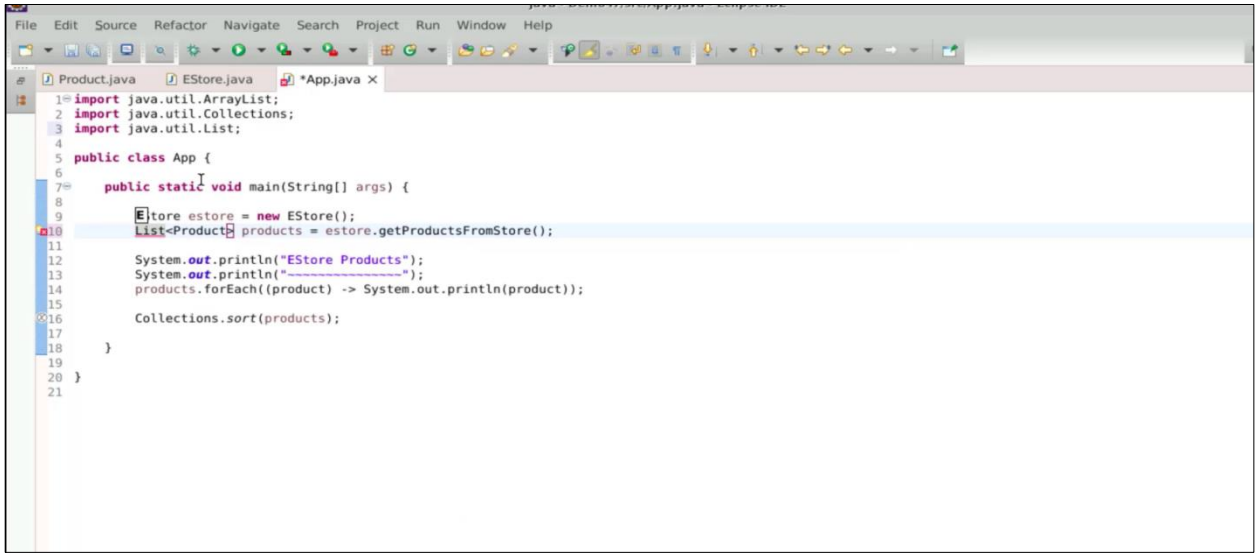


Step 5: Use the sort method that takes a list as input

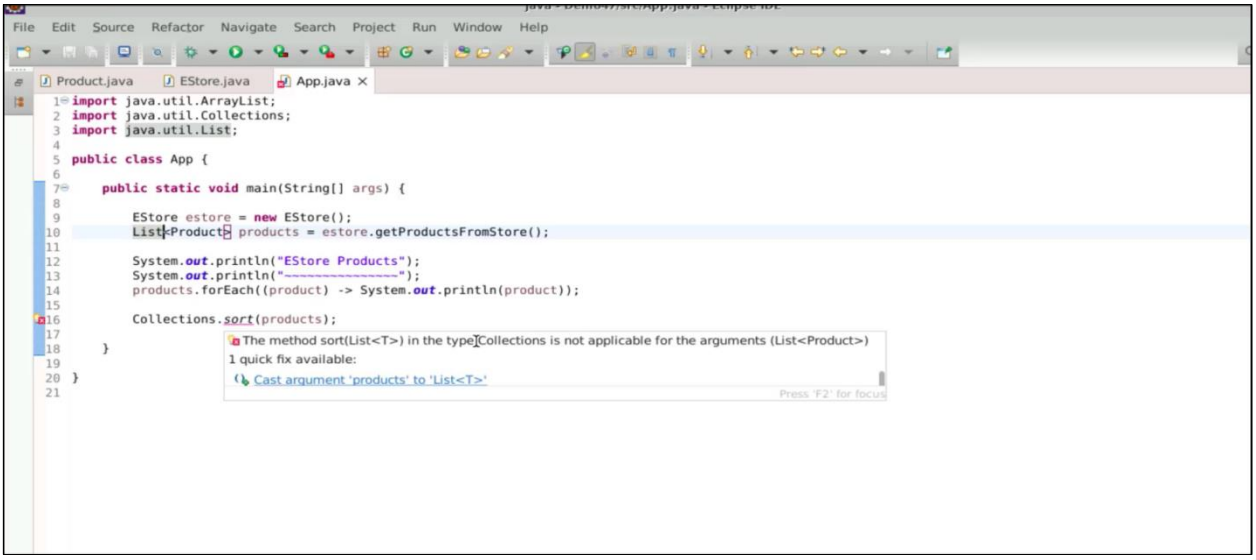
5.1 Now, let us say you want to sort the products. To achieve this, you can use the sort method provided by the Collections class. This **sort** method takes a List as input



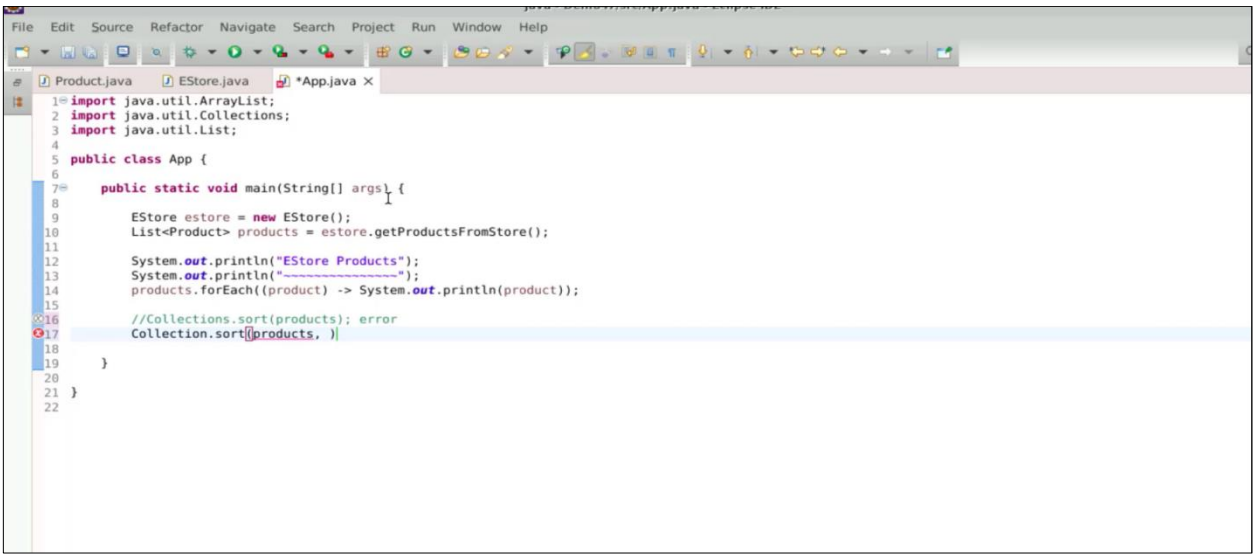
5.2 When you attempt to pass the products list as input, you might encounter an error. To resolve this, ensure that the products list is of type List.



