# UNIQUE REAL-WORLD USE CASE

**“E-Commerce Order Processing Pipeline”**

---

# Order Class (simple model)

```
class Order {
    String customer;
    double amount;
    String status; // "DELIVERED", "PENDING", "CANCELLED"
    List<String> items;

    public Order(String customer, double amount, String status,
List<String> items) {
        this.customer = customer;
        this.amount = amount;
        this.status = status;
        this.items = items;
    }

    public String getCustomer() { return customer; }
    public double getAmount() { return amount; }
    public String getStatus() { return status; }
    public List<String> getItems() { return items; }
}
```

---

# Order List (sample data)

```
List<Order> orders = Arrays.asList(
    new Order("Rupali", 5000, "DELIVERED", Arrays.asList("Shoes", "Bag")),
    new Order("Neha", 3000, "PENDING", Arrays.asList("Watch")),
    new Order("Riya", 8000, "DELIVERED", Arrays.asList("Laptop", "Mouse")),
    new Order("Shloka", 2000, "CANCELLED", Arrays.asList("T-shirt")),
    new Order("Rupali", 1500, "DELIVERED", Arrays.asList("Wallet"))
);
```

---

# NOW — ALL STREAM OPERATIONS (FULL PIPELINE EXAMPLE)

---

# 1) filter() — Only delivered orders

```
orders.stream()
```

```
      .filter(o -> o.getStatus().equals("DELIVERED"))
```

---

# □ 2) sorted() — Highest amount first

```
      .sorted((a,b) -> Double.compare(b.getAmount(), a.getAmount()))
```

---

# □ 3) map() — extract customer names

```
      .map(Order::getCustomer)
```

---

# □ 4) distinct() — remove duplicate customer names

```
      .distinct()
```

---

# □ 5) limit() — take only top 2 customers

```
      .limit(2)
```

---

# □ 6) collect() — convert to List

```
      .collect(Collectors.toList());
```

---

# □ Final Pipeline:

```
List<String> topCustomers =
    orders.stream()
          .filter(o -> o.getStatus().equals("DELIVERED"))
          .sorted((a,b) -> Double.compare(b.getAmount(), a.getAmount()))
          .map(Order::getCustomer)
          .distinct()
          .limit(2)
          .collect(Collectors.toList());
```

**Output:**

```
["Riya", "Rupali"]
```

---

# □ EXTRA USE CASES (all remaining operations)

---

### □ 7) flatMap() — Extract ALL ITEMS purchased across all orders

```
List<String> allItems =
    orders.stream()
          .flatMap(o -> o.getItems().stream())
          .collect(Collectors.toList());
```

---

### □ 8) skip() — Skip 1st order

```
orders.stream().skip(1)
```

---

### □ 9) reduce() — Total revenue

```
double totalRevenue =
    orders.stream()
          .map(Order::getAmount)
          .reduce(0.0, (a,b) -> a+b);
```

---

### □ 10) groupingBy() — Group by status

```
Map<String, List<Order>> group =
    orders.stream()
          .collect(Collectors.groupingBy(Order::getStatus));
```

---

### □ 11) counting() — Count orders per status

```
Map<String, Long> statusCount =
    orders.stream()
          .collect(Collectors.groupingBy(Order::getStatus,
Collectors.counting()));
```

---

### □ 12) summingInt() — Total spent per customer

```
Map<String, Double> totalPerCustomer =
    orders.stream()
          .collect(Collectors.groupingBy(
              Order::getCustomer,
              Collectors.summingDouble(Order::getAmount)
          ));
```

---

### □ 13) maxBy() — Highest order per customer

```
Map<String, Optional<Order>> highestOrder =
    orders.stream()
          .collect(Collectors.groupingBy(
              Order::getCustomer,

Collectors.maxBy(Comparator.comparingDouble(Order::getAmount))
```

```
        ));
```

---

## 14) findFirst() — first delivered order

```
orders.stream()
      .filter(o -> o.getStatus().equals("DELIVERED"))
      .findFirst();
```

---

## 15) findAny() — any cancelled order (parallel fast)

```
orders.parallelStream()
      .filter(o -> o.getStatus().equals("CANCELLED"))
      .findAny();
```

---

## 16) anyMatch() — any order > 10k

```
orders.stream().anyMatch(o -> o.getAmount() > 10000);
```

---

## 17) noneMatch() — check no order has negative amount

```
orders.stream().noneMatch(o -> o.getAmount() < 0);
```

---

## 18) allMatch() — all are valid amounts

```
orders.stream().allMatch(o -> o.getAmount() >= 0);
```