A way to visualize Java Streams

Damian-Teodor Beleș

10 November, 2020

Context

As a support for a visual description of streams, the following exercise will be implemented:

Create a Product class that has as attributes the following:

- amount
- currentPrice
- previousPrices (Map<String, ArrayList<Double>> with keys:"recommendedPrices" and "actualPrices")
- name

Using streams, solve the following tasks:

1. Validate the currentPrice, amount and previousPrices to be positive numbers and the recommendedPrices and actualPrices to exist and have same lengths
2. Create a list of products and print it
3. Extract a list with products that are over a specified price
4. Print products on discount (below their average price over time)
5. Print the products that have the most changes of price over time
6. Print the total price of the list

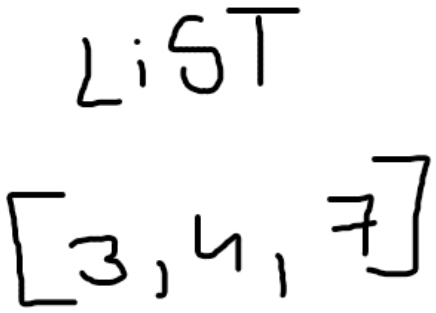
As a short notice, I would like to say that only three structures are presented here, for more examples of how to use streams check the code.

Notation

For understanding this paper one should get used to the notations used.

List Notation

In the following picture one can see a list with elements 3, 4 and 7:



ListNotation.png

Stream Notation

In the following picture one can see a stream with elements 3, 4 and 7:

StreamNotation.png

Products List

We're working with the following products list:

name: Wireless Mouse
currentPrice: 30.99
amount: 40
previousPrices:
"recommendedPrices": [31.59, 31.59, 30.0]
"actualPrices": [31.99, 31.59, 30.59]

name: Wired Keyboard
currentPrice: 50.99
amount: 20
previousPrices:
"recommendedPrices": [59.0, 59.49, 55.0]
"actualPrices": [59.49, 59.99, 55.49]

name: Phone Screen Protector
currentPrice: 7.0
amount: 800
previousPrices:
"recommendedPrices": [5.0, 6.49, 5.49, 4.0, 5.0]
"actualPrices": [5.49, 6.99, 5.49, 4.49, 5.49]

name: PC Case
currentPrice: 30.99
amount: 40
previousPrices:
"recommendedPrices": [31.99, 31.59]
"actualPrices": [31.49, 31.49]

name: Headphones
currentPrice: 115.0
amount: 15
previousPrices:
"recommendedPrices": [110.0, 105.0, 120.0]
"actualPrices": [110.0, 105.0, 120.0]

name: Phone Case
currentPrice: 7.49
amount: 1000
previousPrices:
"recommendedPrices": [6.0, 7.49, 6.49, 4.0, 6.0]
"actualPrices": [6.49, 7.99, 6.49, 4.49, 6.49]

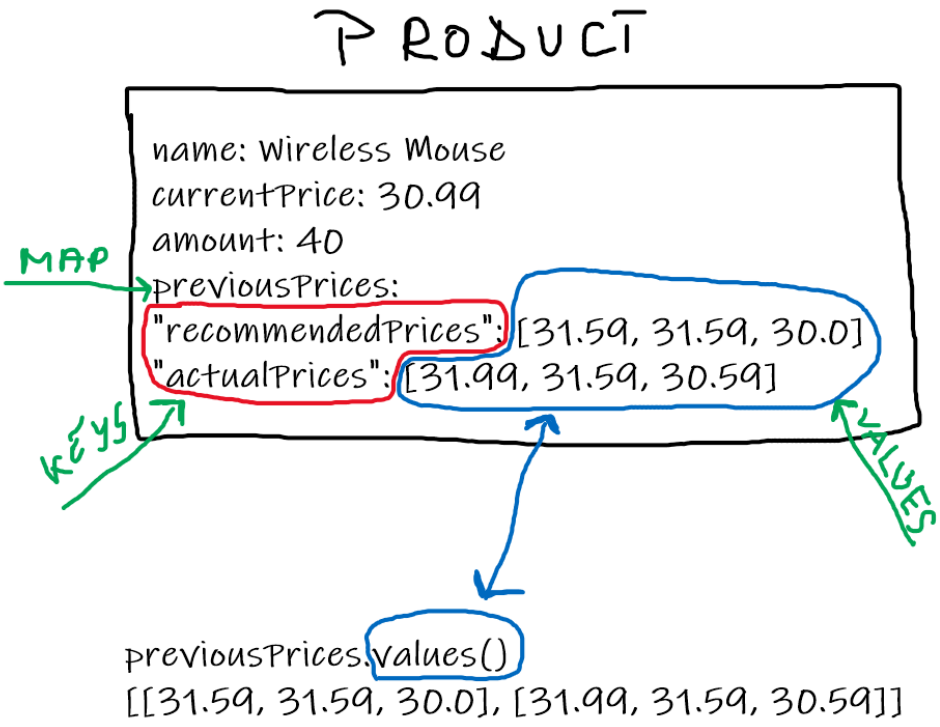
ObjectList.png

```
new Product(40, 30.99, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(31.99, 31.59, 30.59)));
        put("recommendedPrices", new ArrayList<Double>(List.of(31.59, 31.59, 30.0)));
}}, "Wireless Mouse"),
new Product(20, 50.99, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(59.49, 59.99, 55.49)));
        put("recommendedPrices", new ArrayList<Double>(List.of(59.0, 59.49, 55.0)));
}}, "Wired Keyboard"),
new Product(40, 30.99, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(31.99, 31.59)));
        put("recommendedPrices", new ArrayList<Double>(List.of(31.49, 31.49)));
}}, "PC Case"),
new Product(15, 115.0, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(110.0, 105.0, 120.0)));
        put("recommendedPrices", new ArrayList<Double>(List.of(110.0, 105.0, 120.0)));
}}, "Headphones"),
new Product(1000, 7.49, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(6.49, 7.99, 6.49, 4.49, 6.49)));
        put("recommendedPrices", new ArrayList<Double>(List.of(6.0, 7.49, 6.49, 4.0, 6.0)));
}}, "Phone Case"),
new Product(800, 7.0, new HashMap<String, ArrayList<Double>>() {{
        put("actualPrices", new ArrayList<Double>(List.of(5.49, 6.99, 5.49, 4.49, 5.49)));
        put("recommendedPrices", new ArrayList<Double>(List.of(5.0, 6.49, 5.49, 4.0, 5.0)));
}}, "Phone Screen Protector");
```