5.3 Despite changing the list type, the **sort** method might still show an error. The reason for this is that **Collections.sort** is not applicable for a List of type Product.

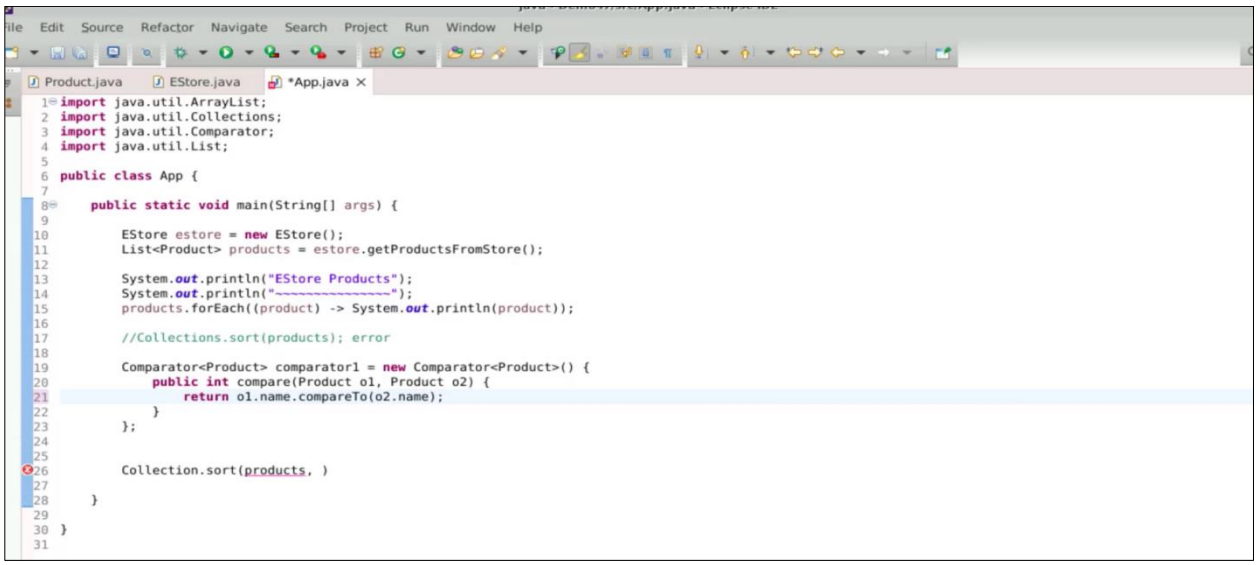


5.4 To overcome this issue, you need to use a comparator. Add the appropriate comparator object as an argument to the **Collections.sort** method, along with the products list.

