ASSIGNMENT 1

Snippets, Explanations

Reference

Question 1

Here are 12 different test cases that test the program efficiently and cover various aspects of the input domain:

1. **Letter Test**: Test the conversion of individual Roman numeral characters.
   - Input: ["I", "X", "Z", "!"]
   - Expected output: [1, 10, 0, 0]
   - This test checks if the program can handle individual characters correctly, including both valid and invalid characters.

2. **Additive Combinations Test**: Test the conversion of additive combinations of Roman numerals.
   - Input: ["III", "LXVII", "XXM", "XXLC"]
   - Expected output: [3, 67, 0, 0]
   - This test checks if the program correctly handles valid combinations where the values are added together. Two cases for which the additive combinations would work, and two cases it wouldn't.

3. **Subtractive Combinations Test**: Test the conversion of subtractive combinations of Roman numerals.
   - Input: ["IV", "XCIX", "DM", "CDM"]
   - Expected output: [4, 99, 0, 0]
   - This test checks if the program correctly handles valid combinations where a smaller value is subtracted from a larger value. Two cases for which the additive combinations would work, and two cases it wouldn't.

4. **Maximum Roman Numeral Test**: Test the conversion of the maximum possible Roman numeral and an invalid Roman numeral that exceeds the maximum value.
   - Input: ["MMMCMXCIX", "MMMM"]
   - Expected output: [3999, 0]
   - This test checks if the program correctly handles the maximum Roman numeral value (3999) and correctly identifies an invalid Roman numeral that exceeds the maximum value. One case for which the additive combinations would work, and one case it wouldn't.

5. **Invalid Combinations Test**: Test the conversion of invalid Roman numeral combinations that contain non-Roman numeral characters.
   - Input: ["ABC", "X!V", "123"]
   - Expected output: [0, 0, 0]
   - This test checks if the program can correctly identify and handle invalid combinations that contain non-Roman numeral characters. ALL cases should fail.

6. **Invalid Repetition Test**: Test the conversion of invalid Roman numeral combinations that contain repeated characters exceeding the allowed repetition.
   - Input: ["IIII", "CCCCC"]
   - Expected output: [0, 0]
   - This test checks if the program can correctly identify and handle invalid combinations that exceed the allowed repetition of certain characters. ALL cases should fail.

7. **Invalid Subtractive Combinations Test**: Test the conversion of invalid subtractive combinations of Roman numerals.
   - Input: ["VV", "DD", "LL"]
   - Expected output: [0, 0, 0]
   - This test checks if the program can correctly identify and handle invalid subtractive combinations where a smaller value appears before a larger value. ALL cases should fail.

8. **Mixed Case Test**: Test the conversion of mixed-case Roman numerals (both uppercase and lowercase).
   - Input: ["mcxi", "IiX"]
   - Expected output: [1111, 0]
   - This test checks if the program correctly handles mixed-case input and converts it to the correct integer value. One case for which the additive combinations would work, and one case it wouldn't.

9. **Mixed Complex Combinations Test**: Test the conversion of complex combinations of Roman numerals with mixed cases.
   - Input: ["McmXIIv", "mmMCMxcIx"]
   - Expected output: [1944, 3999]
   - This test checks if the program correctly handles complex combinations of Roman numerals with mixed cases and converts them to the correct integer values. Both cases should pass.

10. **Alpha-Numerical Test Cases**: Test the conversion of input strings that contain alphanumeric characters.
   - Input: ["0XX", "CD2", "70G", "h^8"]
   - Expected output: [0, 0, 0, 0]
   - This test checks if the program can correctly identify and handle input strings that contain non-Roman numeral alphanumeric characters. ALL cases should fail.

11. **Empty and Null Inputs Test**: Test the conversion of empty strings and null inputs.
   - Input: ["", " ", null]
   - Expected output: [0, 0, 0]
   - This test checks if the program can handle empty strings and null inputs gracefully and returns the expected result. ALL cases should fail.

