Programming languages, Java (BSc) -- Lab 7

Explanation

Compile and run `SimpleTest.java` in the following way:

```
javac -cp junit-4.12.jar;hamcrest-core-1.3.jar;. SimpleTest.java
java -cp junit-4.12.jar;hamcrest-core-1.3.jar;. org.junit.runner.JUnitCore
SimpleTest
```

https://www.vogella.com/tutorials/Hamcrest/article.html
https://www.codejava.net/testing/how-to-compile-and-run-junit-tests-in-command-line
http://tutorials.jenkov.com/java-unit-testing/simple-test.html
http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Task 1

Create a class `MathUtils` containing a static method `Increment()`, which increases an integer number by one in a safe way: it only increases its parameter if it is not the largest representable `int` value. The result of the computation should be returned by this method.

https://www.geeksforgeeks.org/integer-max_value-and-integer-min_value-in-java-with-examples/

Test the `Increment()` method with white-box testing.

https://www.guru99.com/software-testing-introduction-importance.html

```
public class MathUtils {
    public static int Increment(int x) {
        if (x == Integer.MAX_VALUE) {
            return x;
        }
        else {
            return x + 1;
        }
    }
}
```

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;


public class MathUtilsTest {
    @Test
    public void testPositive() {
        int res = MathUtils.Increment(4);
        assertEquals(5, res);
    }

    @Test
    public void testNegative() {
        int res = MathUtils.Increment(-3);
        assertEquals(-2, res);
    }

    @Test
    public void testZero() {
        int res = MathUtils.Increment(0);
        assertEquals(1, res);
    }

    @Test
    public void testMaxValue1() {
        int res = MathUtils.Increment(Integer.MAX_VALUE);
        assertEquals(Integer.MAX_VALUE, res);
    }

    @Test
    public void testMaxValue2() {
        int res = MathUtils.Increment(Integer.MAX_VALUE - 1);
        assertEquals(Integer.MAX_VALUE, res);
    }
}
```

Task 2

The `Book` class represents an item (a book) at an auction.

Test the `Book` class with white-box testing. Try to maximize the following metrics.

- coverage of methods
- coverage of branches (both branches of each decision point)
- coverage of conditions (subconditions in boolean expressions)
- coverage of loops (execution of loop bodies 0, 1, 2 times)

The class contains a nested enumeration type (`Book.Genre`), which contains the following values: `FANTASY`, `SATIRE`, `SCIFI`, `PHILOSOPHY`, `EDUCATIONAL`.

The class has 5 private fields.

- author (`String`)
- title (`String`)
- reserve price (`int`)
- identifier (`int`)
- genre (`Book.Genre`).

The class has a private constructor: it takes as parameter the author, the title, the genre and the reserve price, and sets the corresponding fields. The identifier should be initialized to a value which is larger by one than the previously assigned identifier. (The first identifier should be `0`.)

The class contains a static `make()` method. This method takes almost the same parameters as the constructor. The only difference is that the genre is represented as a `String` rather than a `Book.Genre`. The `make()` method checks that the parameters are valid and meaningful. In the positive case, it constructs and returns a new `Book` object, otherwise it returns a `null` reference.

- The name of the author and the title are accepted, if they are not `null`, they contain at least two characters, and they only contain letters, digits and spaces.
- The reserve price should be a positive value.
- The genre should be convertible into a `Book.Genre` value.

Prepare a static method `isSameGenre()`, which takes two `Book` objects as parameter and returns whether they have the same genre.

Create a `compare()` instance method, which takes another `Book` object as parameter. This method can only compare books from the same genre - if called with books of different genres, the method should throw an `IllegalArgumentException`. When two books of the same genre are compared, they are compared based on the reserve price: the larger the reserve price, the "larger" the book is. If the current book is larger than the one passed as parameter, the `compare()` method should return 1; if smaller, it should return -1; and if equal, it should return 0.

https://developer.apple.com/documentation/xctest/xctestcase/understanding_setup_and_teardown_for_test_methods

https://pjcj.net/testing_and_code_coverage/paper.html