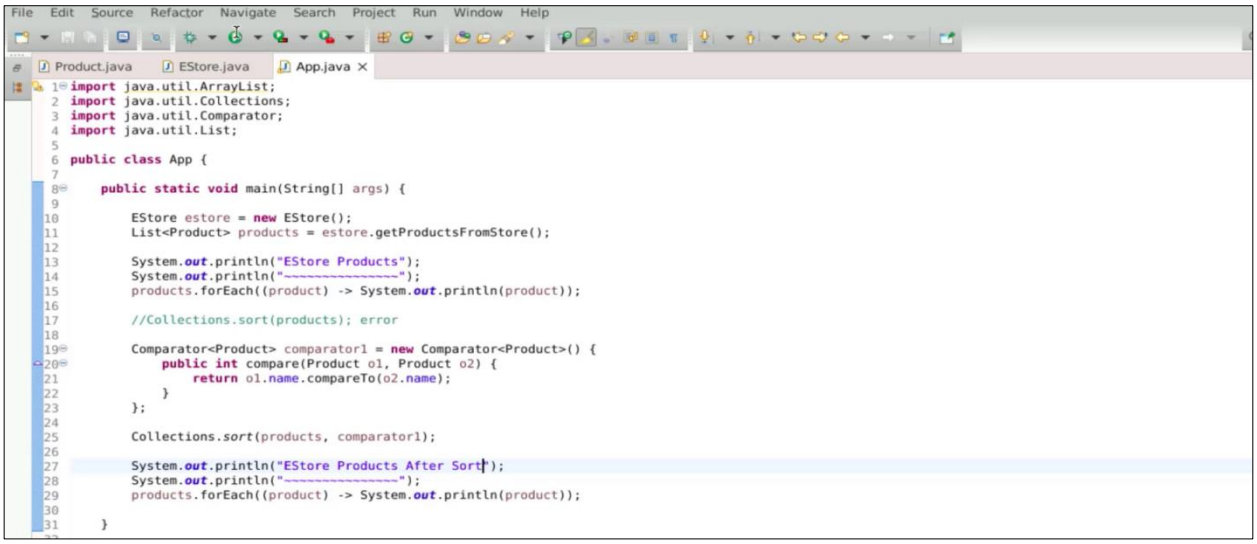


Step 6: Create a comparator object and execute the code

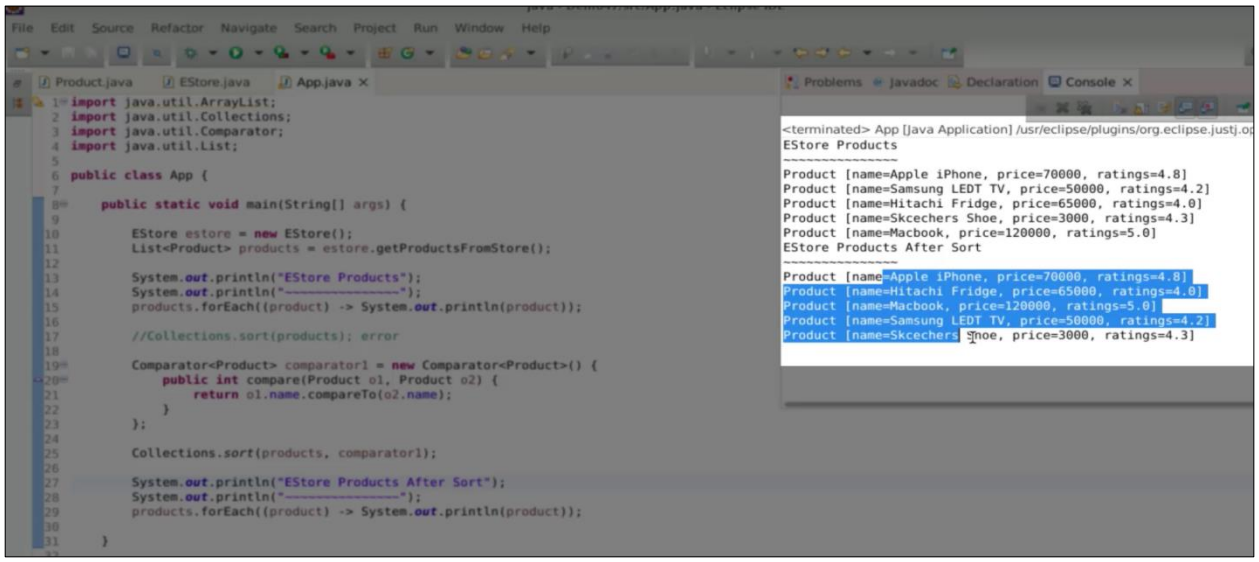
6.1 Before proceeding, let's create a comparator object. We will use the **Comparator** class from **java.util** with the type **Product**. Create a new comparator, referred to as **comparator01**, as a new instance of Comparator. In the anonymous class for comparator01, override the **compare** method to compare two **Product** objects. For sorting based on the name, use **return o1.getName().compareTo(o2.getName());**



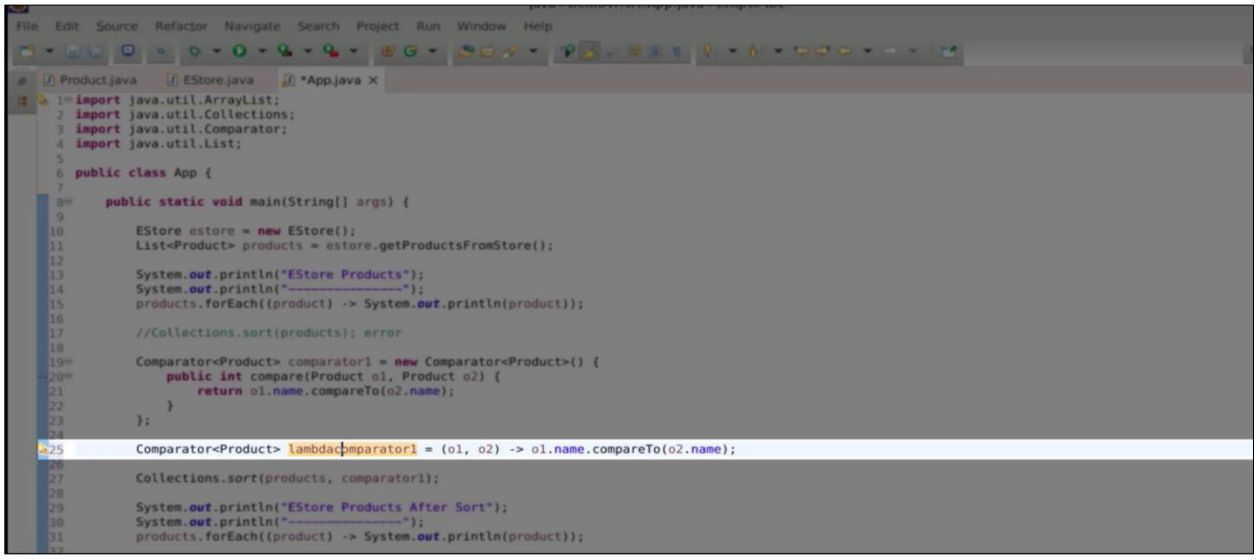
6.2 Now, pass **comparator01** as an argument to **Collections.sort**. You will notice that the error is resolved. Use a loop to iterate through the sorted list and print the data. Display **EStore products after sorting** to indicate the output.