12. **Spacers Test**: Test the conversion of input strings that contain spaces as spacers between characters.
   - Input: ["F HCF", " I", "I II"]
   - Expected output: [0, 1, 0]
   - This test checks if the program can handle input strings that contain spaces as spacers between characters and correctly converts them to the expected integer values.

Each test case areas have been implemented to check specific domains of test cases. The initial code for the Roman-Numerals Test have been updated with each test cases and commented out to reflect the changes made to the code. This is an initial snippet of the code:

CHANGES FOR EACH TEST CASES COMMENTED OUT (1/2):

RomanNumeral.java > RomanNumeral > convert(String)

```java
// package assignment1;

import java.util.*;

public class RomanNumeral {
    private static Map<Character, Integer> map;
    static {
        map = new HashMap<>();
        map.put(key:'I', value:1);
        map.put(key:'V', value:5);
        map.put(key:'X', value:10);
        map.put(key:'L', value:50);
        map.put(key:'C', value:100);
        map.put(key:'D', value:500);
        map.put(key:'M', value:1000);
    }

    public static int convert(String s) {
        // Mixed Case Test
        s=s.toUpperCase();
        if (s.length() == 0 || s.trim().isEmpty()) { return 0; }
        int convertedNumber = 0;
        int prev = 3999;
        // count repeatitive cases of I's, X's, C's before a big number
        int repeat = 0;
        for (int i = 0; i < s.length(); i++) {
            // One Letter Test
            if (!map.containsKey(s.charAt(i))) {
                return 0;
            }

            // Invalid Combinations
            else if (i+1 < s.length() && (!map.containsKey(s.charAt(i+1)))) {
                return 0;
            }

            // Finding the character next to the string (if there is one)
            int next = (i + 1 < s.length()) ? map.get(s.charAt(i + 1)) : 0;
            int currentNumber = map.get(s.charAt(i));

            // Invalid Subtractive Combinations Test
            if ((prev == currentNumber) && (prev == 5 || prev == 50 || prev == 500)) {
                return 0;
            }
```

CHANGES FOR EACH TEST CASES COMMENTED OUT (2/2):

```
46              // if current number == I/X/C == previous number <-- state: true
47              boolean RepeatState = ((prev == currentNumber) && (prev == 1 || prev == 10
48              || prev == 100 || prev == 1000));
49
50              // InValid Repetitive Test (If more than 3 I's/X's/C's <-- return 0)
51              // Maximum Roman Numeral Test
52              if ((repeat = RepeatState ? ++repeat : 0) >= 3) {
53                  return 0;
54              }
55
56              // additive combination test addition
57              // substractive combinations test
58              if (currentNumber < next && (RepeatState)) {
59                  return 0;
60              } else if (currentNumber >= next) {
61                  convertedNumber += currentNumber;
62              } else if (currentNumber == 1 || currentNumber == 10 || currentNumber == 100) {
63                  convertedNumber -= currentNumber;
64              } else {
65                  return 0;
66              }
67              prev = currentNumber;
68          }
69          return convertedNumber;
70      }
71  }
72
```

