Stream API

By Hatef Palizgar

Task 1 - Filtering and Mapping a List

You have a list of `Employee` objects, each containing their `name`, `salary`, and `department`. You need to find all employees whose `salary` is greater than 5000 and belong to the `"Sales"` department. Then, you need to map each of these employees to their `name` as a string and store the result in a new list.

Write a Java method that takes in a `List<Employee>` and returns a `List<String>` that contains the `names` of all employees who belong to the `"Sales"` department and have a `salary` greater than 5000.

Unit Test:

```
@Test
void testFilterAndMap() {
    Employee e1 = new Employee("John", 7000, "Sales");
    Employee e2 = new Employee("Mary", 5500, "Sales");
    Employee e3 = new Employee("Mike", 4500, "Marketing");
    Employee e4 = new Employee("David", 6000, "Sales");
    List<Employee> employees = Arrays.asList(e1, e2, e3, e4);

    List<String> expected = Arrays.asList("John", "Mary",
"David");
    List<String> actual = filterAndMap(employees);

    assertEquals(expected, actual);
}
```

Task 2: Grouping and Counting by a Field

You have a list of `Employee` objects, each containing their `name`, `salary`, and `department`. You need to group the employees by their `department` and count the number of employees in each department.

Write a Java method that takes in a `List<Employee>` and returns a `Map<String, Long>` where the key is the name of the department and the value is the number of employees in that department.

Unit test:

```
@Test
void testGroupAndCount() {
    Employee e1 = new Employee("John", 7000, "Sales");
    Employee e2 = new Employee("Mary", 5500, "Sales");
    Employee e3 = new Employee("Mike", 4500, "Marketing");
    Employee e4 = new Employee("David", 6000, "Sales");
    List<Employee> employees = Arrays.asList(e1, e2, e3, e4);

    Map<String, Long> expected = new HashMap<>();
    expected.put("Sales", 3L);
    expected.put("Marketing", 1L);
    Map<String, Long> actual = groupAndCount(employees);

    assertEquals(expected, actual);
}
```

Task 3: Reducing a List to a Single Object

You have a list of `Student` objects, each containing their `name`, `age`, and `grade`. You need to find the student with the highest `grade` and return their `name` as a string.

Write a Java method that takes in a `List<Student>` and returns a string containing the `name` of the student with the highest `grade`.

Unit test:

```
@Test
void testFindTopStudent() {
    Student s1 = new Student("John", 18, 85);
    Student s2 = new Student("Mary", 19, 90);
    Student s3 = new Student("Mike", 20, 80);
    Student s4 = new Student("David", 18, 95);
    List<Student> students = Arrays.asList(s1, s2, s3, s4);

    String expected = "David";
    String actual = findTopStudent(students);

    assertEquals(expected, actual);
}
```

Task 4: Sorting a List

You have a list of `Product` objects, each containing their `name`, `price`, and `quantity` in stock. You need to sort the products by their `price` in descending order, then by their `name` in ascending order, and return a new sorted list.

Write a Java method that takes in a `List<Product>` and returns a new `List<Product>` containing the sorted products.

Unit test:

```
@Test
void testSortProducts() {
    Product p1 = new Product("Laptop", 1200.00, 10);
    Product p2 = new Product("Mouse", 25.00, 50);
    Product p3 = new Product("Keyboard", 50.00, 20);
    Product p4 = new Product("Monitor", 300.00, 5);
    List<Product> products = Arrays.asList(p1, p2, p3, p4);

    List<Product> expected = Arrays.asList(p1, p4, p3, p2);
    List<Product> actual = sortProducts(products);

    assertEquals(expected, actual);
}
```

Task 5: Grouping and Counting

You have a list of `Transaction` objects, each containing their `id`, `type`, and `amount`. You need to group the transactions by their `type` (“credit” or “debit”), count the number of transactions for each `type`, and return a `Map<String, Long>` containing the `type` as the key and the count as the value.

Write a Java method that takes in a `List<Transaction>` and returns a `Map<String, Long>` containing the transaction `types` as the keys and the count of transactions for each type as the values.

Unit test:

```
@Test
void testGroupTransactionsByType() {
    Transaction t1 = new Transaction(1, "Credit", 100.00);
    Transaction t2 = new Transaction(2, "Debit", 50.00);
    Transaction t3 = new Transaction(3, "Credit", 75.00);
    Transaction t4 = new Transaction(4, "Credit", 200.00);
    Transaction t5 = new Transaction(5, "Debit", 20.00);
    List<Transaction> transactions = Arrays.asList(t1, t2, t3, t4,
t5);
    Map<String, Long> expected = new HashMap<>();
    expected.put("Credit", 3L);
    expected.put("Debit", 2L);
    Map<String, Long> actual =
groupTransactionsByType(transactions);

    assertEquals(expected, actual);
}
```

Task 6: Filtering and Mapping

You have a list of `Order` objects, each containing their `id`, `items`, and total `price`. You need to filter out the `orders` where the total `price` is less than $50, then map the remaining orders to their `id` and total `price`, and return a new list of `OrderSummary` objects.

Write a Java method that takes in a `List<Order>` and returns a `List<OrderSummary>` containing the filtered and mapped orders.

