# SQA tutorial 1

Report by group 1 - tutorial class 2:

| Student ID | Name | Class |
| --- | --- | --- |
| 1901040174 | Lê Quang Sáng | 2C-19 |
| 2001040024 | Đỗ Hữu Đạt | 4C-20 |
| 2001040056 | Nguyễn Thế Trường Giang | 7C-20 |
| 2001040064 | Nguyễn Văn Hải | 7C-20 |
| 2001040088 | Nguyễn Viết Hưng | 7C-20 |
| 2001040095 | Chu Quang Huy | 6C-20 |

In Test-Driven Development (TDD), we start by writing tests before implementing the actual code. Here we used JUnit4 to create tests before implementing the code.

In the our test scenarios, multiple test cases have been combined within a single test function for the purpose of brevity and clarity in this documentation. However, in real-world software testing best practices, it is generally recommended to separate each test case into its own dedicated test function.

## Exercise 1

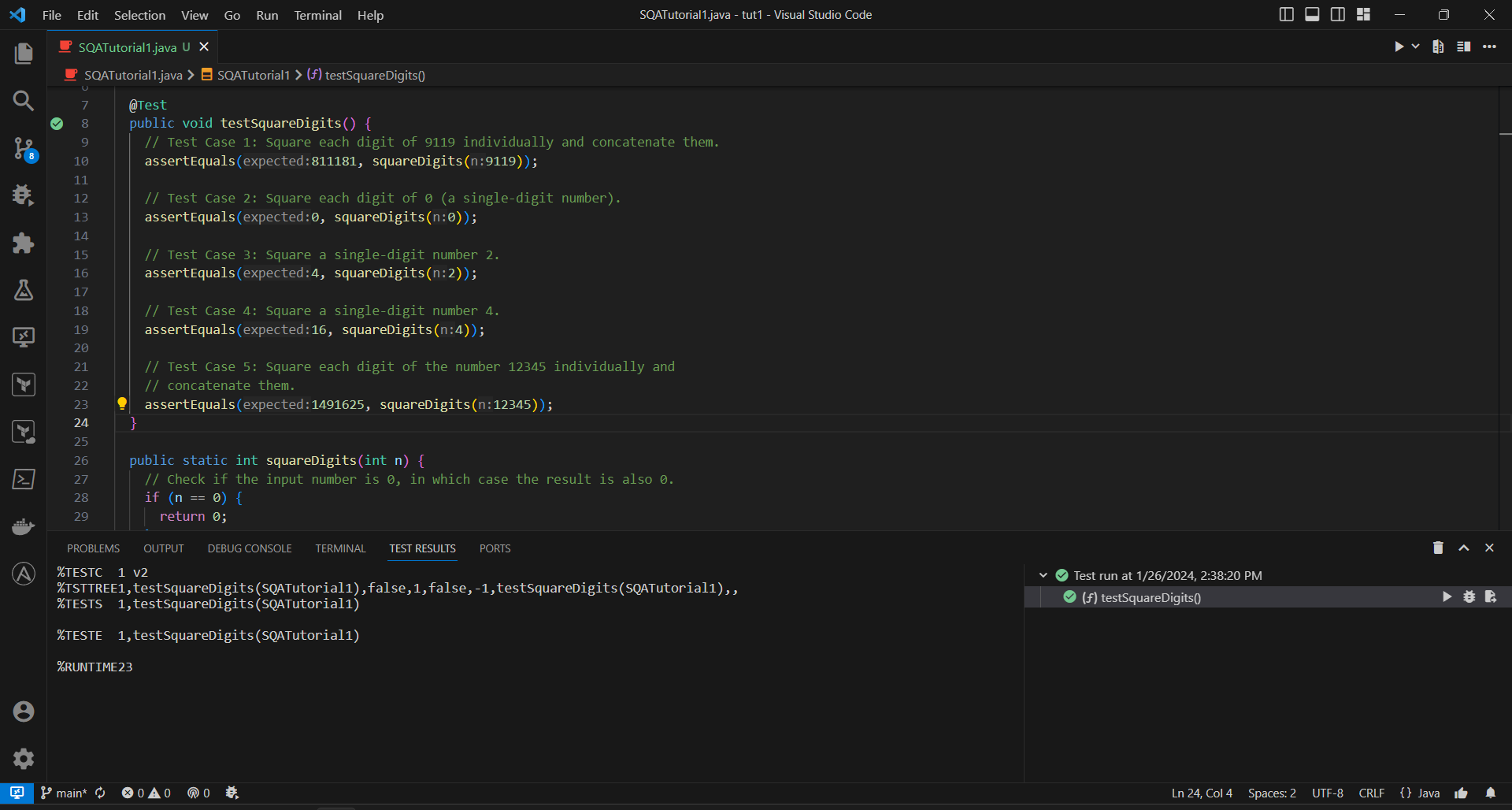
### Test cases

@Test  
public void testSquareDigits() {  
 // Test Case 1: Square each digit of 9119 individually and concatenate them.  
 assertEquals(811181, squareDigits(9119));  
  
 // Test Case 2: Square each digit of 0 (a single-digit number).  
 assertEquals(0, squareDigits(0));  
  
 // Test Case 3: Square a single-digit number 2.  
 assertEquals(4, squareDigits(2));  
  
 // Test Case 4: Square a single-digit number 4.  
 assertEquals(16, squareDigits(4));  
  
 // Test Case 5: Square each digit of the number 12345 individually and  
 // concatenate them.  
 assertEquals(1491625, squareDigits(12345));  
}

### Method implementation

public static int squareDigits(int n) {  
 // Check if the input number is 0, in which case the result is also 0.  
 if (n == 0) {  
 return 0;  
 }  
  
 String result = "";  
 while (n != 0) {  
 // Extract the last digit of the number.  
 int digit = n % 10;  
 // Square the digit and append the result to the front (left) of the string.  
 result = digit \* digit + result;  
 // Remove the last digit from the number.  
 n /= 10;  
 }  
 return Integer.parseInt(result);  
}