I generated a **converter** function for each of the test case areas for this test.

```
private static Stream<Arguments> generator() {

    return Stream.of(

        Arguments.of(
                    "Test Case 1: Letter Test",
                    new String[] { "I", "X", "Z", "!" },
                    new int[] { 1, 10, 0, 0 }),
        Arguments.of(
                    "Test Case 2: Additive Combinations Test",
                    new String[] { "III", "LXVII", "XXM", "XXLC" },
                    new int[] { 3, 67, 0, 0 }),
        Arguments.of(
                    "Test Case 3: Subtractive Combinations Test",
                    new String[] { "IV", "XCIX", "DM", "CDM" },
                    new int[] { 4, 99, 0, 0 }),
        Arguments.of(
                    "Test Case 4: Maximum Roman Numeral Test",
                    new String[] { "MMMCMXCIX", "MMMM" },
                    new int[] { 3999, 0 }),
        Arguments.of(
                    "Test Case 5: Invalid Combinations Test",
                    new String[] { "ABC", "X!V", "123" },
                    new int[] { 0, 0, 0 }),
        Arguments.of(
                    "Test Case 6: Invalid Repetition Test",
                    new String[] { "IIII", "CCCCC" },
                    new int[] { 0, 0 }),
        Arguments.of(
                    "Test Case 7: Invalid Subtractive Combinations",
                    new String[] { "VV", "DD", "LL" },
                    new int[] { 0, 0, 0 }),
        Arguments.of(
                    "Test Case 8: Mixed Case Test",
                    new String[] { "mcxi", "IiX" },
                    new int[] { 1111, 0 }),
        Arguments.of("Test Case 9: Mixed Complex Combinations",
                    new String[] { "McmXlIv", "mmMCMxcIx" },
                    new int[] { 1944, 3999 }),
        Arguments.of(
                    "Test Case 10: AlphaNumerical Cases",
                    new String[] { "0XX", "CD2", "70G", "h^8" },
                    new int[] { 0, 0, 0, 0 }),
        Arguments.of(
                    "Test Case 11: Empty and Null Inputs",
                    new String[] { "", " ", null },
                    new int[] { 0, 0, 0 }),
        Arguments.of(
                    "Test Case 12: Spacers",
                    new String[] { "F HCF", " I", "I II" },
                    new int[] { 0, 1, 0 }));
}
```

Question 2

Test 1: Inside the board (2, 3) The chosen position represents an arbitrary point inside the board.

Test 2: Inside the board (0, 5) When player is positioned at the top-right corner of the board, specifically at coordinates (0, 5).

Test 3: Inside the board (4, 0) When player is positioned at the bottom-left corner of the board.

Test 4: Inside the board (1, 1) When player is positioned in the top-left quadrant.

Test 5: Inside the board (5, 4) When player is positioned in the bottom-right corner of the board.

Test 6: Outside the board (-1, 3) By placing the player one unit to the left of the leftmost column, at coordinates (-1, 3), this test case assesses whether the GameBoard identifies positions outside the board to the left.

Test 7: Outside the board (2, 6) Position outside the board's top boundary, specifically.

Test 8: Outside the board (6, 4) One unit to the right of the rightmost column.

Test 9: Outside the board (7, -2) Positioned below the board's bottom boundary.

Test 10: On the top boundary (5, 5)
Top boundary at coordinates (5, 5).

Test 11: Off the top boundary (5, 6): Beyond the top boundary.

Test 12: On the left boundary (0, 3): Leftmost column at coordinates (0, 3).

Test 13: Off the left boundary (-1, 3)
Outside the leftmost boundary.

Test 14: On the right boundary (3, 5) Rightmost Column.

Test 15: Off the right boundary (6, 5) Beyond the right boundary.

Test 16: On the bottom boundary (3, 0) bottom-most boundary.

Test 17: Off the bottom boundary (3, -1) Below the bottom boundary.

I generated a **converter** function for each of the test case areas for this test.

```
return Stream.of(
        // Good Weather Test Cases
        Arguments.of("Inside the board", 2, 3, true),
        Arguments.of("Inside the board", 0, 5, true),
        Arguments.of("Inside the board", 4, 0, true),
        Arguments.of("Inside the board", 1, 1, true),
        Arguments.of("Inside the board", 5, 4, true),

        // Bad Weather Test Cases
        Arguments.of("Outside the board", -1, +3, false),
        Arguments.of("Outside the board", +2, +6, false),
        Arguments.of("Outside the board", +6, +4, false),
        Arguments.of("Outside the board", +7, -2, false),

        // Test cases for the top boundary
        Arguments.of("On the top boundary", 5, 5, true),
        Arguments.of("Off the top boundary", 5, 6, false),

        // Test cases for the left boundary
        Arguments.of("On the left boundary", 0, 3, true),
        Arguments.of("Off the left boundary", -1, 3, false),

        // Test cases for the right boundary
        Arguments.of("On the right boundary", 3, 5, true),
        Arguments.of("Off the right boundary", 6, 5, false),

        // Test cases for the bottom boundary
        Arguments.of("On the bottom boundary", 3, 0, true),
        Arguments.of("Off the bottom boundary", 3, -1, false)
);
```

