jvo, hdaniel, jbarateiro

Lab 3: Unit tests

Este lab deve ser feito individualmente.

Submeta o seu código ao mooshak http://deei-mooshak.ualg.pt/~jvo/ usando o seu login da ualg (sem @ualg.pt). Ex: a123456

Este lab é composto por um tutorial e um problema.

Uma submissão ao problema permanecerá *pending* até que seja validada pelo professor durante a aula prática. Só as submissões *final* serão consideradas para avaliação.

Todas as submissões deverão ser feita até

8 de março 2024

A validação poderá ser feita posteriormente, se necessário, até 22 de março de 2024

Tutorial on Unit tests and JUnit (with T2 times)

Runtime logic errors, such as contract violations, are among the more frequent in a poorly debugged program.

Logic errors should be fixed. To fix an error we need to identify it first. Testing is one of the more powerful ways of identifying bugs. However, we should keep in mind that testing only reveals errors, it does not proof their absence.

In this tutorial we will be exercising unit tests and the JUnit framework. In addition, we will see good examples of Java code.

Unit testing allows us to check the correctness of a simple unit of code – the smallest testable part of a program. Ideally unit tests should be simple enough for a fast running time, independent from each other, and should be independent of their execution order. With a Unit Testing framework, such as JUnit, unit test will run automatically each time the code is compiled.

Tests should be written before the code. This is the basic principle of the test-driven approach to software development. In this approach,

1. First, we write the tests
2. Second we write the simplest code that passes the test
3. and then, we improve the quality of the written code.

For the sake of introduction to JUnit and unit tests in software development we will use the example below on time in planet T2.

Example: T2 time (Problem F in mooshak)

Credits
(Fig. from http://www.sci-news.com/astronomy/water-world-exoplanets-life-06365.html)

The national space agency has just announced the discovery of the exoplanet T2 in a habitable zone. T2 has a higher rotation speed than Earth, which implies that its days have exactly 13h 27m 16s each, if measured in terrestrial time. This creates some difficulties so the agency has hired you to develop a program that does operations on times of T2.

Task
Develop a clock for T2 with the following characteristics. Each hour still has 60 minutes and each minute still has 60 seconds. However, arrived at 13h 27m 15s, in the next instant, the clock returns to 00h 00m and 00s (which is considered midnight). Given an instant of time in the format hours (hh), minutes (mm), and seconds (ss) (<hh>:<mm>:<ss>) it is necessary to compute the number of seconds that have passed since midnight. It should also be able to create a time instant in the format <hh>:<mm>:<ss> from a given number of seconds. Also, the clock should also be able to sum up two T2 time instants and return the result as a time instant of T2.

Input
The input has two or three lines.
The first line has either a time instant in the format <hh>:<mm>:<ss> or a positive integer representing a number of seconds.
The second line has one of the following strings “asSeconds”, “asTime” or “add”
Should the second line be “add”, them a third line exists with exactly the same characteristics of the first one.

Output
The output has a single line that could be either a number of seconds - when the second line of the input is “asSeconds” – or a time instant in the format <hh>:<mm>:<ss> for all the other cases.

Sample Input 1
```
00:01:15
asSeconds
```

Sample Output 1
```
75
```

Sample Input 2
```
75
asTime
```

Sample Output 2
00:01:15

Sample Input 3
13:27:15
add
1

Sample Output 3
00:00:00

Sample Input 4
13:27:15
add
00:00:02

Sample Output 4
00:00:01

0. To implement all the specifications, namely adding times, it makes sense to have a class that stores times and implements the required operations. Let this class be named: **T2time**.

1. From the problem description we know that the day in this planet lasts 13h 27m 16s. T2 daytime starts at 00:00:00 and ends at 13:27:15. Adding one more second will reset the time to 00:00:00, since a new day begins. So we must make sure that the class invariant is never violated, i.e., Time must always be in range: 00:00:00 and ends at 13:27:15. The class cannot store times outside this range.

2. To simplify operations, we can store the time in seconds. The daytime range in seconds is:

   {0, 1, …., 13*3600 + 27 *60 + 15} = {0, 1, … 48435}

3. We can immediately think of two ways of ensuring the class invariant:

   i) We "fix" values outside the valid range to be inside. For instance, if we have a time (in seconds) greater than 48435 we store the reminder of the division by 48436, i.e., we remove the information on the number of days and store only the remaining time.

   ii) We give an error if the time is outside the valid range.