### Screenshot



Running test for exercise 1

## Exercise 2

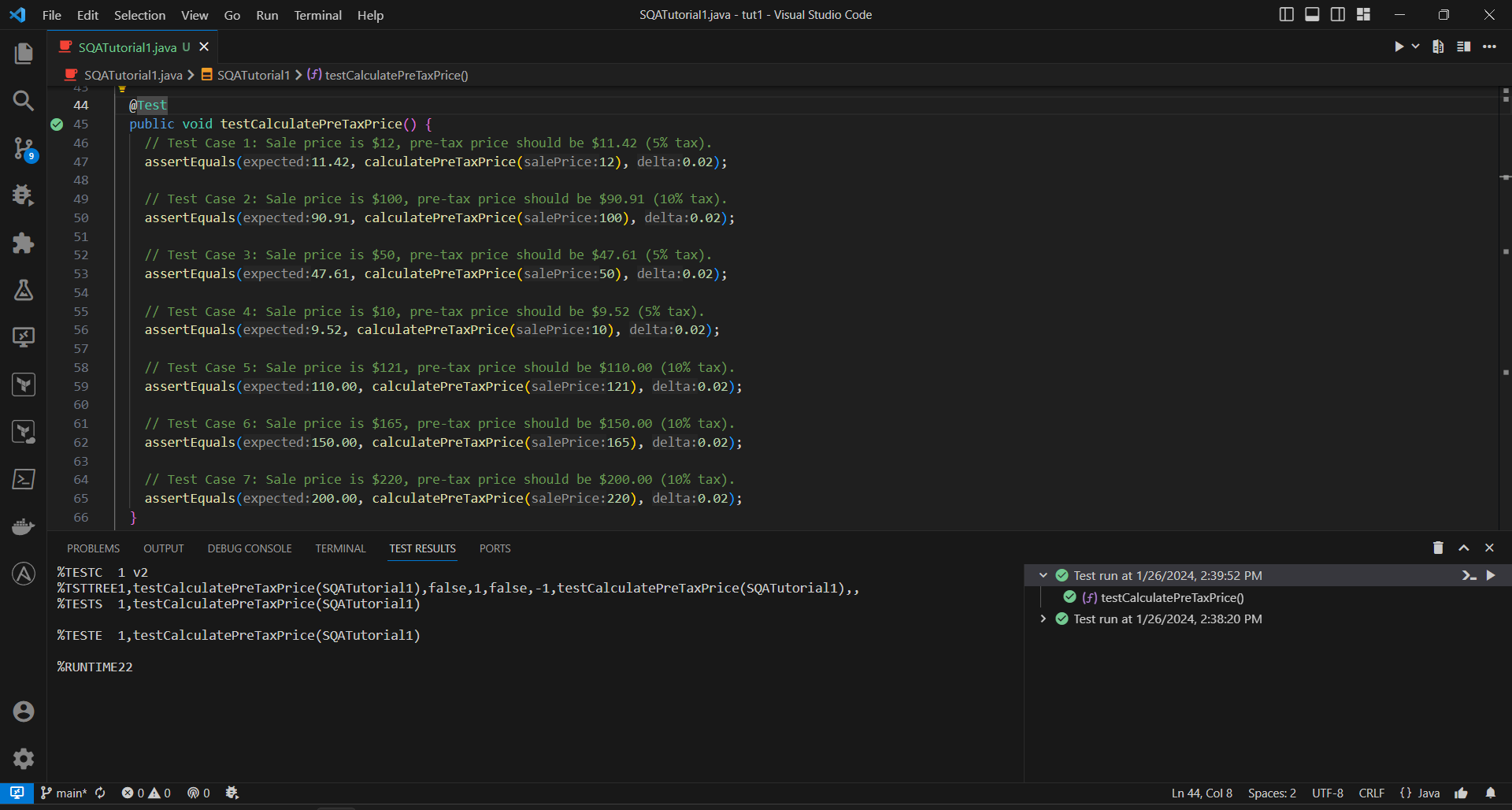
### Test cases

@Test  
public void testCalculatePreTaxPrice() {  
 // Test Case 1: Sale price is $12, pre-tax price should be $11.42 (5% tax).  
 assertEquals(11.42, calculatePreTaxPrice(12), 0.02);  
  
 // Test Case 2: Sale price is $100, pre-tax price should be $90.91 (10% tax).  
 assertEquals(90.91, calculatePreTaxPrice(100), 0.02);  
  
 // Test Case 3: Sale price is $50, pre-tax price should be $47.61 (5% tax).  
 assertEquals(47.61, calculatePreTaxPrice(50), 0.02);  
  
 // Test Case 4: Sale price is $10, pre-tax price should be $9.52 (5% tax).  
 assertEquals(9.52, calculatePreTaxPrice(10), 0.02);  
  
 // Test Case 5: Sale price is $121, pre-tax price should be $110.00 (10% tax).  
 assertEquals(110.00, calculatePreTaxPrice(121), 0.02);  
  
 // Test Case 6: Sale price is $165, pre-tax price should be $150.00 (10% tax).  
 assertEquals(150.00, calculatePreTaxPrice(165), 0.02);  
  
 // Test Case 7: Sale price is $220, pre-tax price should be $200.00 (10% tax).  
 assertEquals(200.00, calculatePreTaxPrice(220), 0.02);  
}

### Method implementation

public static double calculatePreTaxPrice(double salePrice) {  
 double taxRate = 0.05;  
 if (salePrice >= 100) {  
 taxRate = 0.10;  
 }  
  
 double preTaxPrice = salePrice / (1 + taxRate);  
 preTaxPrice = Math.floor(preTaxPrice \* 100) / 100;  
 return preTaxPrice;  
}

