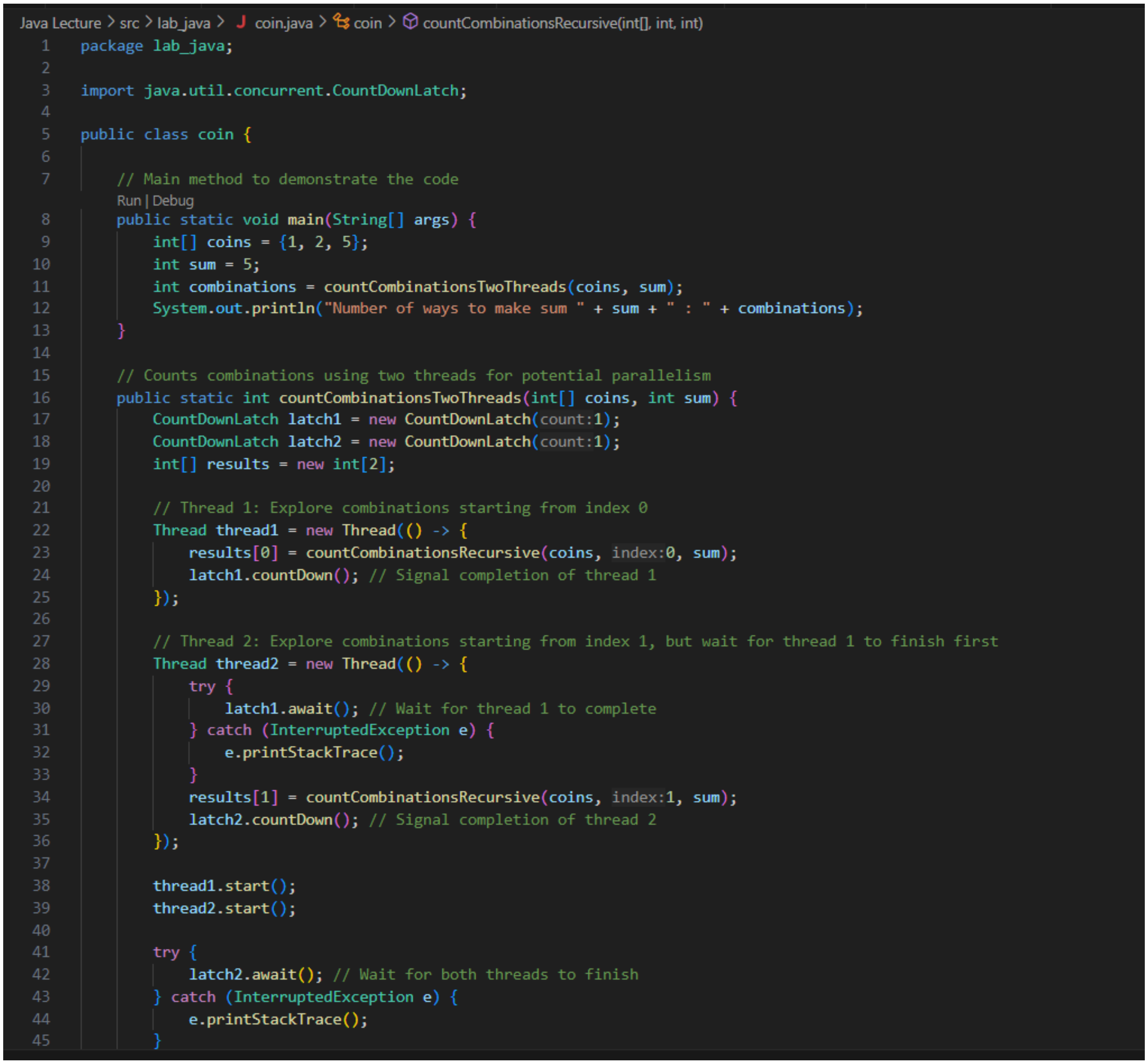
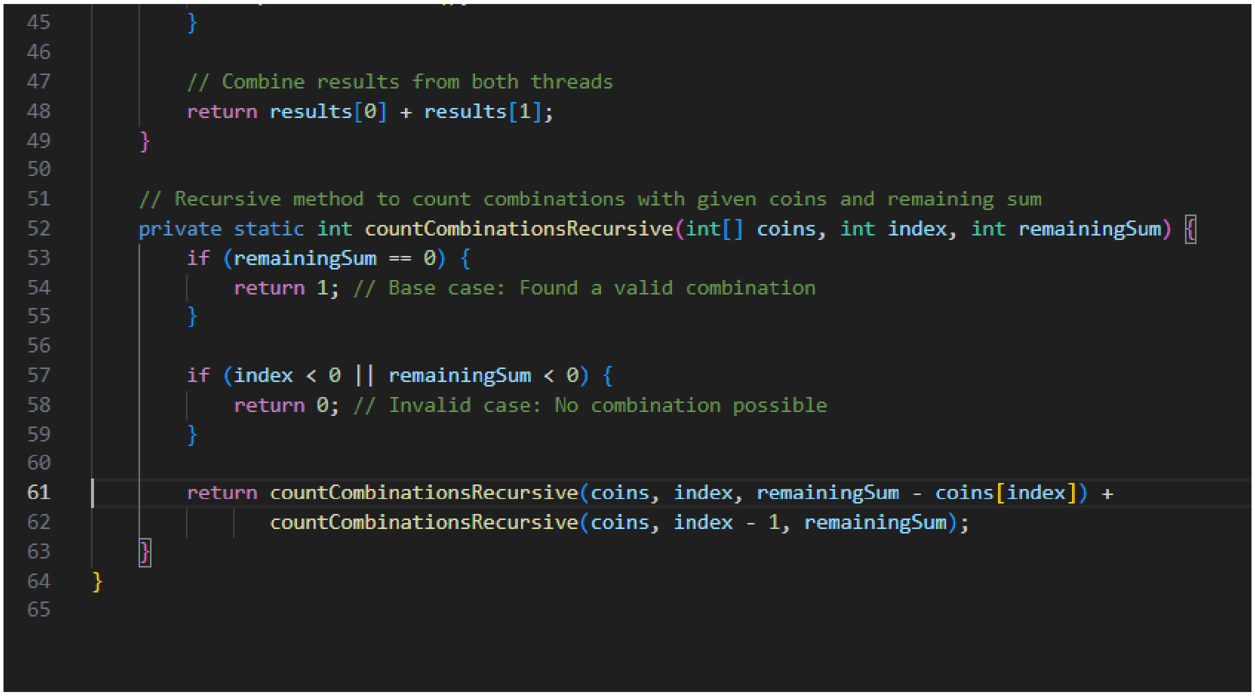
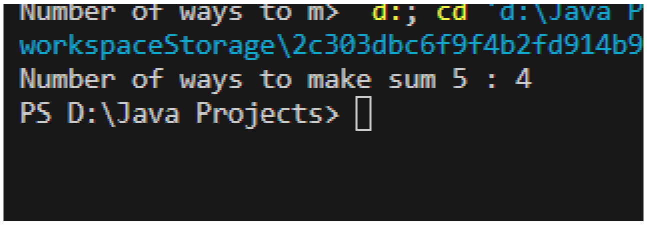
Program 01:


Java Lecture > src > lab_java > coin.java > coin > countCombinationsRecursive(int[], int, int)
Run | Debug

```java
package lab_java;

import java.util.concurrent.CountDownLatch;

public class coin {

    // Main method to demonstrate the code
    public static void main(String[] args) {
        int[] coins = {1, 2, 5};
        int sum = 5;
        int combinations = countCombinationsTwoThreads(coins, sum);
        System.out.println("Number of ways to make sum " + sum + " : " + combinations);
    }

    // Counts combinations using two threads for potential parallelism
    public static int countCombinationsTwoThreads(int[] coins, int sum) {
        CountDownLatch latch1 = new CountDownLatch(count:1);
        CountDownLatch latch2 = new CountDownLatch(count:1);
        int[] results = new int[2];

        // Thread 1: Explore combinations starting from index 0
        Thread thread1 = new Thread(() -> {
            results[0] = countCombinationsRecursive(coins, index:0, sum);
            latch1.countDown(); // Signal completion of thread 1
        });

        // Thread 2: Explore combinations starting from index 1, but wait for thread 1 to finish first
        Thread thread2 = new Thread(() -> {
            try {
                latch1.await(); // Wait for thread 1 to complete
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            results[1] = countCombinationsRecursive(coins, index:1, sum);
            latch2.countDown(); // Signal completion of thread 2
        });

        thread1.start();
        thread2.start();

        try {
            latch2.await(); // Wait for both threads to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

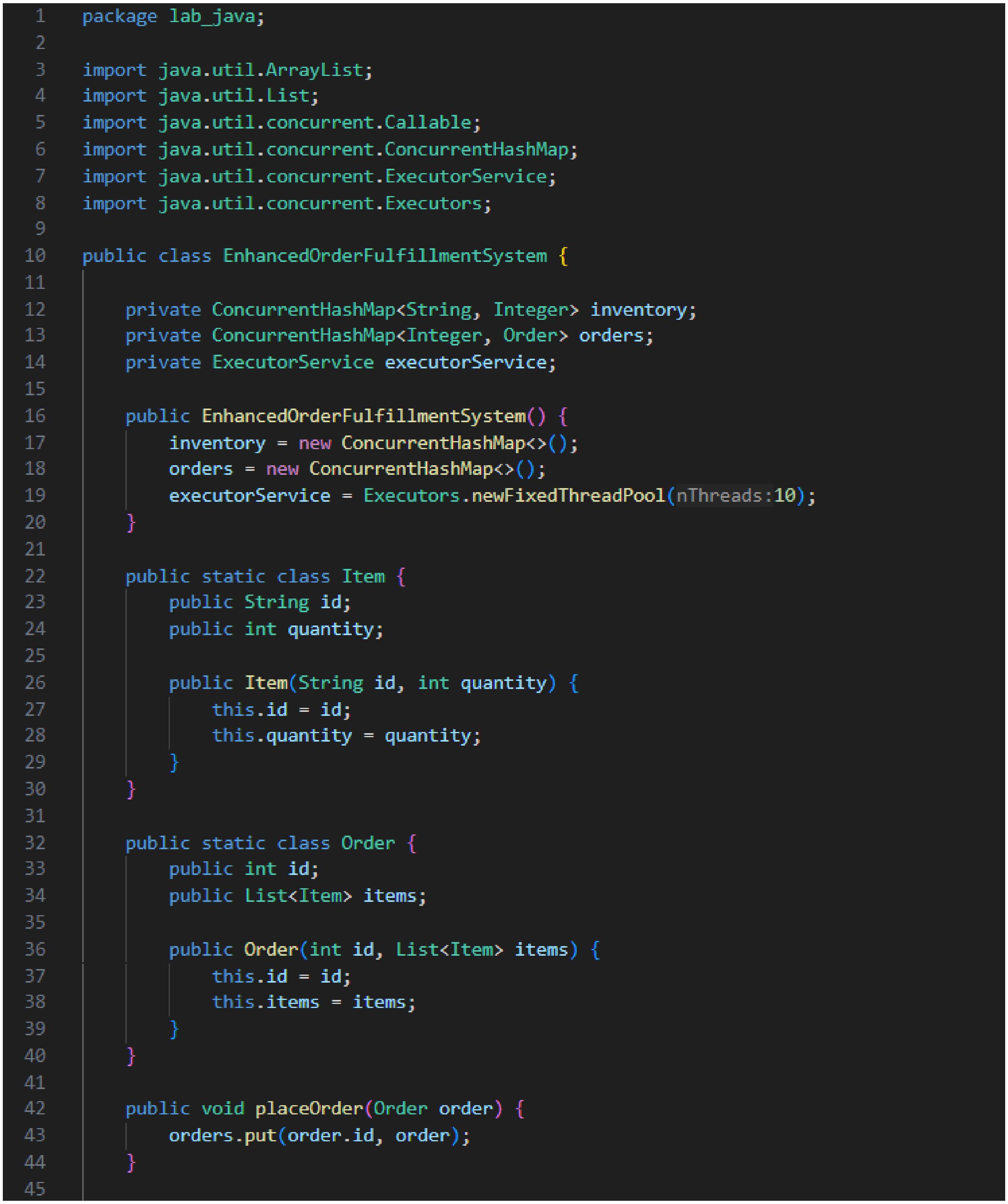