Validation of prices

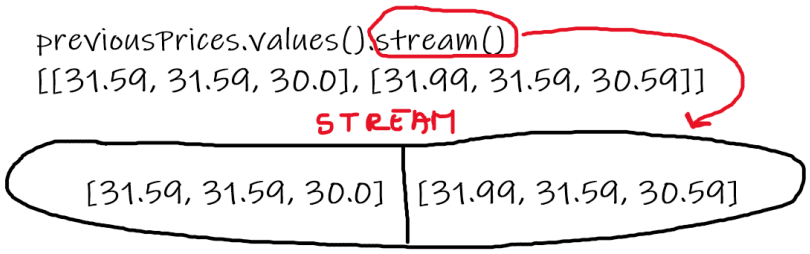
```
boolean previousPricesArePositive =

previousPrices.values()
              .stream()
              .flatMap(priceList -> priceList.stream())
              .allMatch(price -> price >= 0);

if (!previousPricesArePositive) {
        throw new ProductException(ProductException.INVALID_PRICE);
}
```

Let's suppose we want to validate the previous prices for the first object for the sake of the example.
previousPrices.values() gives us a list of values of the *previousPrices* map, as shown in the following picture:



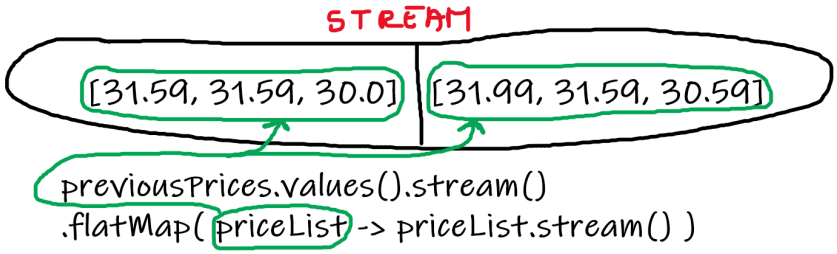
PreviousPricesValues.png

We then call *.stream*() in order to transform the list of values into a Java stream so we can further manipulate the data in the way we want it to.
We should be aware that a *stream* offers us already implemented methods unlike the *ArrayList*. The following transformation occurs:



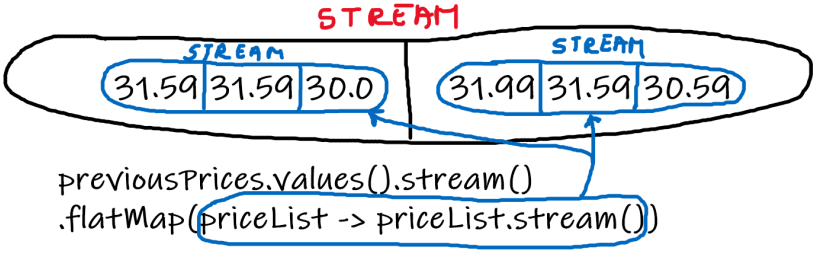
PreviousPricesStream.png

To further analyze the manipulation of the data, for the *.flatMap*() we'll construct the result from inside out.
Firstly, in the following picture, with green, when using a predicate I decided to name each element with *priceList* since that's what our current stream contains.



