

IIP (E.T.S. de Ingeniería Informática)
Academic Year 2019-2020
*Lab practice 5 – Control flow structures – conditionals:
cross method of Point class*

Profesores de IIP
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València



Contents

1	Objectives and previous work	1
2	Problem description	1
3	Design and implementation of the required classes	2
4	Design of the cross method	3
5	Lab activities	5

1 Objectives and previous work

The main objective of this lab practice is working with the syntax and semantics of the conditional instructions in Java presented in Unit 5 (“Control flow: selection”). More specifically, it is proposed to complete a method in a data-type class for defining a point in the Cartesian plane or 2D space.

2 Problem description

A common problem in graphical applications is to check whether a point is inside a polygon. One of the most used algorithms is the *ray* algorithm, that basically consists in going from the point in one direction, for instance parallel to x -axis in the positive sense, and then to count how many times the *ray* goes through anyone of the segments of the polygon. If the number is even the point is outside the polygon, otherwise the point is inside. Intuitively, if we are inside an enclosure and go in a fixed direction, if we skip the fence an odd number of times, then we are out of the enclosure, but if we skip the fence an even number of times, then we are inside the enclosure. Analogously, if start from an external point skipping the fence an odd number of times means that we are then inside the enclosure, and an even number of times means we are outside the enclosure. Formally, this idea is the *Jordan's theorem* ^{1,2}.

¹<http://erich.realtimerendering.com/ptinpoly/>

²https://wrf.ecse.rpi.edu//Research/Short_Notes/pnpoly.html

In this lab practice you have to implement the first method for checking if a ray starting from a point p goes through a segment delimited by points u and v .

If we have a polygon defined as a sequence of vertex, the first method you have to implement facilitates the checking if a point p is inside a polygon as Figure 1 shows. Basically, we have to count how many sides of the polygon intersect with the ray starting from the point p and check if the result is even or odd. But there is a special case when the ray goes through a segment just in one of their vertex, because all vertex belong to two segments. In this case, it must be taken into account when the number of crossings. The algorithm for counting the number of crossings will be implemented in the lab practice 7.

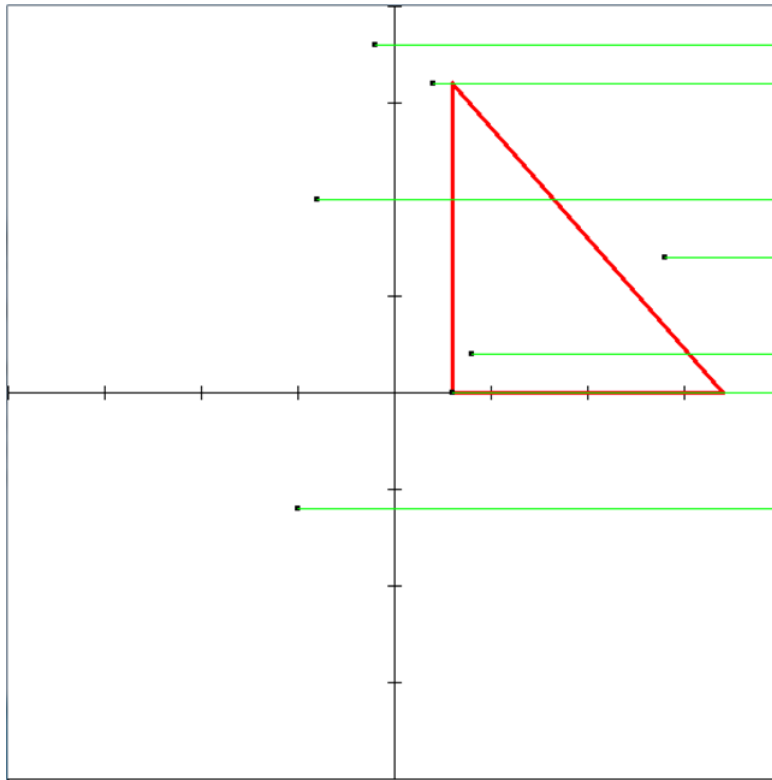


Figure 1: A triangle and several rays crossing its sides.