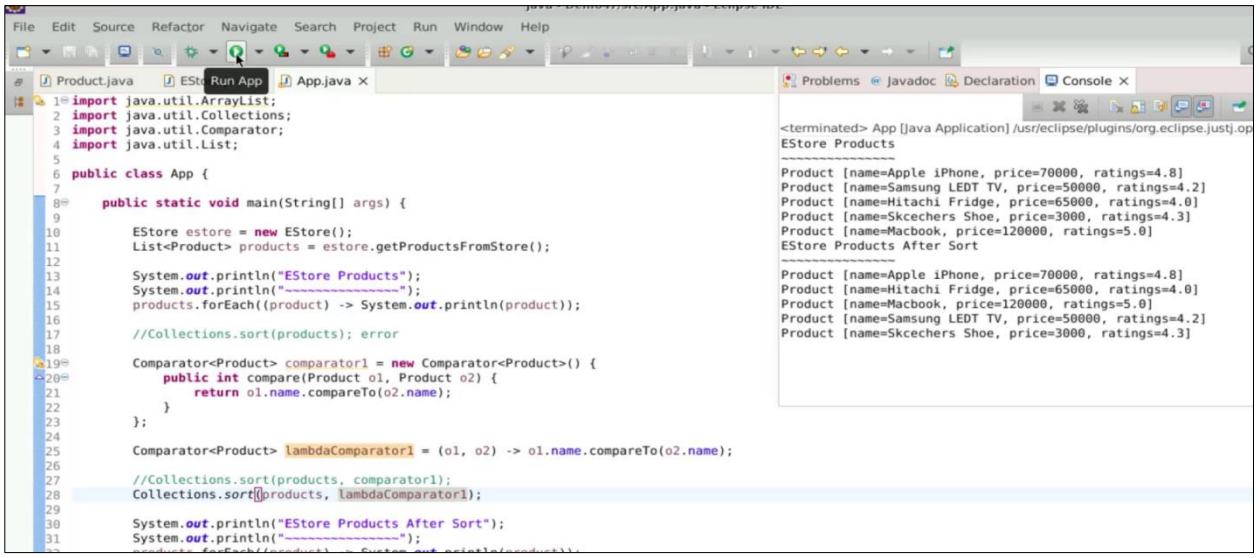
6.3 When you run the program, you will see that the data is sorted based on the names. The products are displayed in the order: Apple, Hitachi, MacBook, Samsung, and Skechers.
Note that products starting with **Sa** come before those starting with **SK**



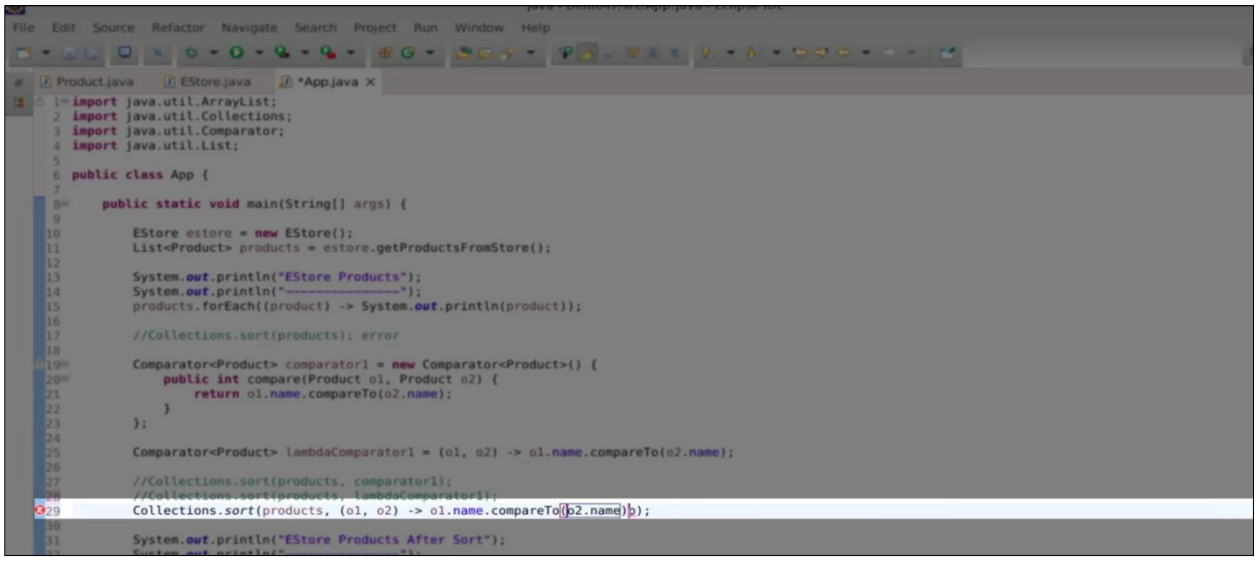
6.4 If you prefer, you can use a lambda expression to define the comparator instead of the anonymous class. Replace **comparator01** with a lambda expression that takes two objects, **o1** and **o2**, and returns **o1.getName().compareTo(o2.getName()).** You can refer to this as **lambdaComparator**



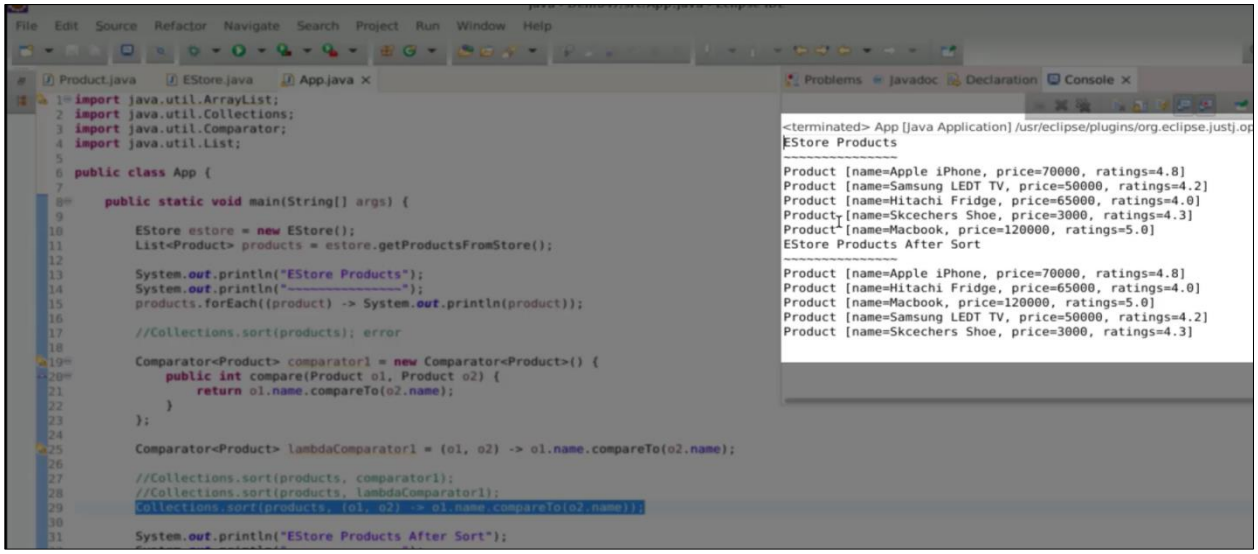
6.5 Even if you pass **lambdaComparator,** the output will be the same. Running the program will yield a sorted list based on names. This approach simplifies the comparator definition by using a lambda expression