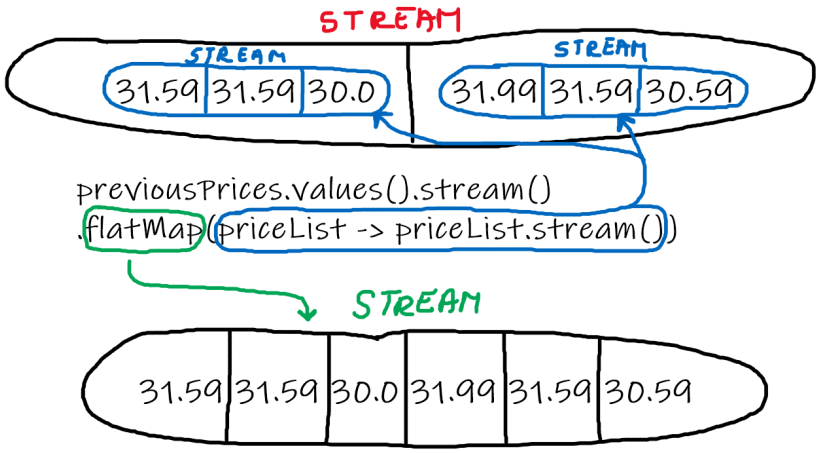
PreviousPricesVarNaming.png

After naming how each element of the stream should be referred as, the predicate will be evaluated, in this case, each list will become a stream, since we call *priceList.stream*():



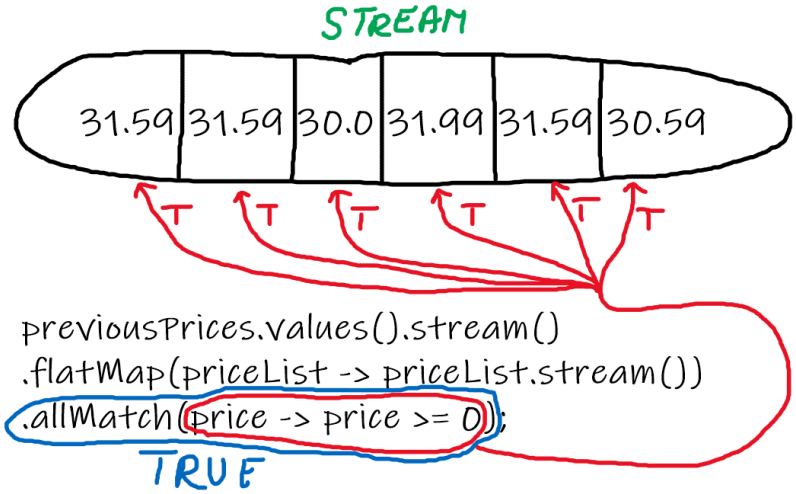
PreviousPricesStreamPredicateEvaluation.png

Finally we can now conclude what the effect of $.flatMap(priceList -> priceList.stream()$ will be. *flat* in *flatMap* stands for flattening, meaning that the elements inside a stream will become part of the bigger stream. For example, in the previous picture, the two blue stream will be merged into one:



PreviousPricesStreamAfterFlat.png

At the end, we conclude our stream journey with the $.allMatch()$ method, which returns true if the predicate returns true for each one of the elements of the stream:



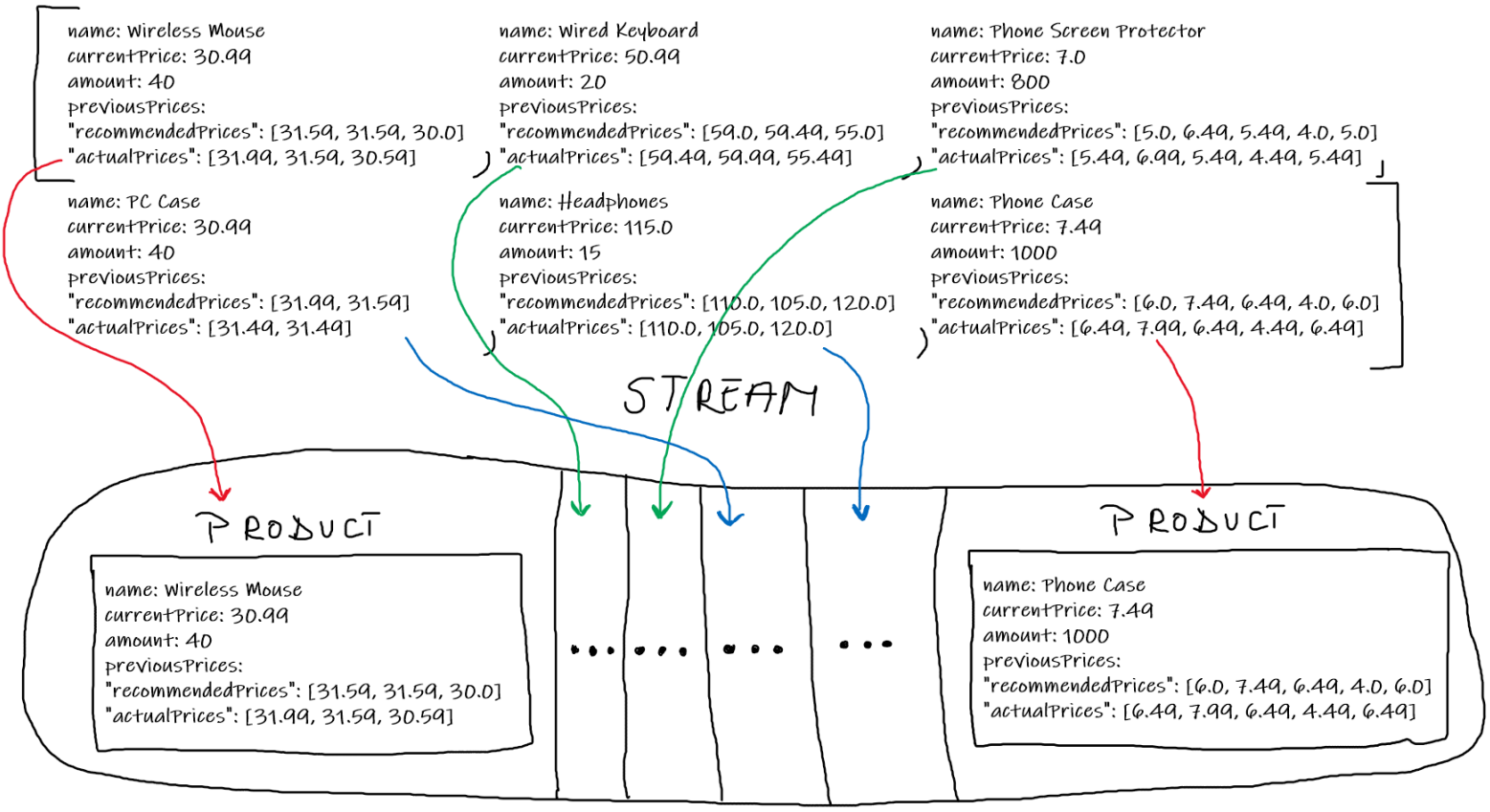
PreviousPricesStreamAllMatch.png

Here we can see that the predicate returns true for all the elements (see red branches, T = true), hence the overall $.allMatch()$ returns true, the value saved in *previousPricesArePositive* in our code.

Printing the products ArrayList

```
private static void printProductArrayList(ArrayList<Product> products) {
        products.stream().forEach(product -> System.out.println(product));
}
```