   The first option may come handy in further operations but many times "fixing" stuff under the hood can be dangerous. A client of our class may not be aware of the fixing and thus does not understand the results. It is safer to explicitly tell the clients of our class that it does not support the given values, the second option.

4. From the problem description we can assume that clients of our class may want to create objects with a full hour description in HH:MM:SS or just specifying a number of seconds, so we can implement 2 constructors.

5. Let's start implementing the class constructors and then develop unit tests to verify that the class invariant is not violated.

6. Open eclipse, and create a new java project called **T2time**;

7. Add to the project a new class: **T2time**. This class will have the constructors with arguments but empty body. For now, to perform the first unit tests, we will need also an accessor to read the stored seconds:

```java
public class T2time {
   static final int T2DAYSECONDS = 13*3600+27*60+16;  //48436 secs/day
   private int seconds = 0; //valid range [0, T2DAYSECONDS-1]

   public T2time(int secs) { }
   public T2time(int h, int m, int s) { }
   public int asSeconds() { return 0; }
}
```

8. Prepare the project for test-driven development.

9. In Eclipse, add to the project a new JUnit Test Case (File -> New –> JUnit Test Case). At the top select **New JUnit Jupiter test (JUnit5)** and name it **T2timeTests**, then press Finnish. A dialog may open asking this: JUnit 5 is not on the build path. Do you want to add it? If this appears press OK.

10. In IntelliJ, put the caret (cursor) at the class declaration, right-click an select "Show Context Actions" (alternatively, press Alt + Enter). From the menu, select Create Test. Ensure the testing library is Junit5 and name the class as **T2timeTests**. If you see a message "Junit5 library not found in the module", press Fix (it will download the library from a Maven Repository and add this dependency to your project).

11. Now, within the file **T2timeTests.java** we should have:

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class T2timeTests {

   //Tests will go here
}
```

12. The first test we will develop will test the behaviour of the **T2time** constructors. A constructor can be seen as a special function, with the same name as the class and without return type, that will be called automatically each time a new class object is created. The **constructor(s) should enforce the class invariant**. This test will check valid objects:

```
@Test
public void testConstructor0() {
        assertEquals(0,     new T2time(    0).asSeconds());
        assertEquals(3661,  new T2time( 3661).asSeconds());
        //...
        assertEquals(48435, new T2time(48435).asSeconds());
}
```

In this test we are using **assertEquals** using two integer arguments. The first is the expected value of the test and the second is the actual value being tested. See at the end, the many possible arguments that **assertEquals** may run with.

13. [*Advanced*] The next two tests check invalid arguments. Actually, they will test if the constructor, given an argument outside the valid range, throws the expected exception.
The expected exception is specified using **assertThrows**.

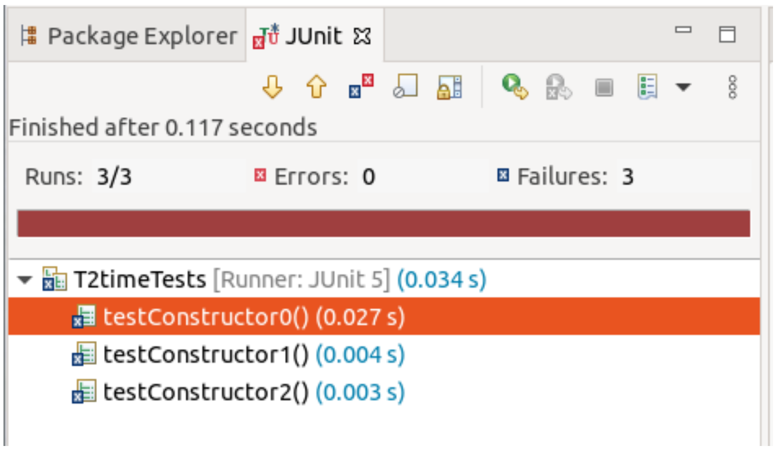
```
@Test
public void testConstructor1() {
        assertThrows(IllegalArgumentException.class, () -> {
                new T2time(-1);
         });
}