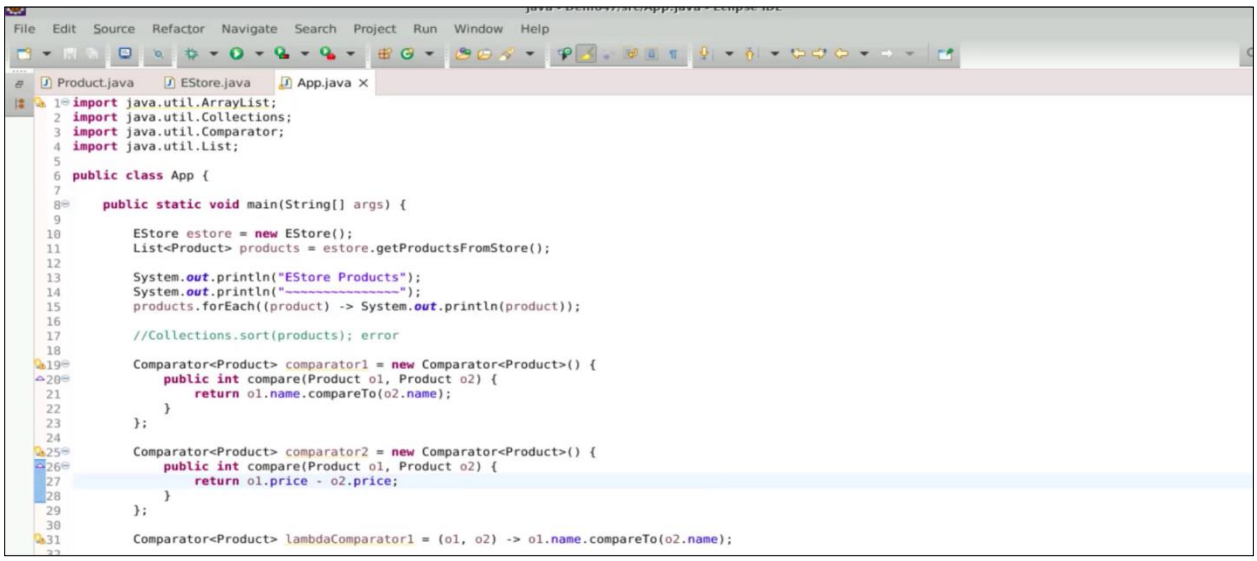
6.6 Alternatively, you can further simplify the code by directly using the lambda expression in the **Collections.sort** method. Replace the entire comparator definition with **Collections.sort(products, (o1, o2)** -> **o1.name.compareTo(o2.name()));**



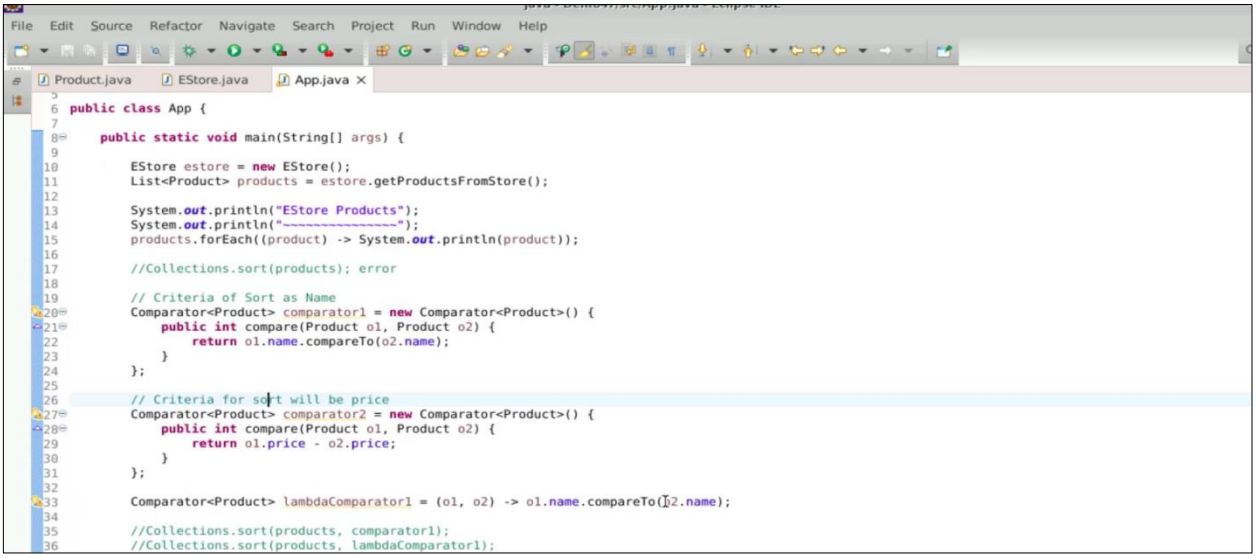
6.7 Run the code, as this is a single line of code with which you can sort your product, so you can see the same output