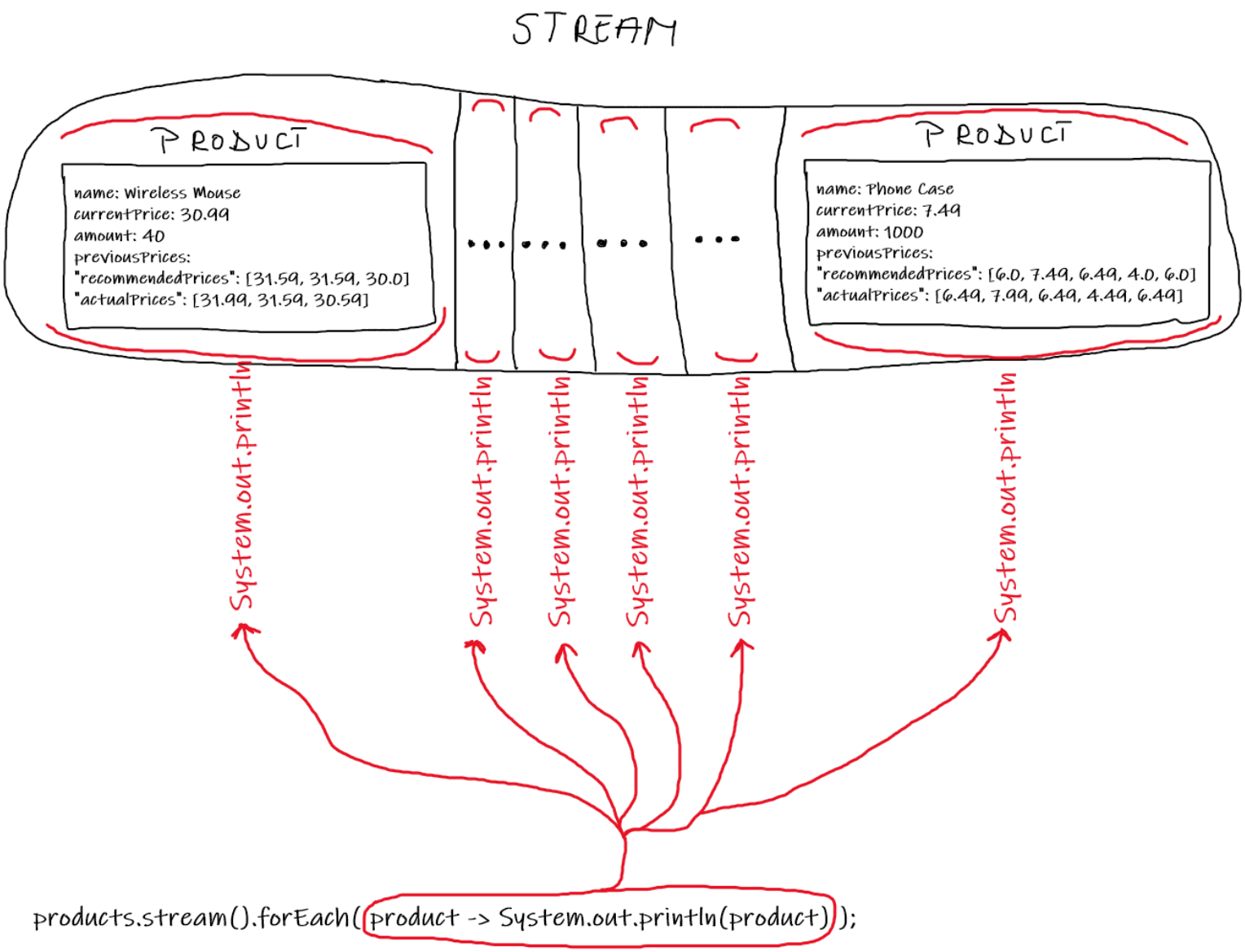
In the above presented code, a stream is used to get access to the *.forEach()* method, which, combined with a predicate, applies it on every single item of the stream.
When applying *.stream()* the following stream will be constructed:



FullObjectStream.png

For the sake of the used space, I represented the middle fourth objects with dots, but think of them as they were there.

The predicate used inside .*forEach*() will be applied on each of the stream elements as can be seen in the following picture:



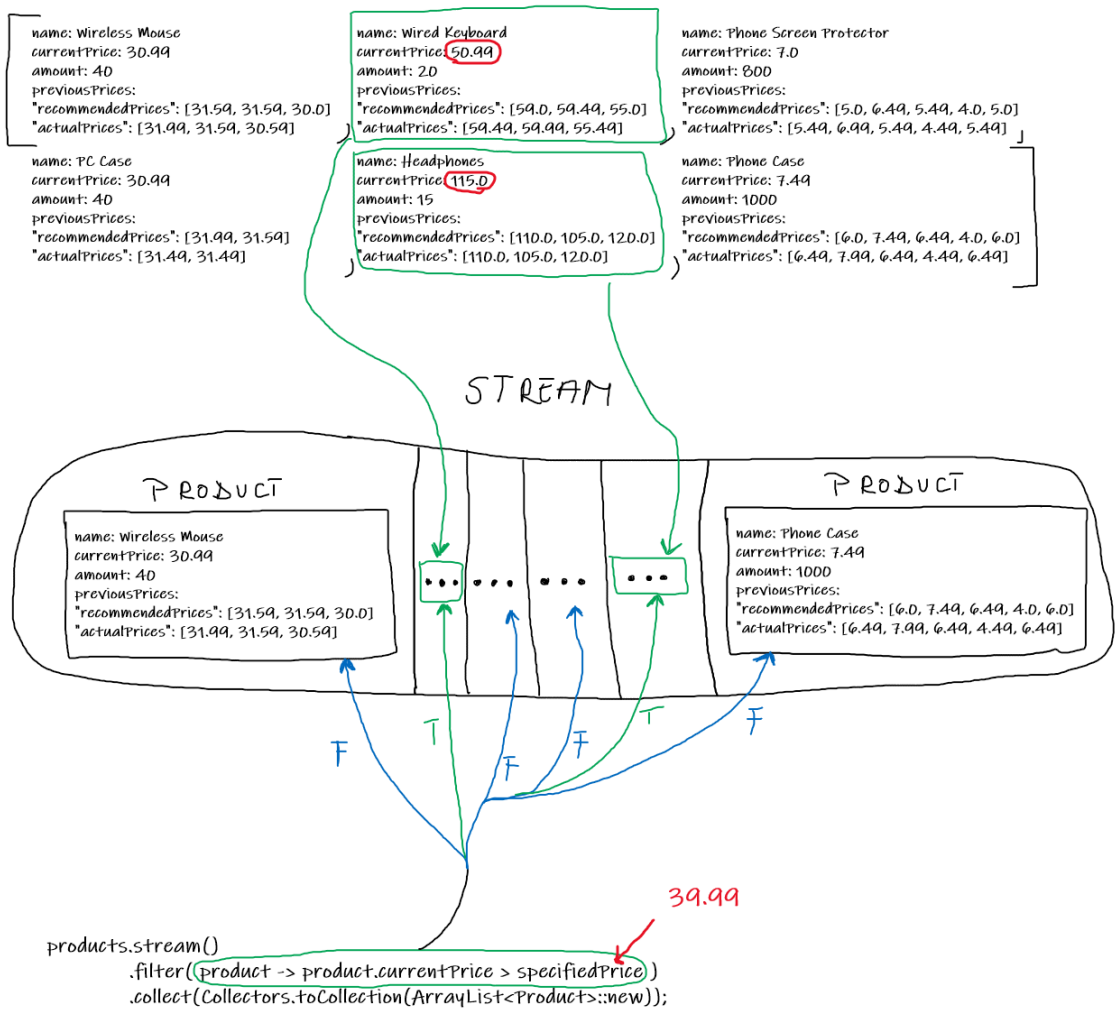
ObjectStreamForeach.png

In the above picture one can see that for each of the stream elements, referred by us as *product*, the *System.out.println*(*product*) method is called, hence producing the desired effect of printing the entire ArrayList of products.

Filtering items and collecting results

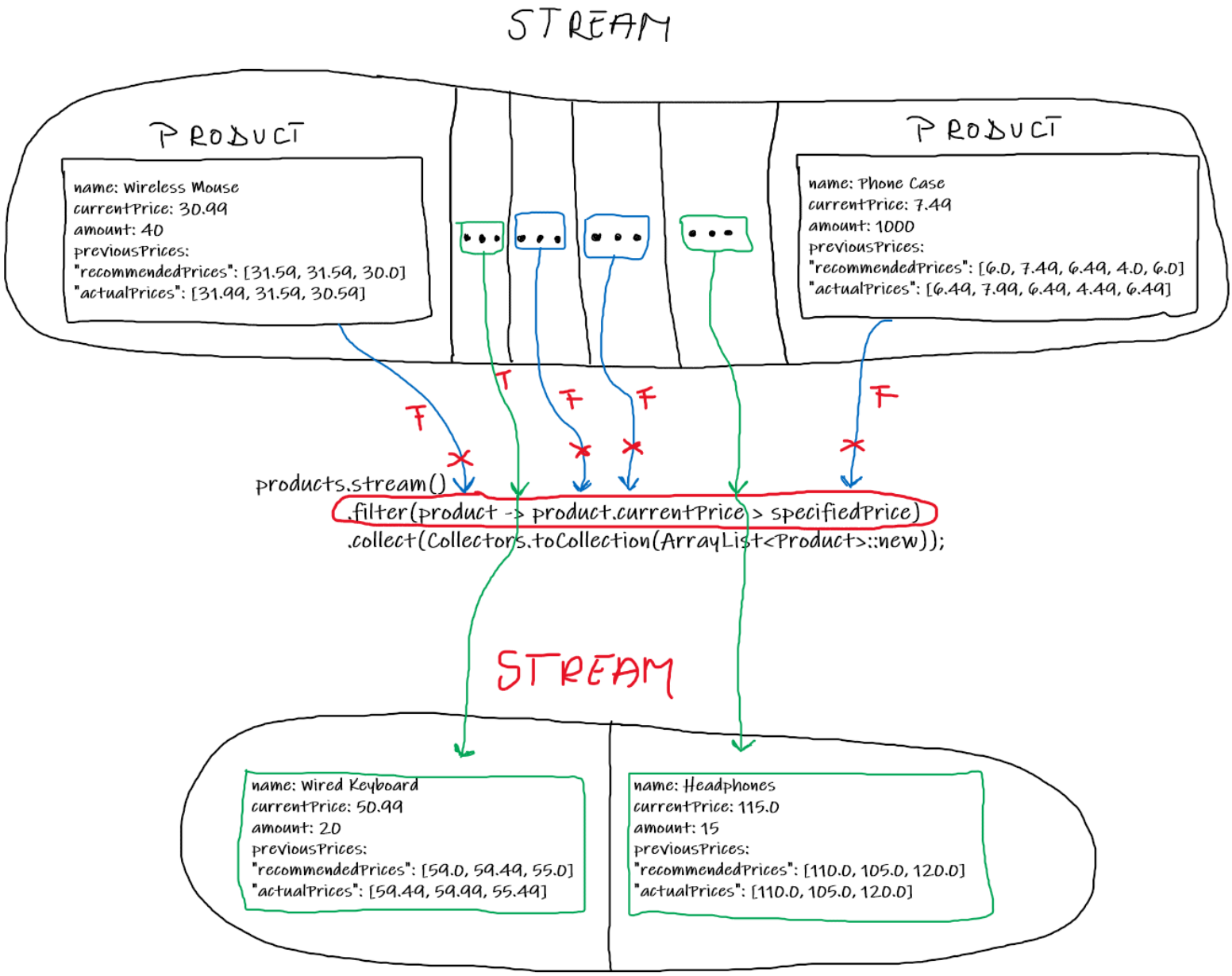
```
private static ArrayList<Product> extractProductsOver(ArrayList<Product> products,
                                                      double specifiedPrice) {
    return products.stream()
                   .filter(product -> product.currentPrice > specifiedPrice)
                   .collect(Collectors.toCollection(ArrayList<Product>::new));
}
```