@Test
public void testConstructor2() {
        assertThrows(IllegalArgumentException.class, () -> {
                new T2time(48436);
        });
}
```

Note the lambda functions used to create new objects: **() -> { new T2time(-1); }.**

Exceptions and lambdas will be studied in deep later in the course.

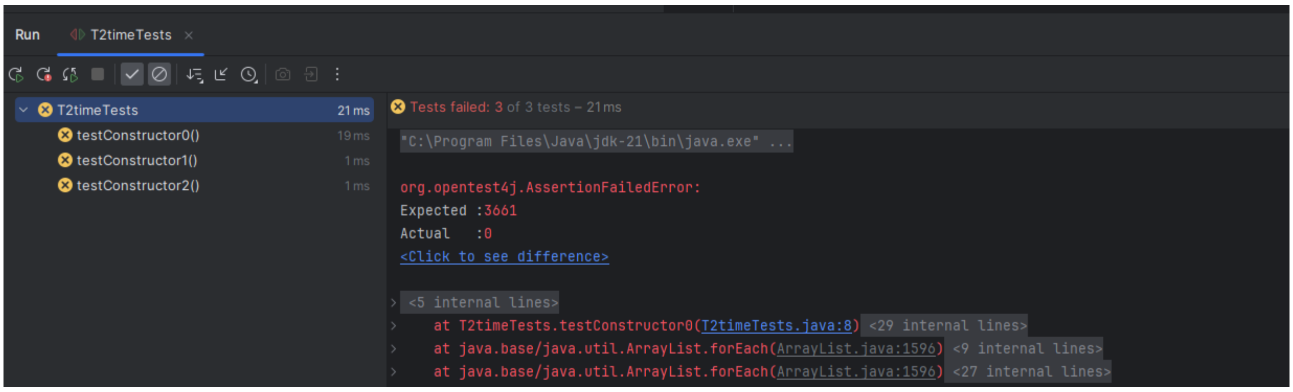
14. Run the tests just created as JUnit Test. We can observe immediately that the test is doing its job by providing us with a red bar indicating that **no tests have passed.**



```
Finished after 0.117 seconds
Runs: 3/3          Errors: 0          Failures: 3

T2timeTests [Runner: JUnit 5] (0.034 s)
    testConstructor0() (0.027 s)
    testConstructor1() (0.004 s)
    testConstructor2() (0.003 s)
```

IntelliJ provides a similar interface, showing that all 3 tests have failed.



```
T2timeTests                21 ms
  testConstructor0()       19 ms
  testConstructor1()        1 ms
  testConstructor2()        1 ms

Tests failed: 3 of 3 tests – 21 ms

"C:\Program Files\Java\jdk-21\bin\java.exe" ...

org.opentest4j.AssertionFailedError: 
Expected :3661
Actual   :0
<Click to see difference>

<5 internal lines>
    at T2timeTests.testConstructor0(T2timeTests.java:8) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) <27 internal lines>
```

15. Now, change the code of the 3 methods in class **T2time** to:

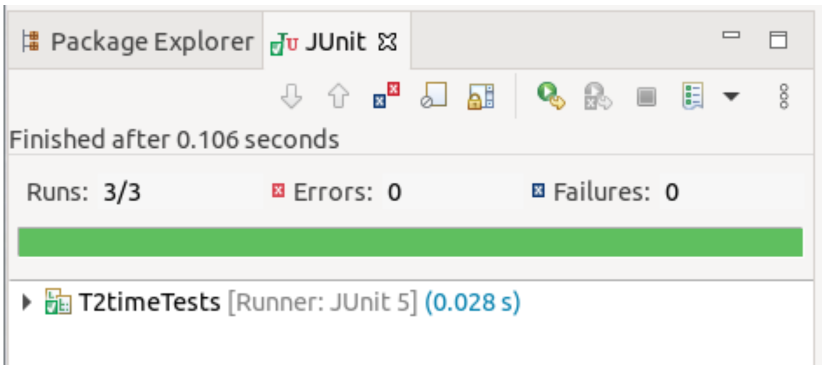
```
    /**
     * @param  secs an integer representing the number of seconds since midnight
     * @inv    this.seconds in 0..T2DAYSECONDS-1
     * @post   this.seconds==secs
     * @throws IllegalArgumentException if secs not in 0..T2DAYSECONDS-1
     */
    public T2time(int secs) {
          if (secs < 0 || secs >= T2DAYSECONDS)
                throw new IllegalArgumentException("0 <= secs < T2DAYSECONDS");
          this.seconds = secs;
    }