```java
        }

        // Combine results from both threads
        return results[0] + results[1];
    }

    // Recursive method to count combinations with given coins and remaining sum
    private static int countCombinationsRecursive(int[] coins, int index, int remainingSum) {
        if (remainingSum == 0) {
            return 1; // Base case: Found a valid combination
        }

        if (index < 0 || remainingSum < 0) {
            return 0; // Invalid case: No combination possible
        }

        return countCombinationsRecursive(coins, index, remainingSum - coins[index]) +
               countCombinationsRecursive(coins, index - 1, remainingSum);
    }
}
```

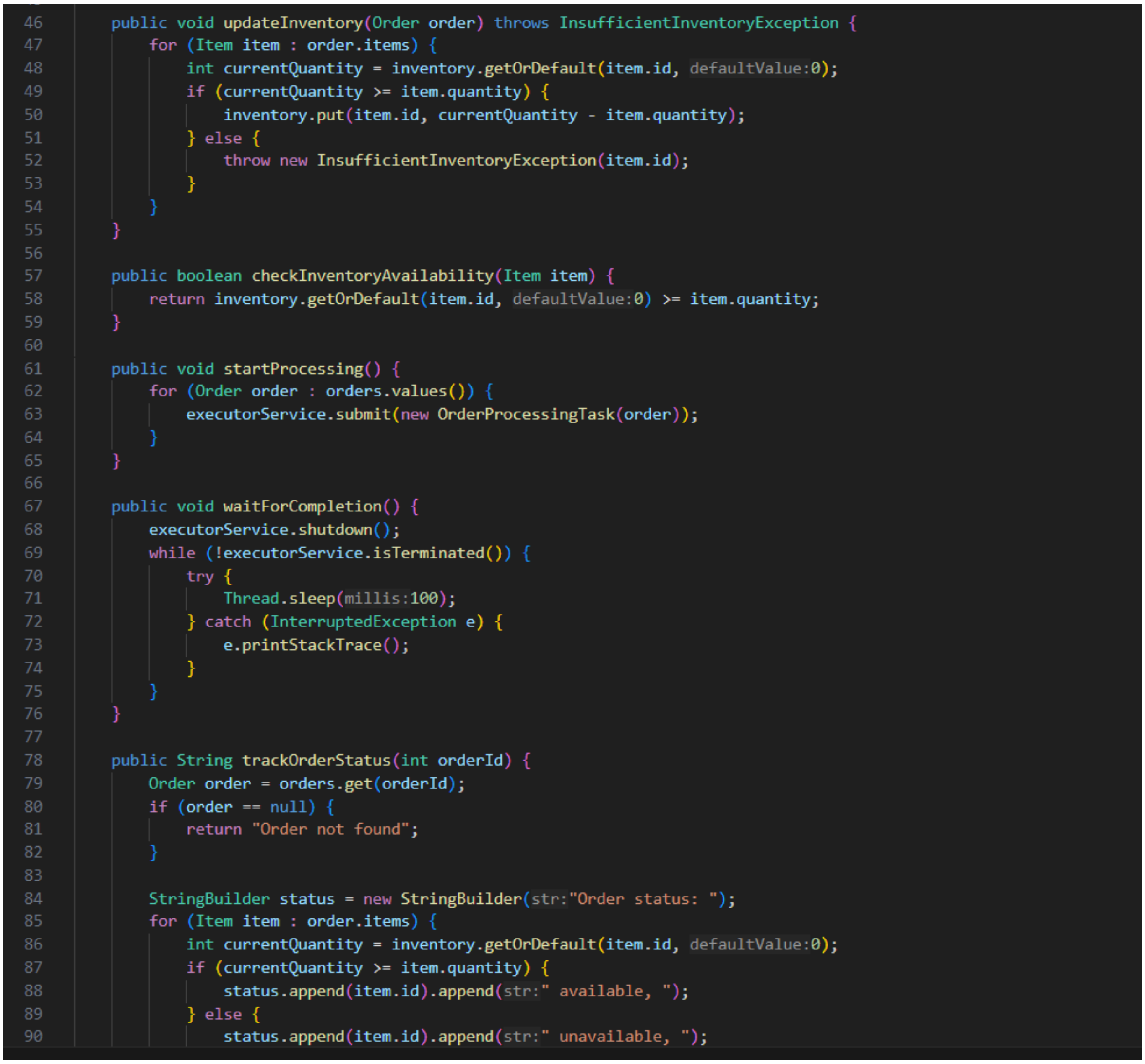


