Java 8 (Modern Java) Functional Streams API & examples

Consider following core classes (POJOs – Plain Old Java Object | Model | Entity) , from typical Ecommerce application.

```
public class Category {
   private int categoryId
   private String name; //unique
   private String description;
   private List<Product> products=new ArrayList<>();
   //constructor , getters setters
}
public class Product {
   private int productId;
   private String name;//unique
   private double price;
   private int quantity;
   private LocalDate manufactureDate;
   private Category category;
//constructor , getters setters
}
 // Sample Data
Category electronics = new Category(1, "Electronics", "Electronic devices", new ArrayList<>());
Category groceries = new Category(2, "Groceries", "Daily essentials", new ArrayList<>());
      Product p1 = new Product(101, "Laptop", 90000, 5, LocalDate.of(2024, 1, 10), electronics);
      Product p2 = new Product(102, "Mobile", 50000, 10, LocalDate.of(2023, 11, 20), electronics);
      Product p3 = new Product(103, "TV", 30000, 3, LocalDate.of(2024, 5, 15), electronics);
      Product p4 = new Product(201, "Rice", 50, 100, LocalDate.of(2025, 1, 1), groceries);
      Product p5 = new Product(202, "Milk", 30, 50, LocalDate.of(2025, 2, 1), groceries);
      electronics.getProducts().addAll(Arrays.asList(p1, p2, p3));
      groceries.getProducts().addAll(Arrays.asList(p4, p5));
      List<Category> categories = Arrays.asList(electronics, groceries);
```

// **STREAM EXAMPLES**

1. **filter → Get all products above 40,000**

```
List<Product> expensive = categories.stream() //Stream<Category>
        .flatMap(c -> c.getProducts().stream()) //Stream<Product>
        .filter(p -> p.getPrice() > 40000) //filtered Stream<Product>
        .collect(Collectors.toList()); //List<Product>
System.out.println("Expensive products (>40k): " + expensive);
```

2. **map → Extract product names**

```
List<String> names = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .map(Product::getName)//Using Method reference, lambda – p-> p.getName() ,
Stream<String>
        .collect(Collectors.toList());
System.out.println("Product names: " + names);
```

3. flatMap → Get all products in one list

```
List<Product> allProducts = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .collect(Collectors.toList());
System.out.println("All products: " + allProducts);
```

4. reduce → Total value of all products (price * quantity)

```
double totalValue = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .mapToDouble(p -> p.getPrice() * p.getQuantity()) //DoubleStream containing total product
price
        .reduce(0.0, Double::sum); //OR simply .sum()
System.out.println("Total stock value = " + totalValue);
```

5. collect → Group by category name

```
Map<String, List<Product>> productsByCategory = categories.stream()
        .collect(Collectors.toMap(
                Category::getName,//key mapper function
                Category::getProducts//value mapper function
        ));
System.out.println("Products by category: ");
productsByCategory.forEach((k,v) -> System.out.println(k+" "+v));
```

6. **findAny → Get any product cheaper than 100**

```
Product cheap = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .filter(p -> p.getPrice() < 100)
        .findAny()
       .orElseThrow(() -> new ProductHandlingException("Cheap product not found !!!!"));
System.out.println("Any cheap product: " + cheap);
```

7. **min & max**

```
Product cheapest = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .min(Comparator.comparingDouble(Product::getPrice))//or use usual lambda here – based on price
        .orElse(null);
System.out.println("Cheapest product = " + cheapest);

Product costliest = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .max(Comparator.comparingDouble(Product::getPrice))
        .orElse(null);
System.out.println("Costliest product = " + costliest);
```

8**. Display products sorted (by price)**

```
categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .sorted(Comparator.comparingDouble(Product::getPrice))
        .forEach(System.out::println);
```

9. distinct

```
List<String> distinctCategoryNames = categories.stream()
        .map(Category::getName)
        .distinct()
        .collect(Collectors.toList());
System.out.println("Distinct category names: " + distinctCategoryNames);
```

10. anyMatch, allMatch, noneMatch

```
boolean hasExpensive = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .anyMatch(p -> p.getPrice() > 80000);
System.out.println("Any product >80k? " + hasExpensive);

boolean allAffordable = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .allMatch(p -> p.getPrice() < 10000);
System.out.println("All affordable (<10K)? " + allAffordable);

boolean noneFree = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .noneMatch(p -> p.getPrice() == 0);
System.out.println("No free products? " + noneFree);
```