```
public class Book {
    static enum Genre {
        FANTASY, SATIRE, SCIFI, PHILOSOPHY, EDUCATIONAL;
    }

    private final String author;
    private final String title;
    private final int reservePrice;
    private final int id;
    private final Genre genre;

    private static int lastId;
    public static void resetId() { lastId = 0; }
    public int getReservePrice() { return reservePrice; }
    public int getId() { return id; }

    private Book(String author, String title, Genre genre, int reservePrice)
{
        this.author = author;
        this.title = title;
        this.reservePrice = reservePrice;
        this.genre = genre;
        id = lastId++;
    }

    public static Book make(String artist, String title, String genreName,
String reservePriceStr) {
        try {
            int reservePrice = Integer.parseInt(reservePriceStr);

            if (artist == null || title == null || title.length() < 2 ||
reservePrice <= 0)
                return null;

            Genre g = Genre.valueOf(genreName);

            int i = 0;
            while (i < title.length()) {
      if (Character.isLetter(title.charAt(i)) ||
Character.isDigit(title.charAt(i)) ||
Character.isWhitespace(title.charAt(i))) {
                    ++i;
                } else {
                    return null;
                }
            }
            return new Book(artist, title, g, reservePrice);
        } catch (NumberFormatException e) {
            return null;
        } catch (IllegalArgumentException e) {
            return null;
        }
    }

    public static boolean isSameGenre(Book b1, Book b2) {
        return b1.genre == b2.genre;
    }

    public int compare(Book that) {
        if (!isSameGenre(this, that)) {
            throw new IllegalArgumentException();
        }
        return Integer.compare(reservePrice, that.reservePrice);
    }
}
import org.junit.Test;
import org.junit.Before;
import org.junit.After;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.fail;

public class BookTest{

    @Before
    public void setUp() {
        Book.resetId();
    }
    @After
    public void tearDown() {
        Book.resetId();
    }

    @Test
    public void Book_setsParameters() {
        Book b1 = Book.make("abc", "def","SCIFI","10000");
        Book b2 = Book.make("abc", "def","SCIFI","10000");

        assertEquals(b1.getReservePrice(), 10000);
        assertEquals(b2.getReservePrice(), 10000);

        assertEquals(b1.getId(), 0);
        assertEquals(b2.getId(), 1);
    }

    @Test
    public void make_parsesTypes() {
        // covers int check fail
        assertEquals(Book.make("abc", "def", "SCIFI","tenthousand"), null);

        // covers enum check fail
        assertEquals(Book.make("abc", "def","AVANTGARDE","10000"), null);

        // covers int and enum check success
        assertNotEquals(Book.make("abc", "def", "SCIFI","10000"), null);
    }

    @Test
    public void make_checksValues() {

        // covers title string null check fail
        assertEquals(Book.make("abc", null,"SCIFI","10000"), null);

        // covers title string length check fail
        assertEquals(Book.make("abc", "d","SCIFI","10000"), null);

        // covers reserve price positivity check fail
        assertEquals(Book.make("abc", "def","SCIFI","0"), null);
        assertEquals(Book.make("abc", "def","SCIFI","-1"), null);

        // covers title string and price check success
        assertNotEquals(Book.make("abc", "def","SCIFI","10000"), null);
    }
    @Test
    public void make_checksTitleContents() {
        // covers letter-digit-whitespace check fail
        assertEquals(Book.make("abc", "$$$","SCIFI","10000"), null);

        // covers letter check success
        assertNotEquals(Book.make("abc", "def","SCIFI","10000"), null);

        // covers digit check success
        assertNotEquals(Book.make("abc", "111","SCIFI","10000"), null);

        // covers whitespace check success
        assertNotEquals(Book.make("abc", "     ","SCIFI","10000"), null);
    }

    @Test
    public void compare_isSameGenre(){
        Book b1 = Book.make("abc", "def1","SCIFI","10000");
        Book b2 = Book.make("abc", "def2","EDUCATIONAL","10000");
        Book b3 = Book.make("abc", "def3","EDUCATIONAL","7777");
        assertFalse(Book.isSameGenre(b1, b2));
        assertTrue(Book.isSameGenre(b2, b3));
    }

    @Test(expected = IllegalArgumentException.class)
    public void compare_NotSameGenre(){
        Book b1 = Book.make("abc", "def","SCIFI","10000");
        Book b2 = Book.make("abc", "def","EDUCATIONAL","10000");
        b1.compare(b2);
    }

    @Test
    public void compare_LessOrGreater() {
        Book b1 = Book.make("abc", "def","EDUCATIONAL","10");
        Book b2 = Book.make("abc", "def","EDUCATIONAL","10000");
        assertEquals(-1, b1.compare(b2));
        assertEquals(1, b2.compare(b1));
    }