Hint: Below is the `OrderSummary` object code:

```
public class OrderSummary {
    private int id;
    private double totalPrice;

    // constructor

    // getters and setters and toString()

 }
```

Unit test:👇

```
@Test
void testFilterAndMapOrders() {
        List<String> items1 = Arrays.asList("Laptop", "Mouse",
"Keyboard");
        List<String> items2 = Arrays.asList("Monitor", "Speakers",
"Headphones");
        List<String> items3 = Arrays.asList("USB Drive", "External
Hard Drive", "Flash Drive");
        List<String> items4 = Arrays.asList("Smartphone",
"Tablet");
        Order o1 = new Order(1, items1, 1200.00);
        Order o2 = new Order(2, items2, 250.00);
        Order o3 = new Order(3, items3, 40.00);
        Order o4 = new Order(4, items4, 600.00);
        List<Order> orders = Arrays.asList(o1, o2, o3, o4);

        List<OrderSummary> expected = Arrays.asList(
```

```
                new OrderSummary(1, 1200.00),
                new OrderSummary(2, 250.00),
                new OrderSummary(4, 600.00)
        );
        List<OrderSummary> actual = filterAndMapOrders(orders);

        assertEquals(expected.size(), actual.size());
        for (int i = 0; i < expected.size(); i++) {
            assertEquals(expected.get(i).getId(),
actual.get(i).getId());
            assertEquals(expected.get(i).getTotalPrice(),
actual.get(i).getTotalPrice(), 0.001);
        }
}
```

Task 7: User Statistics

You have a list of `User` objects that contain a `name`, `age`, and a `list` of friends(list of strings). Your task is to find the top 3 users with the most number of friends, and return their names in a list. Implement this using Java Streams.

Unit test:

```
@Test
void testGetTopUsersWithMostFriends() {
        User alice = new User("Alice", 20, List.of());
        User bob = new User("Bob", 22, List.of(alice));
        User carol = new User("Carol", 25, Arrays.asList(alice, bob));
        User dave = new User("Dave", 26, Arrays.asList(bob, carol));
        User eve = new User("Eve", 28, List.of(dave, bob, carol));

        List<User> users = Arrays.asList(alice, bob, carol, dave, eve);
        List<String> actual = getTopUsersWithMostFriends(users);

        List<String> expected = Arrays.asList("Carol", "Dave", "Eve");
        assertEquals(expected.size(), actual.size());
        assertTrue(actual.contains("Carol"));
        assertTrue(actual.contains("Dave"));
        assertTrue(actual.contains("Eve"));

}
```

Task 8: Parallel Stream Processing

You have a large list of integers and you need to compute the sum of their squares. Implement this using parallel streams to take advantage of multi-core processors and speed up the computation.

Unit test:

```
@Test
void testSumOfSquares() {
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);

  int expected = 385; // 1 + 4 + 9 + ... + 100
  int result = parallelStreamProcess(numbers);

  assertEquals(expected, result);
}
```

Task 9: FlatMap

You have a list of orders placed by customers, and each order contains a list of items. Your task is to create a stream of all the items in all the orders.

Class definitions:

```
public class Order {
    private int orderId;
    private List<Item> items;

    // constructor, getters, and setters
}

public class Item {
    private int itemId;
    private String name;
    private int quantity;
    private double price;

    // constructor, getters, and setters
}
```

Unit test: 👇

```
@Test
public void testFindAllItems() {
   Item item1 = new Item(1, "item1", 2, 5.0);
   Item item2 = new Item(2, "item2", 3, 7.0);
   Item item3 = new Item(3, "item3", 1, 10.0);
   Item item4 = new Item(4, "item4", 4, 3.0);

   List<Order> orders = Arrays.asList(
           new Order(1, Arrays.asList(item1, item2)),
           new Order(2, Arrays.asList(item3, item4))
   );

   List<Item> expectedItems = Arrays.asList(item1, item2, item3,
item4);
```

```
    List<Item> actualItems = findAllItems(orders);

    assertIterableEquals(expectedItems, actualItems);
}
```

Task 10: Filtering and Mapping Nested Collections

You have a list of `Person` objects, each of which has a name and a list of addresses. Each `address` has a street, a city, and a state. Your task is to create a list of all the cities where people with the name "John" live, using Java Streams.

Unit test:

```
@Test
void testFindCitiesOfJohns() {
  Address john1Address1 = new Address("123 Main St", "New York", "NY");
  Address john1Address2 = new Address("456 Elm St", "Boston", "MA");
  Person john1 = new Person("John", Arrays.asList(john1Address1,
john1Address2));

  Address john2Address1 = new Address("789 Oak St", "Chicago", "IL");
  Address john2Address2 = new Address("321 Maple St", "San Francisco", "CA");
  Person john2 = new Person("John", Arrays.asList(john2Address1,
john2Address2));

  Person jane = new Person("Jane", List.of(new Address("111 Pine St", "Seattle",
"WA")));

  List<Person> people = Arrays.asList(john1, john2, jane);

  List<String> expected = Arrays.asList("New York", "Boston", "Chicago", "San
Francisco");
  List<String> result = findCitiesOfJohns(people);

  assertEquals(expected, result);
}
```