11. forEach

```
System.out.println("Print all products:");
```

```
        categories.stream()
                .flatMap(c -> c.getProducts().stream())
                .forEach(System.out::println);
    }
}
```

12. Group products by category name

```
        List<Product> allProducts = categories.stream()
                .flatMap(c -> c.getProducts().stream())
                .collect(Collectors.toList());
        System.out.println("All products: " + allProducts);
        Map<String, List<Product>> productsByCategory = allProducts.stream()
                .collect(Collectors.groupingBy(p -> p.getCategory().getName()));
        productsByCategory.forEach((k,v) -> System.out.println(k+" "+v));
```

13. Count products per category

```
        Map<String, Long> countByCategory = allProducts.stream()
                .collect(Collectors.groupingBy(p -> p.getCategory().getName(), Collectors.counting()));
        countByCategory.forEach((k,v) -> System.out.println(k+" "+v));
```

14. Average price per category

```
        Map<String, Double> avgPriceByCategory = allProducts.stream()
                .collect(Collectors.groupingBy(p -> p.getCategory().getName(),
                        Collectors.averagingDouble(Product::getPrice)));
        avgPriceByCategory.forEach((k,v) -> System.out.println(k+" "+v));
```

15. Total stock value per category (sum of price*qty)

```
        Map<String, Double> totalStockValue = allProducts.stream()
                .collect(Collectors.groupingBy(p -> p.getCategory().getName(),
                        Collectors.summingDouble(p -> p.getPrice() * p.getQuantity())));
        System.out.println("Total stock value per category: " );
        totalStockValue.forEach((k,v) -> System.out.println(k+" "+v));
```

16. Partition products into expensive vs cheap (threshold = 5000)

```
Map<Boolean, List<Product>> partitioned = allProducts.stream()
        .collect(Collectors.partitioningBy(p -> p.getPrice() > 5000));
System.out.println("Partitioned products (expensive vs cheap): ");
partitioned.forEach((k,v) -> System.out.println(k+" "+v));
```

17. Join product names into string

```
String productNames = allProducts.stream()
        .map(Product::getName)
        .collect(Collectors.joining(", ", "[", "]"));//delimiter , prefix,suffix
System.out.println("All product names: "+productNames);
```

18. Max price per category

```
Map<String, Optional<Product>> maxPriceByCategory = allProducts.stream()
        .collect(Collectors.groupingBy(p -> p.getCategory().getName(),
                Collectors.maxBy(Comparator.comparingDouble(Product::getPrice))));
System.out.println("Max priced product per category: ");
maxPriceByCategory.forEach((k,v)-> System.out.println(k+" "+v));
```

19. Mapping products to just names per category

```
Map<String, List<String>> productNamesByCategory = allProducts.stream()
        .collect(Collectors.groupingBy(p -> p.getCategory().getName(),
                Collectors.mapping(Product::getName, Collectors.toList())));
System.out.println("Product names per category: " );
productNamesByCategory.forEach((k,v)-> System.out.println(k+" "+v));
    }
}
```

```
// Sample Data
```

```
Category electronics = new Category(1, "Electronics", "Electronic devices", new ArrayList<>());
Category groceries = new Category(2, "Groceries", "Daily essentials", new ArrayList<>());
Category clothing = new Category(3, "Clothing", "Fashion wear", new ArrayList<>());

Product p1 = new Product(101, "Laptop", 90000, 5, LocalDate.of(2024, 1, 10), electronics);
Product p2 = new Product(102, "Mobile", 50000, 10, LocalDate.of(2023, 11, 20), electronics);
Product p3 = new Product(103, "TV", 30000, 3, LocalDate.of(2024, 5, 15), electronics);
Product p4 = new Product(201, "Rice", 50, 100, LocalDate.of(2025, 1, 1), groceries);
Product p5 = new Product(202, "Milk", 30, 50, LocalDate.of(2025, 2, 1), groceries);
Product p6 = new Product(301, "Jeans", 2000, 20, LocalDate.of(2024, 9, 1), clothing);
Product p7 = new Product(302, "T-Shirt", 1000, 30, LocalDate.of(2024, 8, 15), clothing);

electronics.getProducts().addAll(Arrays.asList(p1, p2, p3));
groceries.getProducts().addAll(Arrays.asList(p4, p5));
clothing.getProducts().addAll(Arrays.asList(p6, p7));

List<Category> categories = Arrays.asList(electronics, groceries, clothing);
```