Question 3

Boundary between Equilateral and Scalene Triangle:

Equilateral Triangle: (5, 5, 5) ---> *ON POINT*
Scalene Triangle: (5, 6, 7) ---> *OFF POINT*

Explanation: By varying two side of the equilateral triangle slightly, such as increasing it from 5 to 6, you move towards the condition of a scalene triangle where all sides have different lengths.

Boundary between Equilateral and Isosceles Triangle:

Equilateral Triangle: (5, 5, 5) ---> *ON POINT*
Isosceles Triangle: (5, 6, 5) ---> *OFF POINT*

Explanation: By changing one side of the equilateral triangle slightly, such as increasing it from 5 to 6, you transition towards the condition of an isosceles triangle where two sides have the same length.

Boundary between Equilateral and Invalid Triangle:

Equilateral Triangle: (5, 5, 5) ---> *ON POINT*
Invalid Triangle: (5, 5, 11) ---> *OFF POINT*

Explanation: By increasing one side significantly, such as increasing it from 5 to 11, you create a triangle where the sum of the lengths of any two sides is less than or equal to the length of the third side, violating the triangle inequality rule and making it an invalid triangle.

I generated a **converter** function for each of the test case areas for this test.

```
private static Stream<Arguments> triangleTestCases() {
    return Stream.of(
            // Equilateral & Scalene Triangle Tests
            Arguments.of("Equilateral Triangle", 5, 5, 5, Triangle.TriangleType.EQUILATERAL),
            Arguments.of("SCALENE Triangle", 5, 6, 7, Triangle.TriangleType.SCALENE),

            // Equilateral & Isosceles Triangle  Tests
            Arguments.of("Equilateral Triangle", 5, 5, 5, Triangle.TriangleType.EQUILATERAL),
            Arguments.of("ISOSCELES Triangle", 5, 6, 5, Triangle.TriangleType.ISOSCELES),

            // Equilateral & Invalid Triangle Tests
            Arguments.of("Equilateral Triangle", 5, 5, 5, Triangle.TriangleType.EQUILATERAL),
            Arguments.of("Not Invalid", 5, 5, 11, Triangle.TriangleType.INVALID)
    );
```

QUESTION 1 MAIN FILE

```
package assignment1;
import java.util.*;

public class RomanNumeral {
    private static Map<Character, Integer> map;
    static {
        map = new HashMap<>();
        map.put('I', 1);
        map.put('V', 5);
        map.put('X', 10);
        map.put('L', 50);
        map.put('C', 100);
        map.put('D', 500);
        map.put('M', 1000);
    }
    public static int convert(String s) {
        if (s==null || s.length() == 0 || s.trim().isEmpty()) { return 0; }
        s=s.trim().toUpperCase(); // Mixed Case Test
        int convertedNumber = 0;
        int prev = 3999;
        int repeat = 0; // count repetitive cases of I's, X's, C's before a
big number
        for (int i = 0; i < s.length(); i++) {
            if (!map.containsKey(s.charAt(i))) { return 0; } // One Letter
Test
            else if (i+1 < s.length() && (!map.containsKey(s.charAt(i+1)))) {
return 0; } // Invalid Combinations
            int next = (i + 1 < s.length()) ? map.get(s.charAt(i + 1)) : 0;
            int currentNumber = map.get(s.charAt(i));
            if ((prev == currentNumber) && (prev == 5 || prev == 50 || prev
== 500)) { return 0; } // Invalid Subtractive Combinations Test
            boolean RepeatState = ((prev == currentNumber) && (prev == 1 ||
prev == 10 || prev == 100 || prev == 1000)); // if current number == I/X/C ==
previous number <-- state: true
            if ((repeat = RepeatState ? ++repeat : 0) >= 3) { return 0; } //
InValid Repetitive Test (If more than 3 I's/X's/C's <-- return 0) //  Maximum
Roman Numeral Test
            if (currentNumber < next && (RepeatState)) { return 0; } //
additive combination test addition
            else if (currentNumber >= next) { convertedNumber +=
currentNumber; }
            else if (currentNumber == 1 || currentNumber == 10 ||
currentNumber == 100) { convertedNumber -= currentNumber; } // substractive
combinations test
            else { return 0; } // subtractive combinations test
            prev = currentNumber; // additive combination test
        }
        return convertedNumber;
    }
}
```