Let's say we would want to get a list of all the products over the price 39.99. We can do that by combining the *.filter()* method with a predicate that returns true if a product is over the specified price.



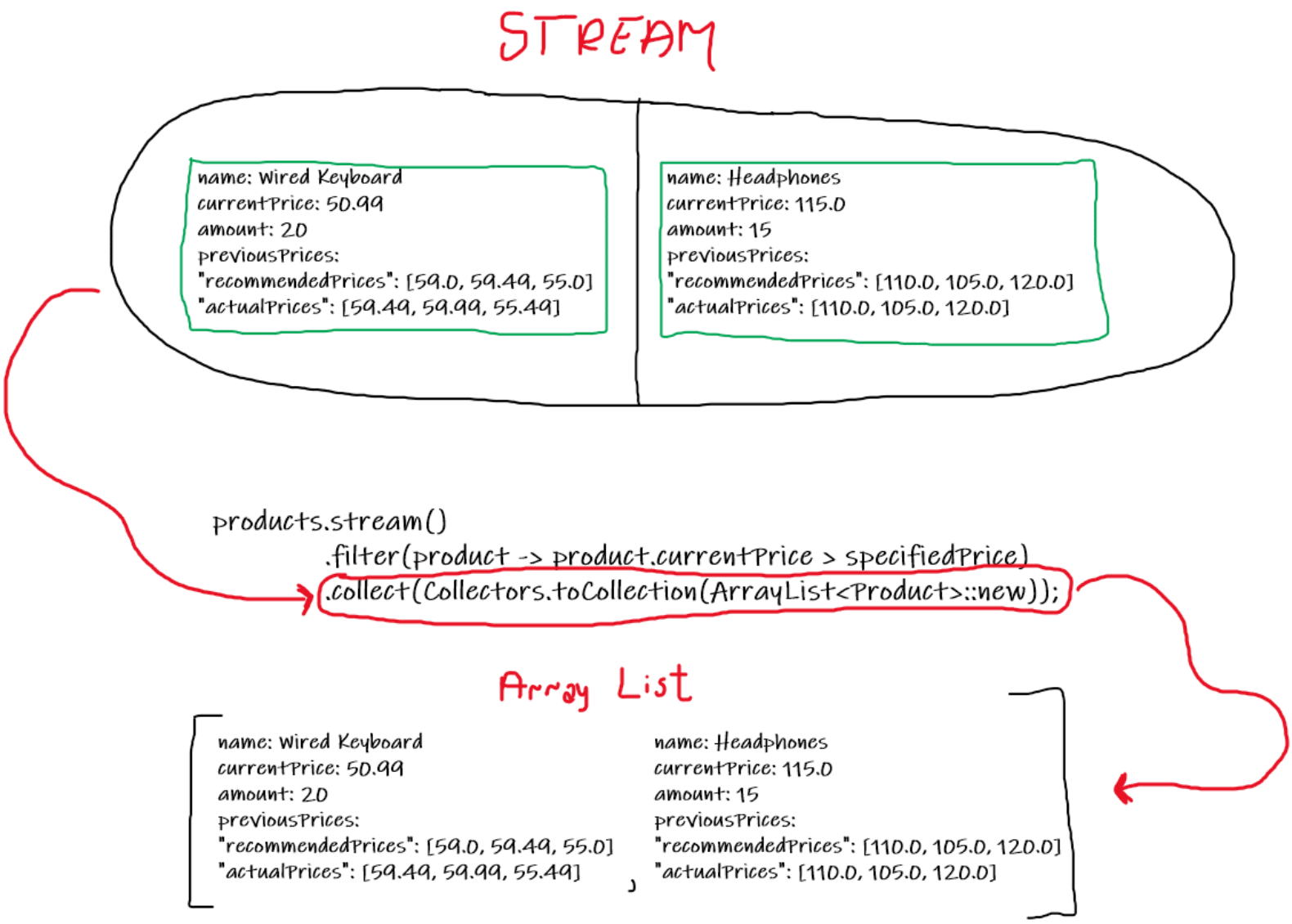
FilterPredicate.png

In the above example, with F are represented connections to objects that return false when we apply the predicate on them, and with T those that return true. After using the $.filter()$ method, in the stream will be only those objects that when applying the predicate on them return T/true:



ObjectStreamFilter.png

Finally, we want to collect the stream elements into a new list (ArrayList). We can do this with the ".collect()" method and with the help of the Collectors, a class which help us with the way we want to collect our items:

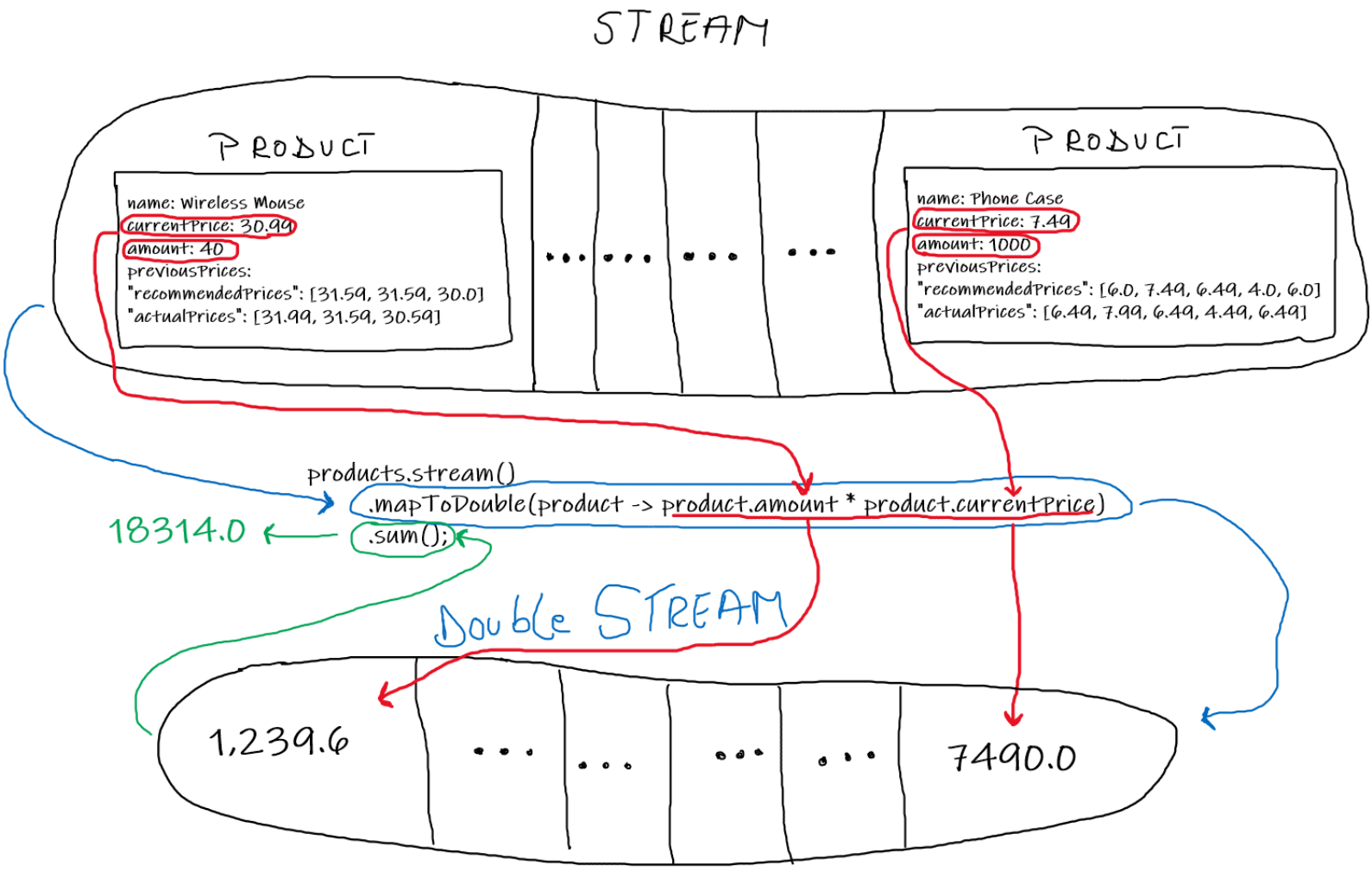


ObjectStreamCollect.png

Calculating the total list price

```
double totalSum =
products.stream()
        .mapToDouble(product -> product.amount * product.currentPrice)
        .sum();
```

For calculating the total price, we should sum the products' prices. We can do this with streams. Firstly, we map each element to a double (*.mapToDouble*()). For this, we use a predicate that computes the total price of a product. With the new created stream, we compute the total price by using *.sum*() method which sums all the stream elements.



ObjectStreamSum.png