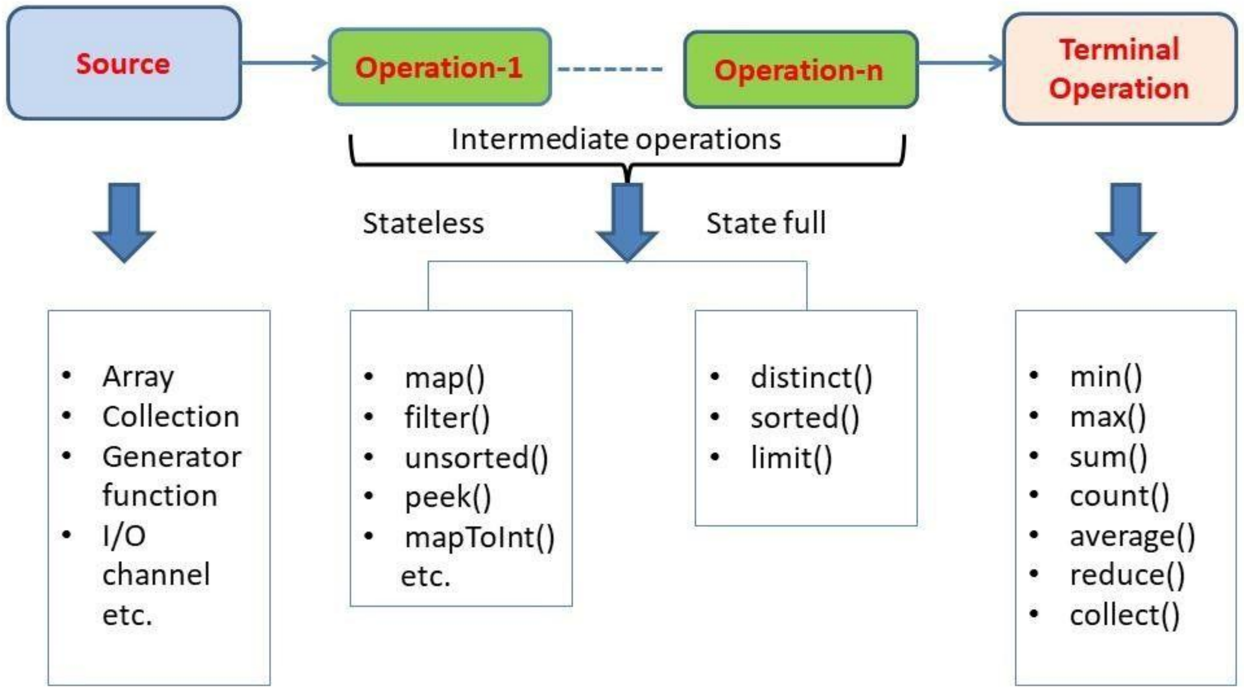
Week 12 - Java Streams Workshop

Louis Botha, `louis.botha@tuni.fi`



A Complete Tutorial on Java Streams - PDF.co

An online grocery shop that allows customers to collect, and save, multiple different shopping carts at the same time has asked you to implement a joint checkout for their multiple-shopping cart system. The checkout procedure should concatenate the price for all items in all shopping carts, and then present that to the customer. To do this, perform the following steps:

1. Create a project in IntelliJ or with VSCode.
2. Create the main entry point for your application, called **Main.java.**

```
public class Main {
    public static void main(String[] args) {
        // write your code here
    }
}
```

3. Create a new class, called **ShoppingArticle**. Make it so that we can easily access it from the main entry point for our program. This class should contain the name of the article and the price of that article.