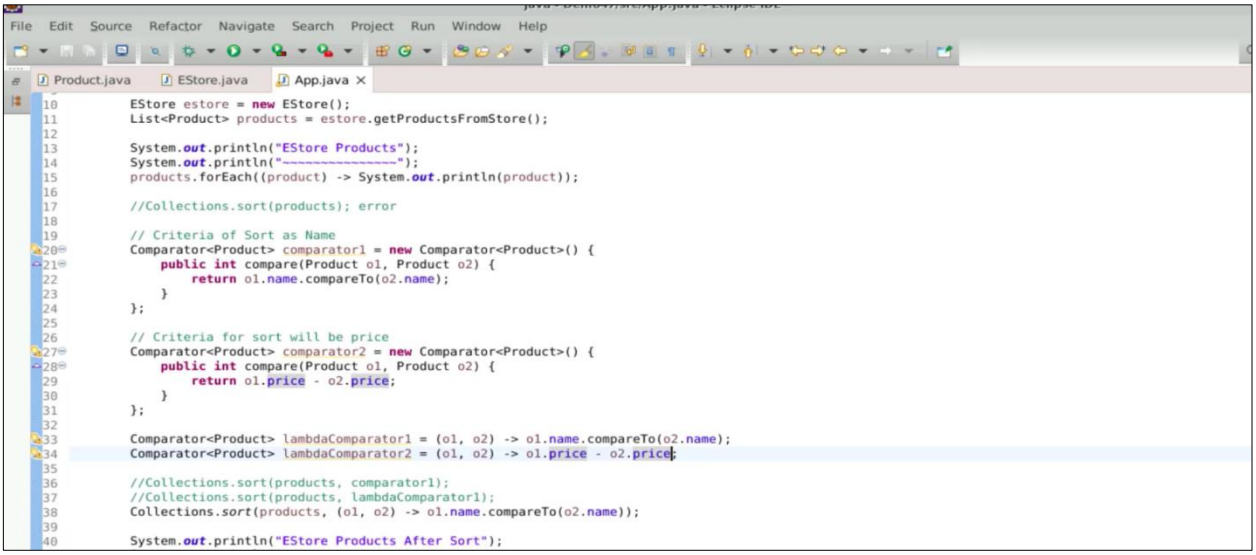
6.8 If you want to sort based on the price, create another comparator called **comparatorTwo**. In this comparator, return **o1.getPrice() - o2.getPrice().** This means you are comparing and returning an integer from the **compare** method for the two objects.



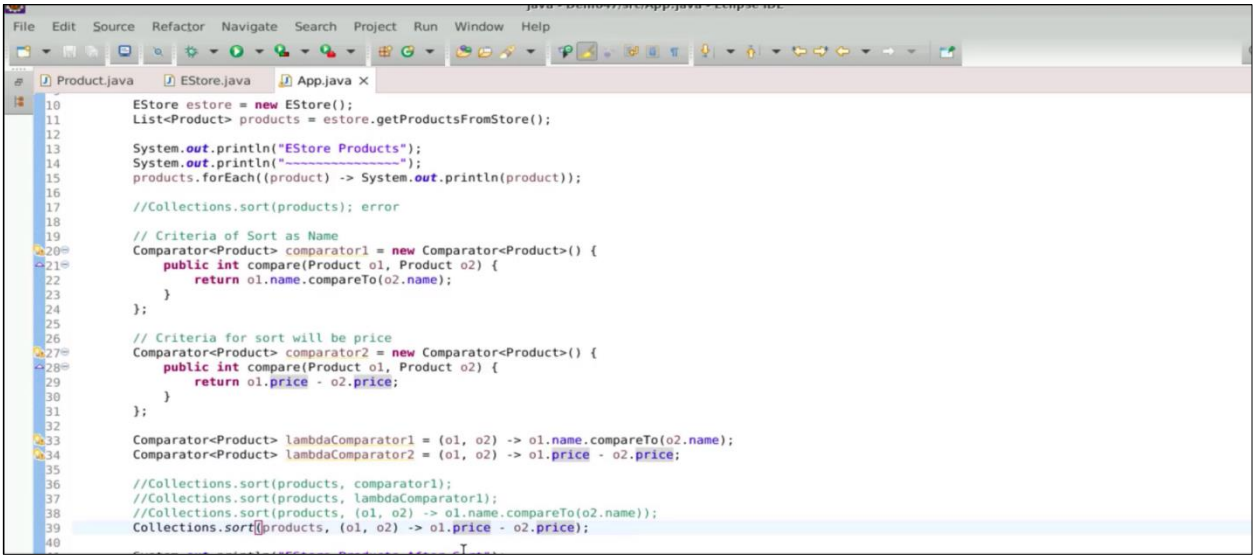
6.9 Now you have **comparator2** which works on the criteria of sorting. The criteria to sort will be price, and here on **comparator1** you got criteria of sort as a name



