# Advanced Use Case: Java Stream API - E-Commerce Order Analytics

## Problem Statement

An e-commerce platform processes millions of customer orders daily. The business wants to:  
1. Find top 3 customers with the highest total order amount.  
2. Get daily sales summary (total revenue per date).  
3. Group orders by customer and summarize total amount and item count.  
4. Extract list of all unique product names sold.  
5. Identify customers who have placed orders on more than 1 day.

## Entity Classes

class Customer {  
 private int id;  
 private String name;  
 // constructor, getters, toString  
}  
  
class Product {  
 private String name;  
 private double price;  
 // constructor, getters, toString  
}  
  
class Order {  
 private int orderId;  
 private Customer customer;  
 private List<Product> products;  
 private LocalDate orderDate;  
 // constructor, getters, toString  
}

## 1. Top 3 Customers by Total Purchase Amount

Map<Customer, Double> customerTotals = orders.stream()  
 .collect(Collectors.groupingBy(  
 Order::getCustomer,  
 Collectors.summingDouble(order ->   
 order.getProducts().stream()  
 .mapToDouble(Product::getPrice)  
 .sum()  
 )  
 ));  
  
List<Customer> topCustomers = customerTotals.entrySet().stream()  
 .sorted(Map.Entry.<Customer, Double>comparingByValue().reversed())  
 .limit(3)  
 .map(Map.Entry::getKey)  
 .collect(Collectors.toList());

## 2. Daily Sales Summary (Total Revenue Per Date)

Map<LocalDate, Double> dailySales = orders.stream()  
 .collect(Collectors.groupingBy(  
 Order::getOrderDate,  
 Collectors.summingDouble(order ->   
 order.getProducts().stream()  
 .mapToDouble(Product::getPrice)  
 .sum()  
 )  
 ));

## 3. Group Orders by Customer and Summarize

Map<Customer, SummaryStatistics> customerOrderSummary = orders.stream()  
 .collect(Collectors.groupingBy(  
 Order::getCustomer,  
 Collector.of(  
 SummaryStatistics::new,  
 (stats, order) -> {  
 double total = order.getProducts().stream().mapToDouble(Product::getPrice).sum();  
 int count = order.getProducts().size();  
 stats.add(total, count);  
 },  
 SummaryStatistics::combine  
 )  
 ));  
  
class SummaryStatistics {  
 private double totalAmount;  
 private int itemCount;  
  
 public void add(double amount, int count) {  
 this.totalAmount += amount;  
 this.itemCount += count;  
 }  
  
 public SummaryStatistics combine(SummaryStatistics other) {  
 this.totalAmount += other.totalAmount;  
 this.itemCount += other.itemCount;  
 return this;  
 }  
}

## 4. Unique Product Names Sold

Set<String> uniqueProducts = orders.stream()  
 .flatMap(order -> order.getProducts().stream())  
 .map(Product::getName)  
 .collect(Collectors.toSet());

## 5. Customers Who Ordered on Multiple Days

Map<Customer, Set<LocalDate>> customerOrderDates = orders.stream()  
 .collect(Collectors.groupingBy(  
 Order::getCustomer,  
 Collectors.mapping(Order::getOrderDate, Collectors.toSet())  
 ));  
  
List<Customer> frequentBuyers = customerOrderDates.entrySet().stream()  
 .filter(entry -> entry.getValue().size() > 1)  
 .map(Map.Entry::getKey)  
 .collect(Collectors.toList());

## Highlights

- Use of groupingBy, mapping, flatMap, and custom collectors.  
- Handles nested collections (List inside List).  
- Real-world reporting via aggregation and summarization.