    /**
     * @param  h an integer representing the number of hours since midnight
     * @param  m an integer representing the number of min since h
     * @param  s an integer representing the number of hours since m
     * @inv    this.seconds in 0..T2DAYSECONDS-1
     * @post   this.seconds = h*3600 + m*60 + s
     * @throws IllegalArgumentException if (h*3600 + m*60 + s) not in
                                                          [0, T2DAYSECONDS-1]
     */
    public T2time(int h, int m, int s) {
          this(h*3600 + m*60 + s);
    }


    /**
     * @return this.seconds
     */
    public int asSeconds() { return seconds; }
}
```

The second constructor, converts hours, minutes and seconds to just seconds, and then calls the first constructor. The first constructor checks if the number of seconds passed as argument are within the valid range. If not throws an exception. Again, exception handling will be discussed further in the future. For now it is enough to understand that when an exception is thrown, the program ends with an error message.

16. Now run the tests again. We can observe now a green bar indicating that all tests have passed:

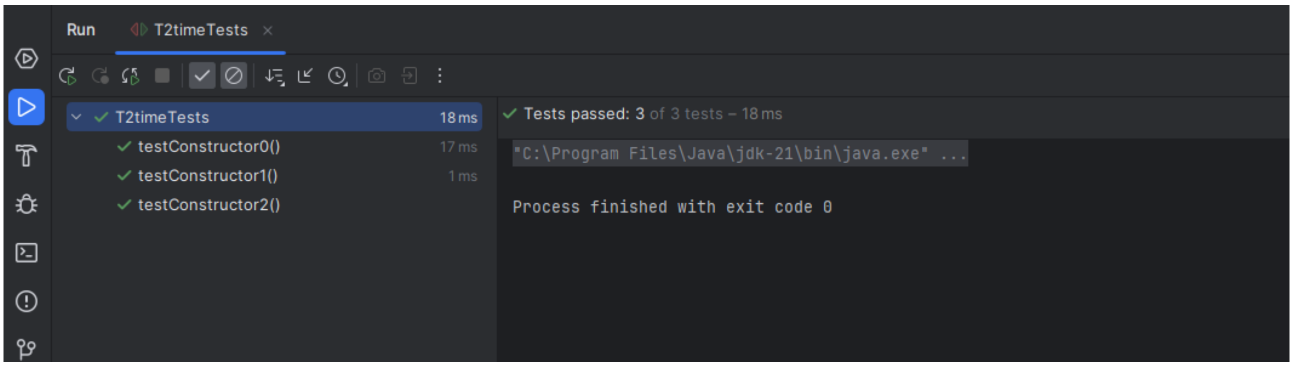


Finished after 0.106 seconds

Runs: 3/3 Errors: 0 Failures: 0

T2timeTests [Runner: JUnit 5] (0.028 s)

IntelliJ shows a similar interface:



```
T2timeTests              18 ms
  testConstructor0()     17 ms
  testConstructor1()      1 ms
  testConstructor2()
```

Tests passed: 3 of 3 tests – 18 ms

```
"C:\Program Files\Java\jdk-21\bin\java.exe" ...

Process finished with exit code 0
```

17. Let´s test the method **String toString()**. This method should return a string with the time expressed as “HH:MM:SS”

```
@Test
public void testtoString0() {
        assertEquals("00:00:00", new T2time(   0).toString());
        assertEquals("00:00:01", new T2time(   1).toString());
        assertEquals("01:01:01", new T2time(3661).toString());
        assertEquals("13:27:15", new T2time(T2time.T2DAYSECONDS-1).toString());
}
```

18. Now write the code of **toString()** method:

```
/**
 * @return this.seconds formatted as "HH:MM:SS"
 */
public String toString() {
        int h = seconds / 3600;
        int m = (seconds % 3600) / 60;
        int s = (seconds % 3600) % 60;

        return String.format("%02d:%02d:%02d",h,m,s);
 }
```

19. Run the tests again.

20. If everything is ok, the JUnit offer you a magnificent green bar, denoting that the code has passed the test again. Otherwise, the bar will be red and the failed tests will be marked failed with an 'x'. Selecting each failed test we can see the expected output and the actual output.

21. Now we will test the **add** operation. If the sum exceeds a day, it is reset.