QUESTION 1 TEST FILE

```
package assignment1;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
```

```
import java.util.stream.Stream;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class RomanNumeralTest {
    @ParameterizedTest
    @MethodSource("generator")
    void convert(String description, Object inputValues, Object
expectedValue) {
        String[] inputs;
        int[] expectedResults;

        if (inputValues instanceof String[]) { inputs =
(String[]) inputValues; }
        else { inputs = new String[]{(String) inputValues}; }
        if (expectedValue instanceof int[]) { expectedResults =
(int[]) expectedValue; }
        else { expectedResults = new int[]{(int) expectedValue};
}

        for (int i = 0; i < Math.min(inputs.length,
expectedResults.length); i++) {
            int expectedResult = expectedResults[i];
            int result = RomanNumeral.convert(inputs[i]);
            assertEquals(expectedResult, result);
        }
    }

    private static Stream<Arguments> generator() {
        return Stream.of(
                Arguments.of("Test Case 1: Letter Test", new
String[] {"I","X","Z","!"}, new int[] {1,10,0,0}),
                Arguments.of("Test Case 2: Additive Combinations
Test", new String[] {"III","LXVII","XXM","XXLC"}, new int[]
{3,67,0,0}),
                Arguments.of("Test Case 3: Subtractive
Combinations Test", new String[] {"IV","XCIX","DM","CDM"}, new
int[] {4,99,0,0}),
                Arguments.of("Test Case 4: Maximum Roman Numeral
Test", new String[] {"MMMCMXCIX","MMMM"}, new int[] {3999,0}),
                Arguments.of("Test Case 5: Invalid Combinations
Test", new String[] {"ABC","X!V","123"}, new int[] {0,0,0}),
                Arguments.of("Test Case 6: Invalid Repetition
Test", new String[] {"IIII","CCCCC"}, new int[] {0,0}),
                Arguments.of("Test Case 7: Invalid Subtractive
Combinations", new String[] {"VV","DD","LL"}, new int[]
{0,0,0}),
                Arguments.of("Test Case 8: Mixed Case Test", new
String[] {"mcxi","IiX"}, new int[] {1111,0}),
                Arguments.of("Test Case 9: Mixed Complex
Combinations", new String[] {"McmXlIv","mmMCMxcIx"}, new int[]
```

```
{1944,3999}),
                Arguments.of("Test Case 10: AlphaNumerical
Cases", new String[] {"0XX","CD2", "70G", "h^8"}, new int[]
{0,0,0,0}),
                Arguments.of("Test Case 11: Empty and Null
Inputs", new String[] {"","  ", null}, new int[] {0,0,0}),
                Arguments.of("Test Case 12: Spacers", new
String[] {"F HCF","  I", "I II"}, new int[] {0,1,0})
        );
    }
}
```

QUESTION 2 MAIN FILE

```
package assignment1;
import java.util.*;

public class GameBoard {
    private int[][] board;
    public GameBoard() { board = new int[6][6]; } // Creating a 6x6 square
board
    public int[][] getBoard() { return board; }
}
```