```
Number of ways to m>  d:; cd 'd:\Java P
workspaceStorage\2c303dbc6f9f4b2fd914b9
Number of ways to make sum 5 : 4
PS D:\Java Projects>
```

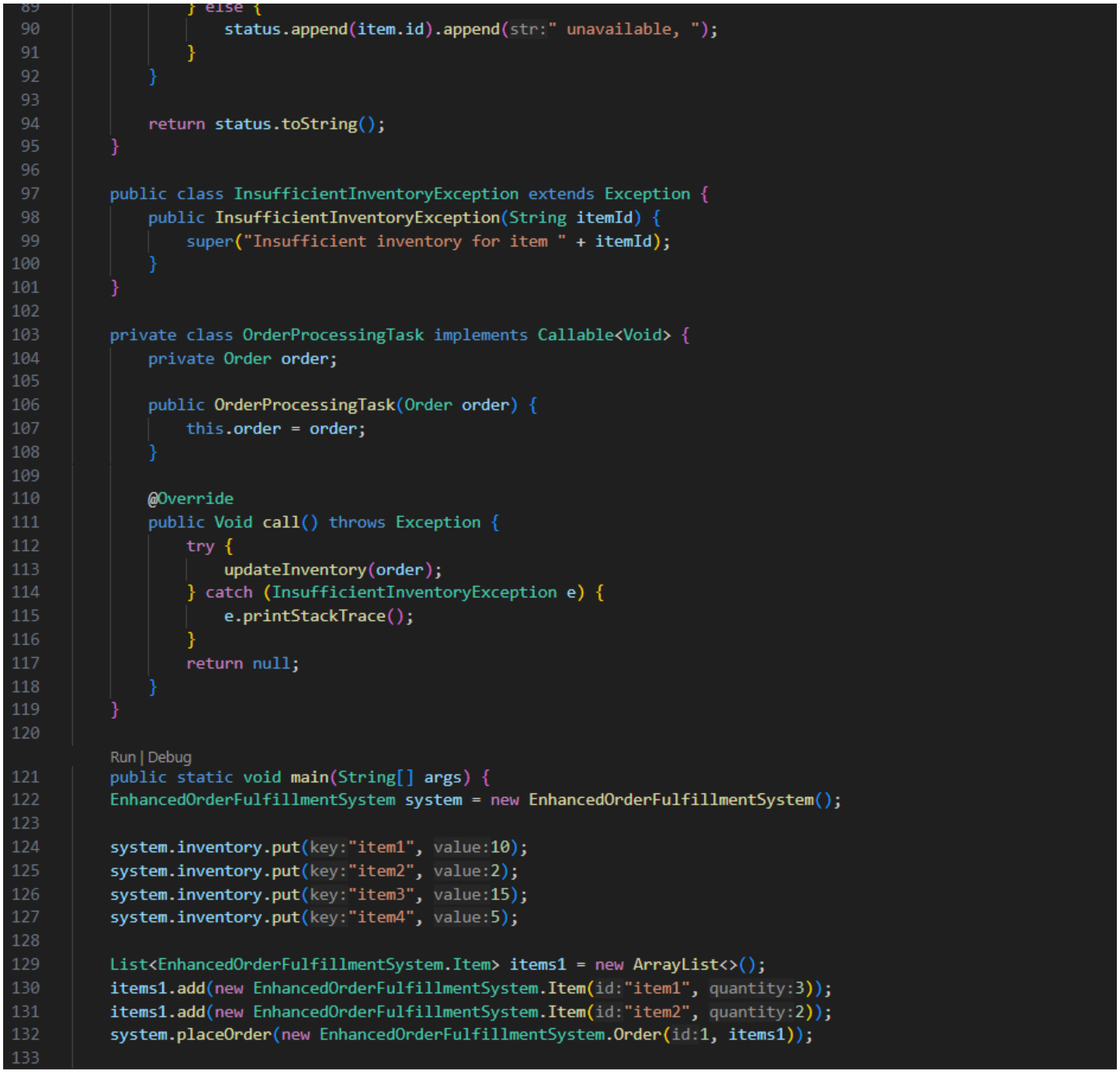
Program 02:



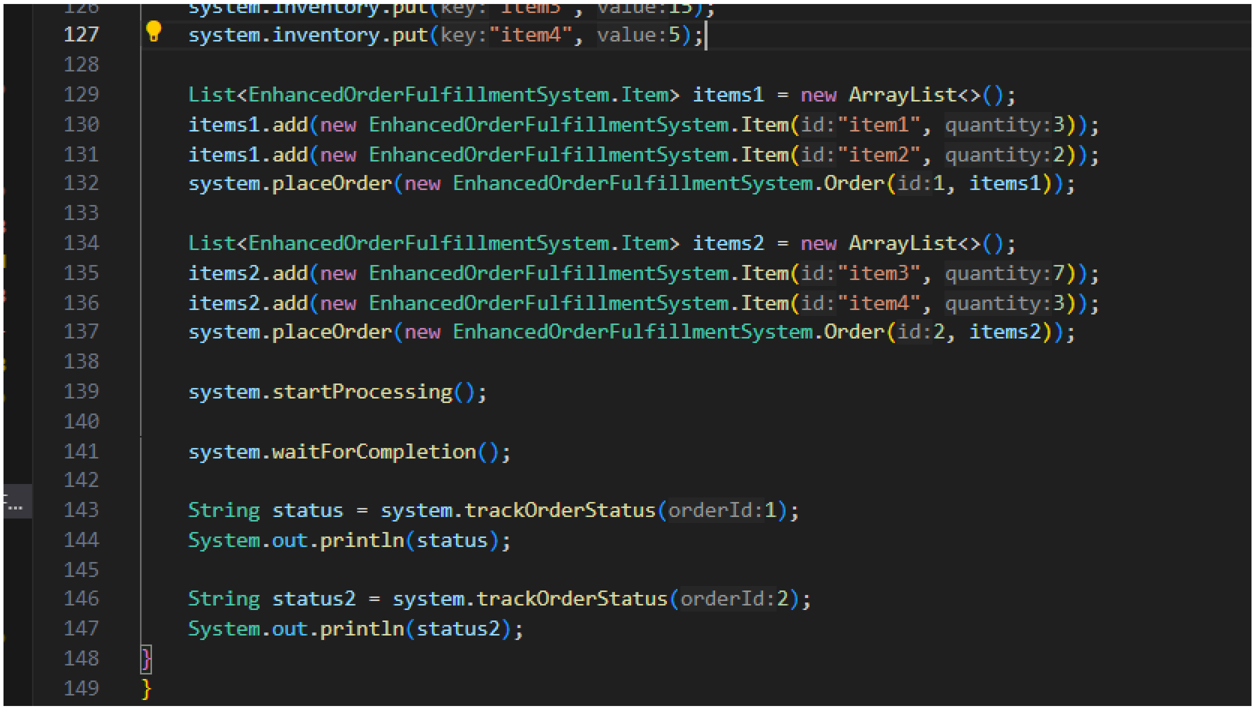
```java
package lab_java;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class EnhancedOrderFulfillmentSystem {