```
@Test
public void testAdd() {
        assertEquals("00:00:00",
                        (new T2time(0,0,0).add(new T2time(0,0,0))).toString());
        assertEquals("00:00:01",
                        (new T2time(0,0,1).add(new T2time(0,0,0))).toString());
        assertEquals("00:00:00",
                        (new T2time(13,27,15).add(new T2time(0,0,1))).toString());
        assertEquals("00:00:01",
                        (new T2time(13,27,15).add(new T2time(0,0,2))).toString());
        assertEquals("01:01:01",
                        (new T2time(13,27,15).add(new T2time(1,1,2))).toString());
}
```

22. The implementation of **add()** method:

```
/**
 * @pre    arg != null
 * @return new T2time with the sum of this and arg mod T2DAYSECONDS
 * @post   this == old this
 */
public T2time add(T2time arg) {
        // YOUR CODE HERE
}
```

23. Run all tests again. Did you get the green bar? Great!

24. Now write a Client class:

```
import java.util.Scanner;

public class Client {
   static final String TOTIME  = "asTime";
   static final String TOSECS  = "asSeconds";
   static final String ADDTIME = "add";


   static public void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        String aString  = sc.nextLine();
        String operator = sc.nextLine();
        String output="";
        T2time receiver;

        switch (operator) {
            case TOTIME:
                receiver = new T2time(Integer.parseInt(aString));
                output = receiver.toString();
                break;
            case TOSECS:
                receiver = new T2time(aString);
```

```
                        output = "" + receiver.asSeconds();
                        break;
                case ADDTIME:
                        T2time argument;
                        receiver = new T2time(aString);
                        String bString = sc.nextLine();
                        if (T2time.isTime(bString))
                                argument = new T2time(bString);
                        else
                                argument = new T2time(Integer.parseInt(bString));
                                output = receiver.add(argument).toString();
                        break;
                default:
                        //Seems right to have a default statement ...
            }
            System.out.println(output);
        }
}
```

25. Finally, complete and submit your code to Mooshak, problem F, and get an **Accepted.**

[*Advanced*]

26. We could also perform the same tests Mooshak does, with JUnit.

27. We create a new JUnit file **ClientTest.java** and define the input and output of each test in Strings:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

//For accessing stdin/stdout
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

public class ClientTest {

   /* Mooshak performs black box tests
    * Each row of the array represents one test
    * First element in the row is the input (what to put in STDIN)
    * Second (and last) element in the row is the expected output (at STDOUT)
    */
   static private String [][] stdiotests = {
                { "00:01:15\nasSeconds\n"    , "75\n" },
                { "75\nasTime\n"             , "00:01:15\n" },
                { "13:27:15\nadd\n1\n"       , "00:00:00\n" },
                { "13:27:15\nadd\n00:00:02\n", "00:00:01\n" },
   };
```

28. Then we set a function to redirect the **stdin** and **stdout** to these strings:

```
   // Redirect STDIN and STDOUT for Mooshak like black box tests
   static private ByteArrayOutputStream setIOstreams(String input) {
          //set stdin
          System.setIn(new ByteArrayInputStream(input.getBytes()));
```

```
        //set stdout
        ByteArrayOutputStream os = new ByteArrayOutputStream();

        PrintStream ps = new PrintStream(os);
        System.setOut(ps);
        return os;
    }
```

29. Now we loop through each test, redirect input/output to the strings and call **main()** in the **Client** class:

```
    /*
     * Mooshak like black box tests
     */
    @Test
    public void testCase0() {
        for (String[] test : stdiotests) {
            String input    = test[0];
            String expected = test[1];

            ByteArrayOutputStream os = setIOstreams(input);
            Client.main(null);  //call Main()
            assertEquals(expected, os.toString());
        }
      }
    }
```

30. There are many other things that we can do with the JUnit 5 framework. Here is a list of the available assertions in JUnit 5. Possible arguments are given within parenthesis.