20. Flatten products

```
List<Product> allProducts = categories.stream()
        .flatMap(c -> c.getProducts().stream())
        .collect(Collectors.toList());
```

21. Top 3 most expensive products

```
List<Product> top3Expensive = allProducts.stream()
        .sorted(Comparator.comparingDouble(Product::getPrice).reversed())//or can use Comparator lambda also
        .limit(3)
        .collect(Collectors.toList());
System.out.println("Top 3 expensive products: ");
 top3Expensive.forEach(System.out::println);
```

22. Find newest product (latest manufacture date)

```
Product newest = allProducts.stream()
    .max(Comparator.comparing(Product::getManufactureDate))
    .orElseThrow();
System.out.println("Newest product: " + newest);
```

23. Find oldest product (earliest manufacture date)

```
Product oldest = allProducts.stream()
    .min(Comparator.comparing(Product::getManufactureDate))
    .orElseThrow();
System.out.println("Oldest product: " + oldest);
```

24. Total inventory value (all categories)

```
double totalInventory = allProducts.stream()
    .mapToDouble(p -> p.getPrice() * p.getQuantity())
    .sum();
System.out.println("Total inventory value = " + totalInventory);
```

25. Products cheaper than 2000 grouped by category

```
Map<String, List<Product>> cheapProductsByCategory = allProducts.stream()
    .filter(p -> p.getPrice() < 2000)
    .collect(Collectors.groupingBy(p -> p.getCategory().getName()));
System.out.println("Cheap products per category (<2000): ");
cheapProductsByCategory.forEach((k,v)-> System.out.println(k+" "+v));
```

26. Average quantity per category

```
Map<String, Double> avgQtyByCategory = allProducts.stream()
```

```
        .collect(Collectors.groupingBy(p -> p.getCategory().getName(),
                Collectors.averagingInt(Product::getQuantity)));
    System.out.println("Average quantity per category: " + avgQtyByCategory);
```

Java.util.stream.Stream<T> API

1. **filter**

Select elements that match a condition.

stream.filter(p -> p.getPrice() > 10000)

Input: Stream<Product>

Output: Stream<Product> (only expensive products)

2. **map**

Transform each element into something else.

stream.map(Product::getName)// p -> p.getName()

Input: Stream<Product>

Output: Stream<String>

3. **flatMap (map & faltten)**

Flatten nested collections into a single stream(suitable for one→many mapping, One Category→Many Products)

categories.stream().flatMap(c -> c.getProducts().stream())

Input: Stream<Category>

Output: Stream<Product>

4. sorted

Sort elements.

stream.sorted(Comparator.comparingDouble(Product::getPrice)) //(p1,p2)-> ((Double)p1.getPrice()).compareTo(p2.getPrice());

Input: Stream<Product>

Output: Stream<Product> (sorted as per price asc)

5. distinct

Remove duplicates (based on equals/hashCode).

stream.distinct()

6**. limit / skip**

Get first N elements or skip N elements.

stream.limit(3) // top 3

stream.skip(5) // ignore first 5

7. **findFirst / findAny**

Short-circuit terminal operations to fetch one element.

stream.findFirst() // deterministic (first element)

stream.findAny() // non-deterministic (any match, faster in parallel)

Output: Optional<T>

8. **min / max**

Find min or max based on comparator.

stream.min(Comparator.comparing(Product::getPrice))

stream.max(Comparator.comparing(Product::getManufactureDate))

Output: Optional<T>

9. **reduce**

Aggregate manually (sum, product, concatenation).

stream.map(Product::getPrice).reduce(0.0, Double::sum) //equivalent to sum()

Output: one value (e.g. Double)

10. **collect (collect stream elements into some collection | map)**

Powerful aggregation into collections or maps.

stream.collect(Collectors.toList()) // List<Product>

stream.collect(Collectors.toSet()) // Set<Product>

stream.collect(Collectors.toMap(Product::getId, Product::getName)) // Map<id,name>

11. **groupingBy**

Group by a classifier (like SQL GROUP BY).