### Screenshot



Running test for exercise 2

## Exercise 3

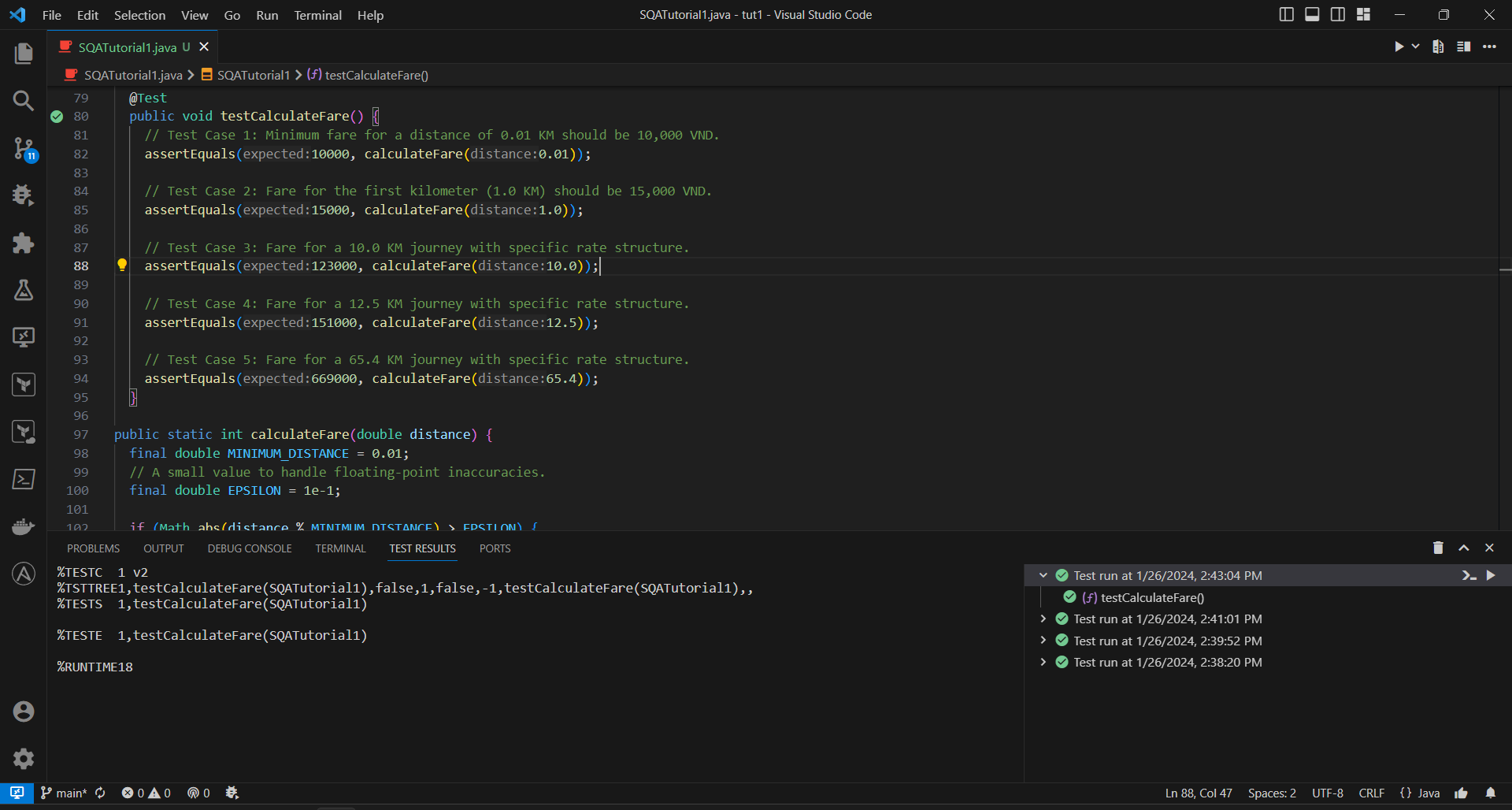
### Test cases

@Test  
public void testCalculateFare() {  
 // Test Case 1: Minimum fare for a distance of 0.01 KM should be 10,000 VND.  
 assertEquals(10000, calculateFare(0.01));  
  