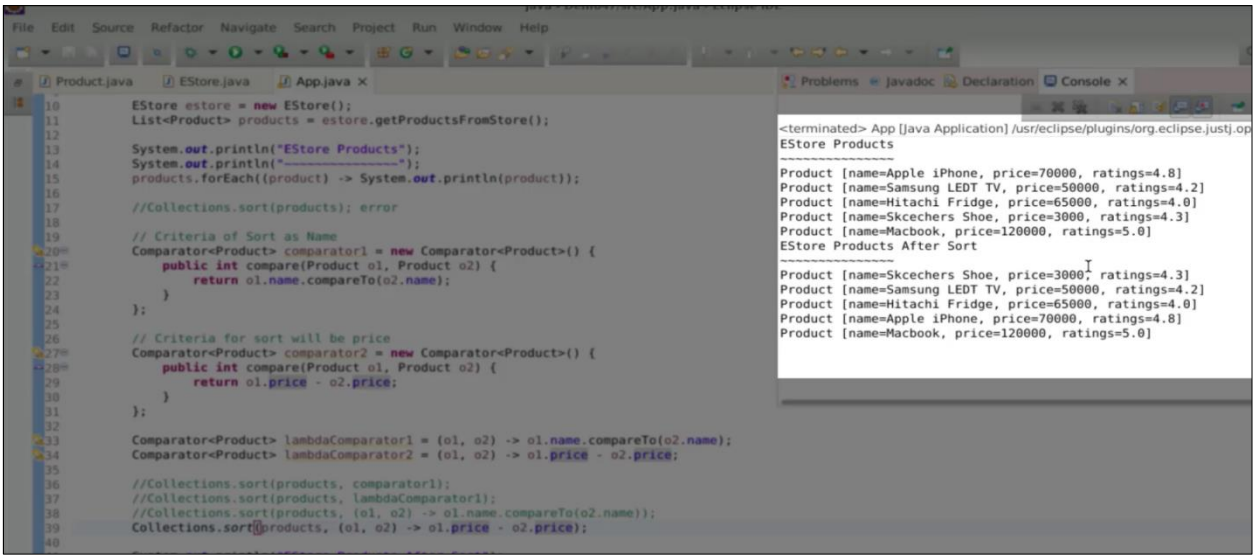
6.10 Similarly, you can create a lambda expression for **comparator2** by using **LambdaComparator2**. Replace the definition with **(o1, o2) -> o1.getPrice() – o2.getPrice()**



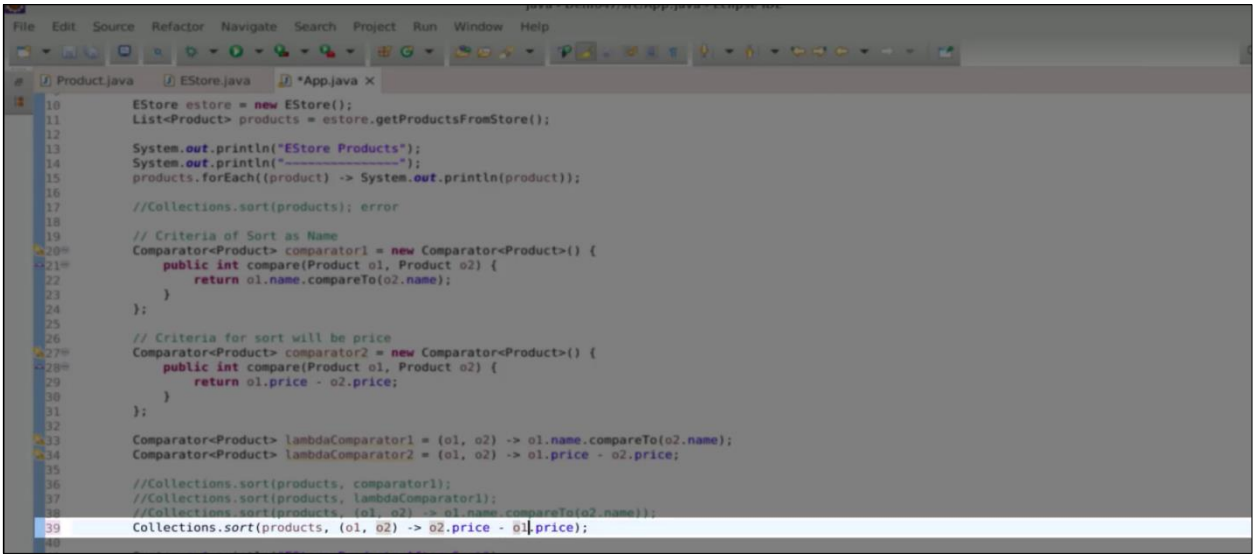
6.11 Next, let us write a more sophisticated version to sort the products based on price. Replace the lambda expression with **o1.getPrice() - o2.getPrice()**



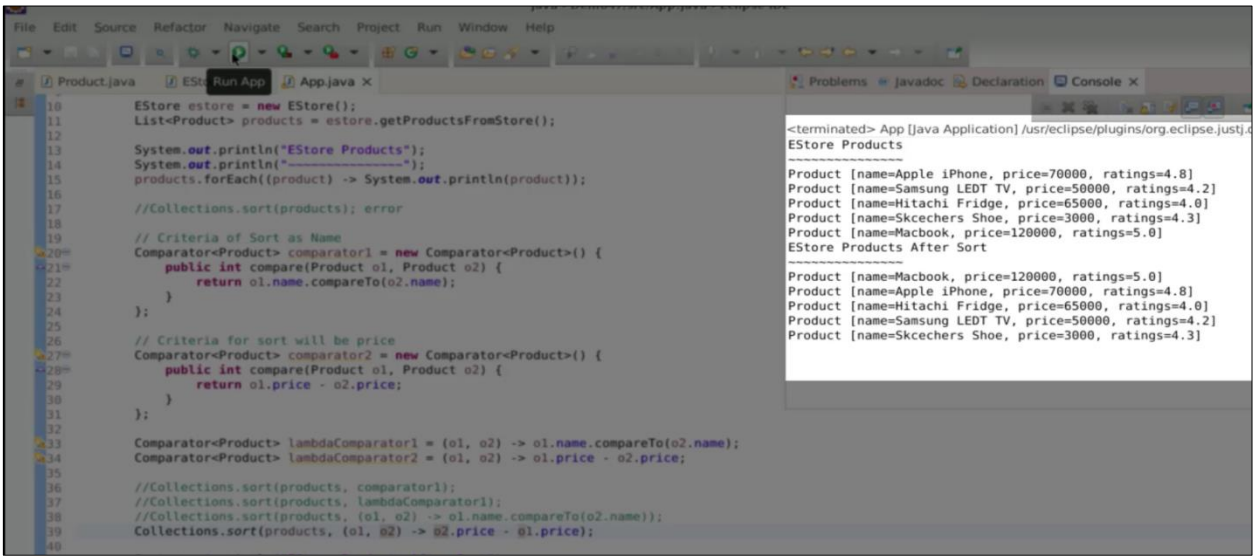
6.12 Running the program, you will observe that the product with the lowest price is at the top. The sorting is done in ascending order.