    private ConcurrentHashMap<String, Integer> inventory;
    private ConcurrentHashMap<Integer, Order> orders;
    private ExecutorService executorService;

    public EnhancedOrderFulfillmentSystem() {
        inventory = new ConcurrentHashMap<>();
        orders = new ConcurrentHashMap<>();
        executorService = Executors.newFixedThreadPool(nThreads:10);
    }

    public static class Item {
        public String id;
        public int quantity;

        public Item(String id, int quantity) {
            this.id = id;
            this.quantity = quantity;
        }
    }

    public static class Order {
        public int id;
        public List<Item> items;

        public Order(int id, List<Item> items) {
            this.id = id;
            this.items = items;
        }
    }

    public void placeOrder(Order order) {
        orders.put(order.id, order);
    }

```



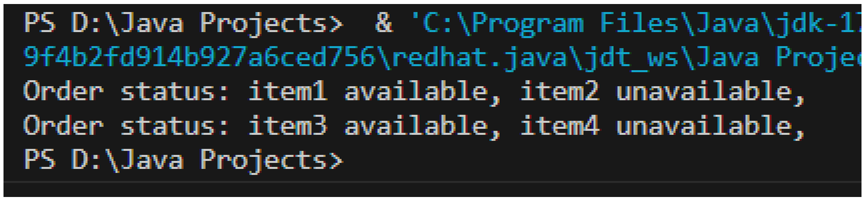
```
    public void updateInventory(Order order) throws InsufficientInventoryException {
        for (Item item : order.items) {
            int currentQuantity = inventory.getOrDefault(item.id, defaultValue:0);
            if (currentQuantity >= item.quantity) {
                inventory.put(item.id, currentQuantity - item.quantity);
            } else {
                throw new InsufficientInventoryException(item.id);
            }
        }
    }