 // Test Case 2: Fare for the first kilometer (1.0 KM) should be 15,000 VND.  
 assertEquals(15000, calculateFare(1.0));  
  
 // Test Case 3: Fare for a 10.0 KM journey with specific rate structure.  
 assertEquals(123000, calculateFare(10.0));  
  
 // Test Case 4: Fare for a 12.5 KM journey with specific rate structure.  
 assertEquals(151000, calculateFare(12.5));  
  
 // Test Case 5: Fare for a 65.4 KM journey with specific rate structure.  
 assertEquals(669000, calculateFare(65.4));  
}

### Method implementation

public static int calculateFare(double distance) {  
 final double MINIMUM\_DISTANCE = 0.01;  
 // A small value to handle floating-point inaccuracies.  
 final double EPSILON = 1e-1;  
  
 if (Math.abs(distance % MINIMUM\_DISTANCE) > EPSILON) {  
 // Print a message and return -1 to indicate an error.  
 System.out  
 .println(  
 "Error: The program only accepts travel distances no more accurate than 10 meters (0.01 kilometers).");  
 return -1;  
 }  
  
 double fare = 10000;  
  
 if (distance < 1) {  
 fare = 10000;  
 } else if (distance == 1.0) {  
 fare = 15000;  
 } else if (distance <= 10) {  
 fare = 15000 + 12000 \* (distance - 1);  
 } else {  
 // Calculate fare for distances beyond 10 KM with decreasing cost every 10 KM.  
 fare = 15000 + 12000 \* 9;  
 double currentCost = 11000;  
 int fullTens = (int) (distance / 10);  
 // Iterate through each 10 KM segment and adjust fare accordingly.  
 for (int i = 0; i < fullTens; i++) {  
 fare += currentCost \* (i == fullTens - 1 ? distance % 10 : 10);  
 if (currentCost > 7000) {  
 currentCost -= 500;  
 }  
 }  
 }  
  
 // Round up the calculated fare to the nearest thousand (VND).  
 return (int) Math.ceil(fare / 1000) \* 1000;  
}