6.13 If you return **o2.getPrice() - o1.getPrice()** instead, the sorting will be in descending order

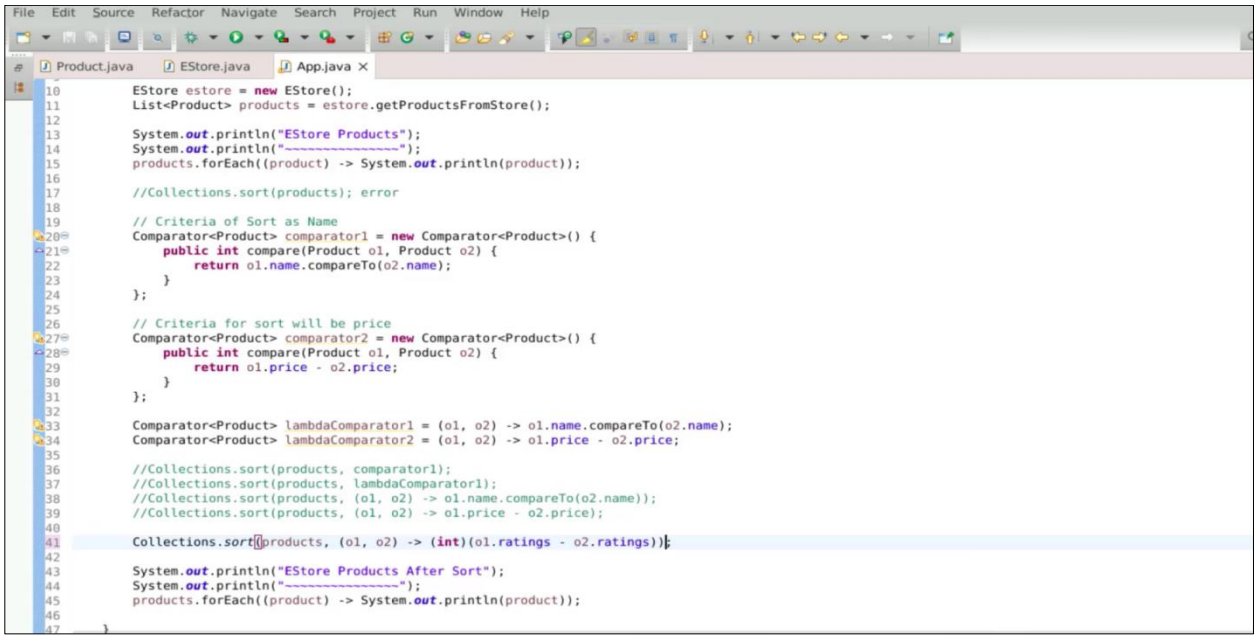


6.14 Running the code again, you will see that the sorting is done in descending order. The sequence of objects is reversed.

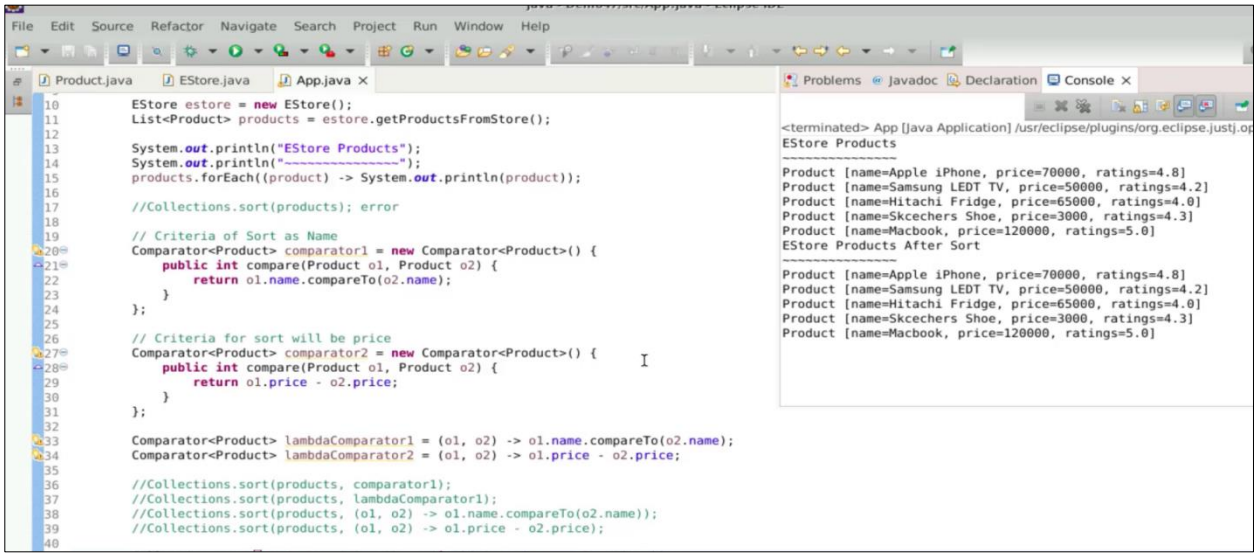


Step 7: Write an anonymous class, or a Lambda expression

7.1 Now, to sort the products based on ratings, modify the code to use **ratings** instead of **price** or **name.** However, when subtracting the ratings, you need to cast the result as an integer.



7.2 Run the code again. Now, you will see the data sorted based on the ratings of the products



By following these steps, you have successfully implemented a comparator with lambda expressions in Java to simplify sorting logic and make the code more concise.