    public boolean checkInventoryAvailability(Item item) {
        return inventory.getOrDefault(item.id, defaultValue:0) >= item.quantity;
    }

    public void startProcessing() {
        for (Order order : orders.values()) {
            executorService.submit(new OrderProcessingTask(order));
        }
    }

    public void waitForCompletion() {
        executorService.shutdown();
        while (!executorService.isTerminated()) {
            try {
                Thread.sleep(millis:100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public String trackOrderStatus(int orderId) {
        Order order = orders.get(orderId);
        if (order == null) {
            return "Order not found";
        }

        StringBuilder status = new StringBuilder(str:"Order status: ");
        for (Item item : order.items) {
            int currentQuantity = inventory.getOrDefault(item.id, defaultValue:0);
            if (currentQuantity >= item.quantity) {
                status.append(item.id).append(str:" available, ");
            } else {
                status.append(item.id).append(str:" unavailable, ");
```



```
            } else {
                status.append(item.id).append(str:" unavailable, ");
            }
        }

        return status.toString();
    }

    public class InsufficientInventoryException extends Exception {
        public InsufficientInventoryException(String itemId) {
            super("Insufficient inventory for item " + itemId);
        }
    }

    private class OrderProcessingTask implements Callable<Void> {
        private Order order;

        public OrderProcessingTask(Order order) {
            this.order = order;
        }

        @Override
        public Void call() throws Exception {
            try {
                updateInventory(order);
            } catch (InsufficientInventoryException e) {
                e.printStackTrace();
            }
            return null;
        }
    }

    Run | Debug
    public static void main(String[] args) {
    EnhancedOrderFulfillmentSystem system = new EnhancedOrderFulfillmentSystem();

    system.inventory.put(key:"item1", value:10);
    system.inventory.put(key:"item2", value:2);
    system.inventory.put(key:"item3", value:15);
    system.inventory.put(key:"item4", value:5);

    List<EnhancedOrderFulfillmentSystem.Item> items1 = new ArrayList<>();
    items1.add(new EnhancedOrderFulfillmentSystem.Item(id:"item1", quantity:3));
    items1.add(new EnhancedOrderFulfillmentSystem.Item(id:"item2", quantity:2));
    system.placeOrder(new EnhancedOrderFulfillmentSystem.Order(id:1, items1));

```



```
    system.inventory.put(key:"item3", value:15);
    system.inventory.put(key:"item4", value:5);

    List<EnhancedOrderFulfillmentSystem.Item> items1 = new ArrayList<>();
    items1.add(new EnhancedOrderFulfillmentSystem.Item(id:"item1", quantity:3));
    items1.add(new EnhancedOrderFulfillmentSystem.Item(id:"item2", quantity:2));
    system.placeOrder(new EnhancedOrderFulfillmentSystem.Order(id:1, items1));

    List<EnhancedOrderFulfillmentSystem.Item> items2 = new ArrayList<>();
    items2.add(new EnhancedOrderFulfillmentSystem.Item(id:"item3", quantity:7));
    items2.add(new EnhancedOrderFulfillmentSystem.Item(id:"item4", quantity:3));
    system.placeOrder(new EnhancedOrderFulfillmentSystem.Order(id:2, items2));

    system.startProcessing();

    system.waitForCompletion();

    String status = system.trackOrderStatus(orderId:1);
    System.out.println(status);

    String status2 = system.trackOrderStatus(orderId:2);
    System.out.println(status2);
}
}
```



```
PS D:\Java Projects>  & 'C:\Program Files\Java\jdk-17
9f4b2fd914b927a6ced756\redhat.java\jdt_ws\Java Proje
Order status: item1 available, item2 unavailable,
Order status: item3 available, item4 unavailable,
PS D:\Java Projects>
```