### Screenshot



Running test for exercise 3

## Exercise 4

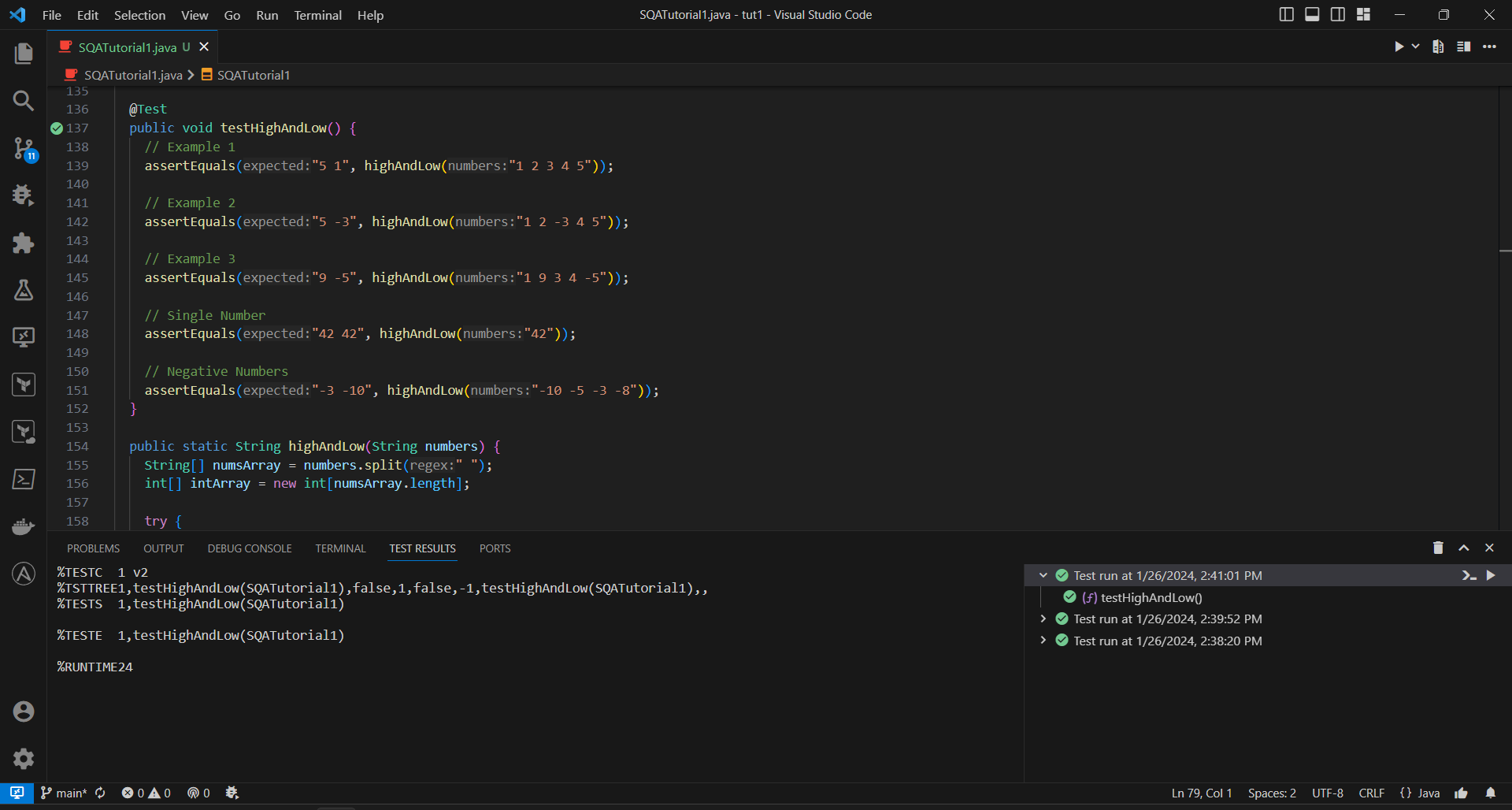
### Test cases

@Test  
public void testHighAndLow() {  
 // Example 1  
 assertEquals("5 1", highAndLow("1 2 3 4 5"));  
  
 // Example 2  
 assertEquals("5 -3", highAndLow("1 2 -3 4 5"));  
  
 // Example 3  
 assertEquals("9 -5", highAndLow("1 9 3 4 -5"));  
  
 // Single Number  
 assertEquals("42 42", highAndLow("42"));  
  
 // Negative Numbers  
 assertEquals("-3 -10", highAndLow("-10 -5 -3 -8"));  
}

### Method implementation

public static String highAndLow(String numbers) {  
 String[] numsArray = numbers.split(" ");  
 int[] intArray = new int[numsArray.length];  
  
 try {  
 for (int i = 0; i < numsArray.length; i++) {  
 intArray[i] = Integer.parseInt(numsArray[i]);  
 }  
 } catch (NumberFormatException e) {  
 // Handle the case where the input contains invalid non-integer values.  
 System.out.println("Error: invalid input");  
 return null;  
 }  
  
 // Initialize variables to keep track of the maximum and minimum values.  
 int max = intArray[0];  
 int min = intArray[0];  
  
 // Iterate through the integer array to find the maximum and minimum values.  
 for (int num : intArray) {  
 if (num > max) {  
 max = num;  
 }  
 if (num < min) {  
 min = num;  
 }  
 }  
  
 // Return a string containing the maximum and minimum values.  
 return max + " " + min;  
}

### Screenshot



Running test for exercise 4