    @Test
    public void compare_Equal(){
        Book b1 = Book.make("abc", "deffff","EDUCATIONAL","10000");
        Book b2 = Book.make("abc", "def","EDUCATIONAL","10000");
        assertEquals(0, b1.compare(b2));
    }
}
```

Task 3

The `Adder` class has an `add()` static method with two parameters representing numbers, and it returns their sum. For some reason, the type of the parameters and the return type are `String`s. Therefore, numbers must be encoded to strings before calling this method, and the result should be also decoded. If the parameter strings cannot be interpreted as numbers, the method should throw `IllegalArgumentException`.

https://www.softwaretestingclass.com/difference-between-black-box-testing-and-white-box-testing

```
public class Adder {
    public static String add(String a, String b){...}
}
```

Test the `add()` method with black-box testing, taking care of the following.

- the correct implementation of addition;
- the algebraic properties of addition: commutativity, associativity, unit value;
- for floating point numbers, precision up to `0.01`;
- ability to accept numbers in radix 2;
- ability to accept numbers written in English;
- invalid parameters (`null` or non-numbers) result in `IllegalArgumentException`
- ability to ignore spaces at the beginning and at the end of the strings.

```
public class Adder
{
    public static String add(final String s, final String s2) {
        try {
            return Integer.toString(Integer.parseInt(s) +
Integer.parseInt(s2));
        }
        catch (NumberFormatException ex) {
            try {
                return Double.toString(Double.parseDouble(s) +
Double.parseDouble(s2));
            }
            catch (NumberFormatException ex2) {
                throw new IllegalArgumentException("One or both of the
parameters were not numbers");
            }
        }
    }
}
```

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import org.junit.Test;
import java.util.Arrays;

public class AdderTest {
  private static final double EPSILON = 0.01;

  @Test
  public void add_isSane(){
```

```
  int result = Integer.parseInt(Adder.add("145", "972"));
  assertEquals(result,145+972);

  assertNotEquals(
    Adder.add("1", "2"),
    Adder.add("1", "3"));
}

@Test
public void add_hasProperties(){
  // commutative
  assertEquals(
    Adder.add("250", "300"),
    Adder.add("300", "250"));

  // associative
  assertEquals(
    Adder.add("500", Adder.add("250", "300")),
    Adder.add(Adder.add("500", "250"), "300"));

  // identity element
  String r1 = Adder.add("442", "0");
  String r2 = Adder.add("0", "442");
  String r3 = "442";
  assertTrue( r1.equals(r2) && r2.equals(r3));
}

@Test
public void add_acceptsDouble(){
  double result2 = Double.parseDouble(Adder.add("2.526", "4.995"));

  assertTrue(result2 - (2.526+4.995) < EPSILON);

  assertEquals(
    result2,
    2.526 + 4.995,
    EPSILON);
}

@Test
public void add_acceptsBinary(){
  int result = Integer.parseInt(Adder.add("0b01", "0b11"));

  assertEquals(result, 5);
}

@Test
public void add_acceptsStrings(){
  int result = Integer.parseInt(Adder.add("eighty-nine", "four"));

  assertEquals("ninety-three", result);
}


public void add_worksAroundZero(){
  assertEquals(
    Integer.parseInt(Adder.add("0", "0")),
    String.valueOf(0));

  assertEquals(
    Integer.parseInt(Adder.add("0", "1")),
    String.valueOf(1));

  assertEquals(
    Integer.parseInt(Adder.add("-1", "0")),
    String.valueOf(-1));

  assertEquals(
    Integer.parseInt(Adder.add("1", "-1")),
    String.valueOf(0));
}

@Test
public void add_recognizesNullReference(){
  try{
      Adder.add(null, "12");
      fail();
  } catch(IllegalArgumentException e){
  }

  try{
      Adder.add("12", null);
      fail();
  } catch(IllegalArgumentException e){
  }

  try{
      Adder.add(null, null);
      fail();
  } catch(IllegalArgumentException e){
  }
}


@Test
public void add_recognizesInvalidString(){
  try{
      Adder.add("Abc", "Def");
      fail();
  } catch(IllegalArgumentException e){
  }

  try{
      Adder.add(" ", "12");
      fail();
  } catch(IllegalArgumentException e){
  }
}


@Test
public void add_trimsString(){
  assertEquals(
      Adder.add(" 21  ", " 23"),
        String.valueOf(21+23));
  }
}
```

https://www.javatpoint.com/method-in-java