   Let the **price** be a double variable:

```
public final class ShoppingArticle {
    private String name;
    private double price;

    public ShoppingArticle(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return this.name;
    }

    public double getPrice() {
        return this.price;
    }
}
```

4. Now create a simple **ShoppingCart** class. In this version, we will only allow one item per article in the cart, so a list will be enough to keep the articles in **ShoppingCart**:

```
import java.util.List;

public class ShoppingCart {
    List<ShoppingArticle> articles;
    public ShoppingCart(List<ShoppingArticle> list) {
        articles = List.copyOf(list);
    }
}
```

5. Create your first shopping cart, **fruitCart**, and add three fruit articles to it – **Orange**, **Apple**, and **Banana**—one of each type. Set the per-unit price to **1.5**, **1.7**, and **2.2** each:

```
import java.util.List;
class Main {
    public static void main(String[] args) {
        ShoppingCart fruitCart = new ShoppingCart(
                List.of(
                        new ShoppingArticle("Orange", 1.5),
                        new ShoppingArticle("Apple", 1.7),
                        new ShoppingArticle("Banana", 2.2)));
    }
}
```

6. Create another **ShoppingCart**, but this time with vegetables—**Cucumber**, **Salad**, and **Tomatoes.** Set a price in Java-$ for them as well, as **0.8**, **1.2**, and **2.7**:

```
ShoppingCart vegetableCart = new ShoppingCart(
        List.of(
                new ShoppingArticle("Cucumber", 0.8),
                new ShoppingArticle("Salad", 1.2),
                new ShoppingArticle("Tomatoes", 2.7)));
```

7. Wrap up the test shopping carts with a third and final **shoppingCart** containing some meat and fish. They're usually a little more expensive than fruit and vegetables:

```
ShoppingCart meatAndFishCart = new ShoppingCart(
        List.of(
                new ShoppingArticle("Cod", 46.5),
                new ShoppingArticle("Beef", 29.1),
                new ShoppingArticle("Salmon", 35.2)));
```

8. Now it's time to start implementing the function that will calculate the total price of all the items in a shopping cart. Declare a new function in main that takes a **ShoppingCart** as an argument and returns a double. Let it be static so that we can easily use it in the **main** function:

```
public static double calculatePrice(ShoppingCart shoppingCart)
{


}
```

9. Build a pipeline starting with a stream of all of the carts:

```
public static double calculatePrice(ShoppingCart shoppingCart)
{
    return Stream.of(shoppingCart);
}
```

10. Add a **flatMap()** operation to extract a single stream of **ShoppingArticles** for all **ShoppingCarts:**

```
private static double calculatePrice(ShoppingCart shoppingCar
t) {
    return Stream.of(shoppingCart)
            .flatMap((cart) -> {
                return cart.articles.stream();
            });
}
```

11. Extract the price for each **ShoppingArticle** using the **mapToDouble()** operation; this will create a **DoubleStream:**

```
private static double calculatePrice(ShoppingCart shoppingCar
t) {
    return Stream.of(shoppingCart)
            .flatMap((cart) -> {
                return cart.articles.stream();
            })
            .mapToDouble((item) -> {
                return item.getPrice();
            })
}
```

12. Finally, reduce the prices of all **ShoppingArticle** to a sum, using the **sum()** method that is available in **DoubleStream**:

```
private static double calculatePrice(ShoppingCart shoppingCar
t) {
    return Stream.of(shoppingCart)
            .flatMap((cart) -> {
                return cart.articles.stream();
            })
            .mapToDouble((item) -> {
                return item.getPrice();
            })
            .sum();
}
```

13. Now you have a function that will reduce a list of **ShoppingCart** to a sum. All you have to do now is to apply this function to your **ShoppingCart** class, and then print out the resulting sum to the terminal, rounding it to two decimals:

```
double fruitCartSum = calculatePrice(fruitCart);
System.out.println(String.format("FruitCartSum: %.2f", fruitCa
rtSum));
```

```
double vegetableCartSum = calculatePrice(vegetableCart);
System.out.println(String.format("VegetableCartSum: %.2f", veg
etableCartSum));


double meatAndFishCartSum = calculatePrice(meatAndFishCart);
System.out.println(String.format("MeatAndFishCartSumSum: %.2
f", meatAndFishCartSum));
```

14. The output of the current implementation should look like this.

```
FruitCartSum: 5,40
VegetableCartSum: 4,70
MeatAndFishCartSumSum: 110,80
```

15. You can also calculate sum of all the shopping carts like this. Create a duplicate of the function and change the **calculatePrice** function.

```
private static double calculatePrice(ShoppingCart... carts) {
    return Stream.of(carts)
            .flatMap((cart) -> {
                return cart.articles.stream();
            })
            .mapToDouble((item) -> {
                return item.getPrice();
            })
            .sum();
}
```