stream.collect(Collectors.groupingBy(p -> p.getCategory().getName()))

Output: Map<String, List<Product>>

12**. partitioningBy**

Special case of grouping (boolean predicate).

stream.collect(Collectors.partitioningBy(p -> p.getPrice() > 1000))

Output: Map<Boolean, List<Product>>

13. **counting / averaging / summing**

Built-in collectors for statistics

Collectors.counting() //count

Collectors.averagingDouble(Product::getPrice) //average

Collectors.summingDouble(p -> p.getPrice() * p.getQuantity())

14. joining

Concatenate strings.

stream.map(Product::getName)

.collect(Collectors.joining(", ", "[", "]")) //delimiter , prefix , suffix

Output: [Laptop, Mobile, TV, Rice, Milk]

15. mapping (inside grouping)

Transform values during grouping.

Collectors.mapping(Product::getName, Collectors.toList())

Output: Map<CategoryName, List<String>>

16. collectingAndThen

Post-process collector result.

Collectors.collectingAndThen(

Collectors.maxBy(Comparator.comparing(Product::getPrice)),

Optional::get

)

Example: directly get max element instead of Optional.

17. anyMatch / allMatch / noneMatch

Boolean checks on stream.

```
stream.anyMatch(p -> p.getPrice() > 100000)   // true/false
```

18. mapToInt / mapToDouble – mapper functions

Primitive specialization for performance.

```
stream.mapToDouble(p -> p.getPrice() * p.getQuantity()).sum()
```

Output: primitive values (int, long, double)

19. forEach

Terminal operation.

```
stream.forEach(System.out::println);
```

Common Optional<T> Handling Methods

1. isPresent() / get() (old-school, not recommended for production)

```
Optional<Product> opt = products.stream()
                        .filter(p -> p.getPrice() > 5000)
                        .findFirst();


if (opt.isPresent()) {
   System.out.println("Found: " + opt.get().getName());
}
```

get() without checking may throw the execption!

2. ifPresent()

Run logic only if value exists.

```
opt.ifPresent(p -> System.out.println("Found: " + p.getName()));
```

3**. ifPresentOrElse() (Java 9+)**

Handle both present & empty cases.

```
opt.ifPresentOrElse(
   p -> System.out.println("Found: " + p.getName()),
   () -> System.out.println("No product found!")
);
```

4**. orElse()**

Provide a default if empty.

Product defaultProduct = opt.orElse(new Product(0, "Default", 0.0, 0, null, null));

5. **orElseGet()**

Like orElse, but lazily computes default (supplier is called only if needed).

Product p = opt.orElseGet(() -> createDummyProduct());

6. **orElseThrow()**

Throw supplied exception if empty.

Product p = opt.orElseThrow(() -> new RuntimeException("Product not found"));

7. **map()**

Transform the value inside the Optional (without unwrapping).

Optional<String> productName = opt.map(Product::getName);

System.out.println(productName.orElse("Unknown"));

8. filter()

Keep only if condition matches.

opt.filter(p -> p.getQuantity() > 10)

.ifPresent(p -> System.out.println("Sufficient stock: " + p.getName()));

Examples

1.Find first expensive product > 70k

Optional<Product> expensiveOpt = products.stream()

.filter(p -> p.getPrice() > 70000)

.findFirst();

expensiveOpt.ifPresent(p -> System.out.println("Expensive: " + p));

2. Use **orElse**

Product safeProduct = expensiveOpt.orElse(new Product(0, "Default", 0, 0, null, null));

System.out.println("OrElse fallback: " + safeProduct);

3. Use orElseThrow

Product mustExist = expensiveOpt.orElseThrow(() -> new RuntimeException("No product found > 70k!"));

System.out.println("OrElseThrow result: " + mustExist);

4. **Extract just name with map**

String productName = expensiveOpt.map(Product::getName).orElse("Unknown");

System.out.println("Name: " + productName);

5. Apply filter on Optional

expensiveOpt.filter(p -> p.getQuantity() > 2)

.ifPresent(p -> System.out.println("Has enough stock: " + p));

6. **expensiveOpt.orElseThrow();** // throws NoSuchElementException if empty

7. **Custom Exception**

// Custom exception for product not found

expensiveOpt.orElseThrow(() -> new ProductNotFoundException(“No product found above 1 lakh!”)); // throws custom exception if empty