QUESTION 2 TEST FILE

```
package assignment1;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

public class GameBoardTest {
    private GameBoard gameBoard;

    @BeforeEach
    public void setUp() { gameBoard = new GameBoard(); }  // Creating a
GameBoard instance before each test

    private boolean isPlayerInsideBoard(int x, int y) { return x >= 0 && x <
6 && y >= 0 && y < 6; }

    @ParameterizedTest
    @MethodSource("generator")
    void testPlayerPosition(String description, int x, int y, boolean
expected) {
        int[][] board = gameBoard.getBoard();
        boolean result = isPlayerInsideBoard(x, y);
        Assertions.assertEquals(expected, result, description);
    }

    private static Stream<Arguments> generator() {
        return Stream.of(
                // Good Weather Test Cases
                Arguments.of("Inside the board", 2, 3, true),
                Arguments.of("Inside the board", 0, 5, true),
                Arguments.of("Inside the board", 4, 0, true),
                Arguments.of("Inside the board", 1, 1, true),
                Arguments.of("Inside the board", 5, 4, true),

                // Bad Weather Test Cases
                Arguments.of("Outside the board", -1, +3, false),
                Arguments.of("Outside the board", +2, +6, false),
                Arguments.of("Outside the board", +6, +4, false),
                Arguments.of("Outside the board", +7, -2, false),

                // Test cases for the top boundary
                Arguments.of("On the top boundary", 5, 5, true),
                Arguments.of("Off the top boundary", 5, 6, false),
```

```
                // Test cases for the left boundary
                Arguments.of("On the left boundary", 0, 3, true),
                Arguments.of("Off the left boundary", -1, 3, false),

                // Test cases for the right boundary
                Arguments.of("On the right boundary", 3, 5, true),
                Arguments.of("Off the right boundary", 6, 5, false),

                // Test cases for the bottom boundary
                Arguments.of("On the bottom boundary", 3, 0, true),
                Arguments.of("Off the bottom boundary", 3, -1, false)
        );
    }
}
```

QUESTION 3 MAIN FILE

```
package assignment1;

public class Triangle {
    public enum TriangleType {
        EQUILATERAL,
        ISOSCELES,
        SCALENE,
        INVALID
    }

    public static TriangleType categorize(int x, int y, int z) {
        if (x > 0 && y > 0 && z > 0 && x + y >= z && x + z >= y && y + z >=
x) {
            if (x == y && y == z) {
                return TriangleType.EQUILATERAL;
            } else if (x == y || y == z || x == z) {
                return TriangleType.ISOSCELES;
            } else {
                return TriangleType.SCALENE;
            }
        } else {
            return TriangleType.INVALID;
        }
    }
}
```

QUESTION 3 TEST FILE

```
package assignment1;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

public class TriangleTest {

    @ParameterizedTest
    @MethodSource("triangleTestCases")
    void testTriangleCategorization(String description, int x, int y, int z,
Triangle.TriangleType expected) {
        Triangle.TriangleType result = Triangle.categorize(x, y, z);
        Assertions.assertEquals(expected, result, description);
    }

    private static Stream<Arguments> triangleTestCases() {
        return Stream.of(
                // Equilateral & Scalene Triangle Tests
                Arguments.of("Equilateral Triangle", 5, 5, 5,
Triangle.TriangleType.EQUILATERAL),
                Arguments.of("SCALENE Triangle", 5, 6, 7,
Triangle.TriangleType.SCALENE),

                // Equilateral & Isosceles Triangle  Tests
                Arguments.of("Equilateral Triangle", 5, 5, 5,
Triangle.TriangleType.EQUILATERAL),
```

```
                Arguments.of("ISOSCELES Triangle", 5, 6, 5,
Triangle.TriangleType.ISOSCELES),

                // Equilateral & Invalid Triangle Tests
                Arguments.of("Equilateral Triangle", 5, 5, 5,
Triangle.TriangleType.EQUILATERAL),
                Arguments.of("Not Invalid", 5, 5, 11,
Triangle.TriangleType.INVALID)
        );
    }
}
```