> 🗒️ ***SideNote***
>
> *Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments. Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.*
>
> *https://www.geeksforgeeks.org/variable-arguments-varargs-in-java/*

16. Now you have a function that will reduce a list of **ShoppingCart** to a sum. All you have to do now is to apply this function to your **ShoppingCart** class, and then print out the resulting sum to the terminal, rounding it to two decimals:

```
double totalSum = calculatePrice(fruitCart, vegetableCart, mea
tAndFishCart);
System.out.println(String.format("Sum: %.2f", totalSum));
```

You've now created your first complete piece of code using the functional Java Stream API. You've created a stream of complex objects, applying a mapping operation to the elements of the stream to transform them, and then another mapping operation to transform the elements yet again, changing the stream type twice. Finally, you reduced the whole stream to a single primitive value that was presented to the user.

A web-based grocery shop has implemented its very own database based on a standard Java **List** collection, and has also implemented a backup system where the database is backed up to CSV files. However, they still haven't built a way of restoring that database from a CSV file. They have asked you to build a system that will read such a CSV file, inflating its contents to a list.

The database backup CSV file contains one single type of object: **ShoppingArticle.** Each article has a **name**, a **price**, a **category**, and finally, a **unit.** The name, category, and unit should each be a **String**, and the price a **double.**

1. Create a project in IntelliJ or with VSCode.
2. Create the main entry point for your application, called **Main.java.**

```
public class Main {
    public static void main(String[] args) {
        // write your code here
    }
}
```

3. Create a new class, called **ShoppingArticle**. Make it so that we can easily access it from the main entry point for our program. This class should contain the name of the article and the price of that article. Override the **toString** method to make it easy to print articles to the terminal later:

```
public class ShoppingArticle {
    private String name;
    private String category;
    private double price;
    private String unit;

    public ShoppingArticle(String name, String category, doubl
e price, String unit) {
        this.name = name;
        this.category = category;
        this.price = price;
        this.unit = unit;
    }

    public String getName() {
        return this.name;
    }

    public String getCategory() {
        return this.category;
    }

    public double getPrice() {
        return this.price;
    }

    public String getUnit() {
        return this.unit;
    }

    @Override
    public String toString() {
        return name + " (" + category + ")";
    }
}
```

4. Create a new file called **database.csv** file, get the content from ::slack:.

5. In your **Main.java** class, add a function that produces **List<ShoppingArticle>**. This will be our function to load the database into a list. Since the function will be loading a file, it needs to throw an I/O exception (**IOException**):

```
private static List<ShoppingArticle> loadDatabaseFile() throws
IOException {
    return null;
}
```

6. Call this function from your **main** method:

```
import java.io.IOException;
import java.util.List;

class Main {
    public static void main(String[] args) {
        try {
            List<ShoppingArticle> database = loadDatabaseFile
();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    private static List<ShoppingArticle> loadDatabaseFile() th
rows IOException {
        return null;
    }
}
```

7. Start by loading the database file with a try-with-resources block. Use **Files.lines** to load all the lines from the **database.csv** file. It should look something like this:

```
  private static List<ShoppingArticle> loadDatabaseFile() thro
ws IOException {
      try (Stream<String> stream = Files.lines(Path.of("databa
se.csv"))) {
      }
      return null;
  }
```

8. Let's peek into the stream in order to look at the state of it right now. Intermediate operations will only run when there's a terminal operation defined, so add a **count()** at the end just to force it to execute the whole pipeline:

```
private static List<ShoppingArticle> loadDatabaseFile() throws
IOException {
    try (Stream<String> stream = Files.lines(Path.of("databas
e.csv"))) {
        stream
            .peek((line) -> {
                System.out.println(line);
            })
            .count();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

9. This should print every single line of the file. Notice that it also prints the header line—which we're not concerned with when converting to **ShoppingArticles**.

10. Since we're not really interested in the first row, the headings, add a **skip** operation just before the **count()** method:

```
private static List<ShoppingArticle> loadDatabaseFile() throws
IOException {
    try (Stream<String> stream = Files.lines(Path.of("databas
e.csv"))) {
        stream
            .peek((line) -> {
                System.out.println(line);
            })
            .skip(1)
            .count();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

11. Now you have every single line of the database file loaded as elements in the stream, except for the header. It's time to extract every piece of data from those lines; a suitable operation for this is **map**. Split every line into **String** arrays using the **split()** function:

```
  private static List<ShoppingArticle> loadDatabaseFile() throws IOException {
      try (Stream<String> stream = Files.lines(Path.of("database.csv"))) {
          stream
              .peek((line) -> {
                  System.out.println(line);
              })
              .skip(1)
              .map((line) -> {
                  return line.split(",");
              })
              .count();
      } catch (IOException e) {
          e.printStackTrace();
      }
      return null;
  }
```

12. Add another **peek** operation to find out how the **map** operation changed the stream; your stream type should now be **Stream<String[]>**:

```
    private static List<ShoppingArticle> loadDatabaseFile() throws IOException {
        try (Stream<String> stream = Files.lines(Path.of("database.csv"))) {
            stream
                .peek((line) -> {
                    System.out.println(line);
                })
                .skip(1)
                .map((line) -> {
                    return line.split(",");
                })
                .peek((arr) -> {
                    System.out.println(Arrays.toString(arr));
                })
                .count();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

13. Add another **map** operation, but this time to turn the stream into **Stream<ShoppingArticle>**:

```
private static List<ShoppingArticle> loadDatabaseFile() throws
IOException {
    try (Stream<String> stream = Files.lines(Path.of("databas
e.csv"))) {
        stream
          .peek((line) -> {
              System.out.println(line);
          })
          .skip(1)
          .map((line) -> {
              return line.split(",");
          })
          .peek((arr) -> {
              System.out.println(Arrays.toString(arr));
          })
          .map((arr) -> {
              return new ShoppingArticle(arr[0], arr[1], Double.valueOf(arr[2]), arr[3]);
          })
          .count();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

14. Now you can **peek** again to ensure the articles were created properly:

```
private static List<ShoppingArticle> loadDatabaseFile() throws IOException {
    try (Stream<String> stream = Files.lines(Path.of("database.csv"))) {
        stream
            .peek((line) -> {
                System.out.println(line);
            })
            .skip(1)
            .map((line) -> {
                return line.split(",");
            })
            .peek((arr) -> {
                System.out.println(Arrays.toString(arr));
            })
            .map((arr) -> {
                return new ShoppingArticle(arr[0], arr[1], Double.valueOf(arr[2]), arr[3]);
            })
            .peek(System.out::println)
            .count();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

15. Collect all the articles in a list. Use an unmodifiable list to protect the database from unwanted modifications:

```
private static List<ShoppingArticle> loadDatabaseFile() throws
IOException {
    try (Stream<String> stream = Files.lines(Path.of("databas
e.csv"))) {
        return stream.peek((line) -> {
                    System.out.println(line);
              })
              .skip(1)
              .map((line) -> {
                    return line.split(",");
              })
              .peek((arr) -> {
                    System.out.println(Arrays.toString(arr));
              })
              .map((arr) -> {
                    return new ShoppingArticle(arr[0], arr[1], D
ouble.valueOf(arr[2]), arr[3]);
              })
              .peek(System.out::println)
              .collect(Collectors.toUnmodifiableList());
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

This may seem verbose, as some operations could have been applied together to make it shorter. However, there's a point to keeping every single operation small, and that's to make the whole logic very transparent. If you find a problem with the pipeline, you can simply move a single operation in the pipeline, and that should sort all problems.
If you combine multiple steps in an operation, it's more difficult to move the operations around in the pipeline or to replace it fully.

Searching for Specifics

With the database loaded, apply some searching logic:

1. Build a function that will find the cheapest fruit from a list of **ShoppingArticles**.
2. Build a function that will find the most expensive vegetable from a list of **ShoppingArticles**.
3. Build a function that will gather all fruits in a separate list.
4. Build a function that will find the five least expensive articles in the database.
5. Build a function that will find the five most expensive articles in the database.