assertTrue
(boolean) Reports an error if boolean is false
(String, boolean) Adds error String to output

assertFalse
(bolean) Reports an error if boolean is true
(String, boolean) Adds error String to output

assertNull
(Object) Reports an error if object is not null
(String, Object) Adds error String to output

assertNotNull
(Object) Reports an error if object is null
(String, Object) *A*dds error String to output

assertSame
(Object, Object) Reports error if two objects are not identical
(String, Object, Object) Adds error String to output

assertNotSame
(Object, Object) Reports error if two objects are identical

(Styring, Object, Object) Adds error String to output

assertEquals
(Object, Object) Reports error if two objects are not equal
(String, Object, Object) Adds error String to output
(String, String) Reports delta between two strings if the two strings are not equal
(String, String, String) Adds error String to output
(boolean, boolean) Reports error if the two booleans are not equal
(String, boolean, boolean) Adds error String to output
(byte,byte) Reports error if the two bytes are not equal
(String, byte, byte) Adds error String to output
(char, char)Reports error if two chars are not equal
(String, char, char) Adds error String to output
(short, short) Reports error if two shorts are not equal
(String, short, short) Adds error String to output
(int, int) Reports error if two ints are not equal
(String, int, int) Adds error String to output
(long, long) Reports error if two longs are not equal
(String, long, long) Adds error String to output
(float, float, float) Reports error if the first two floats are not within range specified by third float
(String, float, float, float) Adds error String to output
(double, double, double) Reports error if the first two doubles are not within range specified by third double
(String, double, double, double) Adds error String to output
(object[], object[]); Reports error if either the legnth or element of each array are not equal. *New to JUnit 4*
(String, object[], object[]) Adds error String to output.

31. Besides that, you can use other annotation below @Test for testing important behaviour of your code. Suppose we want to make sure that some code snippet does not take more than a certain amount of time to execute. For instance, you can make sure that solving a given instance of our problem does not take more than 3s. We specify the maximum amount of time for the test in the first line, right next to **@Test** annotation:

```
import org.junit.jupiter.api.Timeout;
import java.util.concurrent.TimeUnit;
//(…)

    @Test
    @Timeout(3)  // 3 secs
    //@Timeout(value = 3000, unit = TimeUnit.MILLISECONDS)  //3000 ms
    public void testCase0() {

            for (String[] test : stdiotests) {
                    String input    = test[0];
                    String expected = test[1];

                    ByteArrayOutputStream os = setIOstreams(input);
                    Client.main(null);  //call Main()
                    assertEquals(expected, os.toString());
            }
    }
```

32. Learn more on JUnit at **http://www.junit.org**

Problem G: Interception of polygons

In GIS (Geographic Information Systems) is often required to check whether two polygons intercept. In Fig. 1 one can see when two polygons do not intercept (left hand side) and when they do intercept (right hand side).

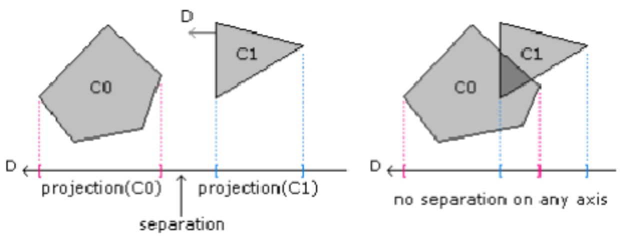


Fig. 1. Intercepting (right) and non-intercepting (left) polygons**,** reproduction from [1].

TASK

Your task is to write the code for checking whether two simple polygons – in the conditions of lab 2 – intercept or not.

For the sake of efficiency, we should first test whether the involving rectangle of a polygon intercepts the involving rectangle of the other. Only and only if these rectangles intercept, polygons should be instantiated and tested for interception.

REQUIREMENTS

Employ the principles and techniques of object-oriented programming. Also, use test-driven development. Due to Mooshak limitations, comment the developed tests.

Input

The first line is a natural, say *k*, indicating the number of vertices of a polygon whose vertices are given by the following *k* lines.

The (*k*+2)-th line is a natural, say *n*, indicating the number of vertices of a polygon whose vertices are given by the following *n* lines.

Output

One line with one of the following:
i) "Point:vi"; message indicating that a point is not from the first quadrant;
ii) "Segment:vi"; indication that two points do not form a line segment;

iii) “Straight:vi”; indication that two points do not form a straight line;
iv) “Poligono:vi”; indication that the given points do not form a polygon;
v) true or false, depending on whether or not the input polygons intercept

Input sample 1

```
4
5 5
8 5
8 7
5 7
4
7 1
9 1
9 3
7 3
```

Output sample 1

```
false
```

Input sample 2

```
4
5 5
8 5
8 7
5 7
4
7 4
9 4
9 6
7 6
```

Output sample 2

```
true
```

References

[1] S. Alawneh and D. Peters, "Ice Simulation Using GPGPU," *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, Liverpool, UK, 2012, pp. 425-431, doi: 10.1109/HPCC.2012.64.