3 Design and implementation of the required classes

The implementation of the application requires the design and implementation of the following classes:

- The “*data-type*” class **Point** for representing a point the Cartesian plane with the following constants, attributes and methods:
 - **Constants** in Java are class variables declared with the qualifier **final** to make them immutable. The qualifier **static** must be used for declaring them as class variables and no attributes.
Here four integer constants must be declared as **public final static int** with the following names: **DONT_CROSS**, **LOW_CROSS**, **CROSS** and **HIGH_CROSS**, initialised with the values -1, 0, 1 and 2 respectively.
 - **Private attributes** or instance variables **x** and **y** of type **double** for abscissa and ordinate respectively.
 - **Public methods**: constructors, *getters*, *setters*, **toString**, **equals** and **cross**.

The method `cross` has two parameters, `u` and `v`, that are objects of the class `Point` which define the segment \overrightarrow{uv} . This method checks if the ray parallel to x -axis and starting at the current point goes through the segment \overrightarrow{uv} , i.e., goes through a unique point of the segment. In section 4 it is described with detail the case analysis necessary for implementing this method.

- The *program* class `RayTest` whose method `main` uses the method `cross` from the class `Point` and shows the results in the standard output and graphically.

4 Design of the cross method

The profile of the method `cross` is:

```
/** Given a ray emitted from the current point that is parallel
 * to the x-axis in the positive sense, this method checks if the
 * ray goes through the segment represented by two points
 * <code>u</code> and <code>v</code>, in other words, checks
 * if the ray intersects with the segment.
 * @param u <code>Point</code>, one end of the segment.
 * @param v <code>Point</code>, the other end of the segment.
 * @return int, a value indicating one of the possible results:
 * <ul>
 * <li> <code>DONT_CROSS = -1</code>
 *     if the ray does not go through the segment.</li>
 * <li> <code>LOW_CROSS = 0</code>
 *     if the ray goes through the lowest point.</li>
 * <li> <code>CROSS = 1</code>
 *     if the ray goes through the segment in an intermediate point.</li>
 * <li> <code>HIGH_CROSS = 2</code>
 *     if the ray goes through the highest point.</li>
 * </ul>
 */
public int cross(Point u, Point v)
```

This method can be implemented by computing the point where the ray goes through the segment \overrightarrow{uv} taking into account the slope of the line defined by `u` and `v`. Let us use two reference variables to objects of the class `Point`, `pHigh` and `pLow`, for referencing the highest and the lowest points respectively. So `pHigh` will reference the highest of `u` and `v` and `pLow` the lowest one. And let us to use `this` for referencing the point where the ray starts, i.e. the current point. Before describing the possible cases we need to introduce some definitions. Let $(x_{\text{Cut}}, y_{\text{Cut}})$ be the intersection point of the ray with the line defined by `u` and `v`, and let $y = a * x + b$ be the equation of the line defined by `u` and `v`, where

$$\begin{aligned} y_{\text{Cut}} &= \text{this.y} \\ a &= \frac{p_{\text{High}}.y - p_{\text{Low}}.y}{p_{\text{High}}.x - p_{\text{Low}}.x} \\ b &= p_{\text{Low}}.y - a \cdot p_{\text{Low}}.x \\ x_{\text{Cut}} &= \frac{y_{\text{Cut}} - b}{a} \end{aligned}$$

so in the code `xCut` will be computed as:

```
double xCut = (this.y - pLow.y) * (pHigh.x - pLow.x)
              / (pHigh.y - pLow.y) + pLow.x;
```

The possible cases are the following:

CASE 1 Segment \vec{uv} is parallel to x -axis, i.e. $pHigh.y == pLow.y$, then the ray **does NOT go through** the segment, because or it does not intersect at any point or it goes through all the points in the segment \vec{uv} .

CASE 2 The ray **DOES go through** the highest point: $this.y == pHigh.y$.

CASE 3 The ray **DOES go through** the lowest point: $this.y == pLow.y$.

CASE 4 $yCut \in [pLow.y, pHigh.y]$ and $xCut > this.x$
so the ray **DOES go through** the segment \vec{uv} .

CASE 5 $yCut \notin [pLow.y, pHigh.y]$ or $yCut \in [pLow.y, pHigh.y]$ and $xCut < this.x$
so the ray **does NOT go through** the segment \vec{uv} .

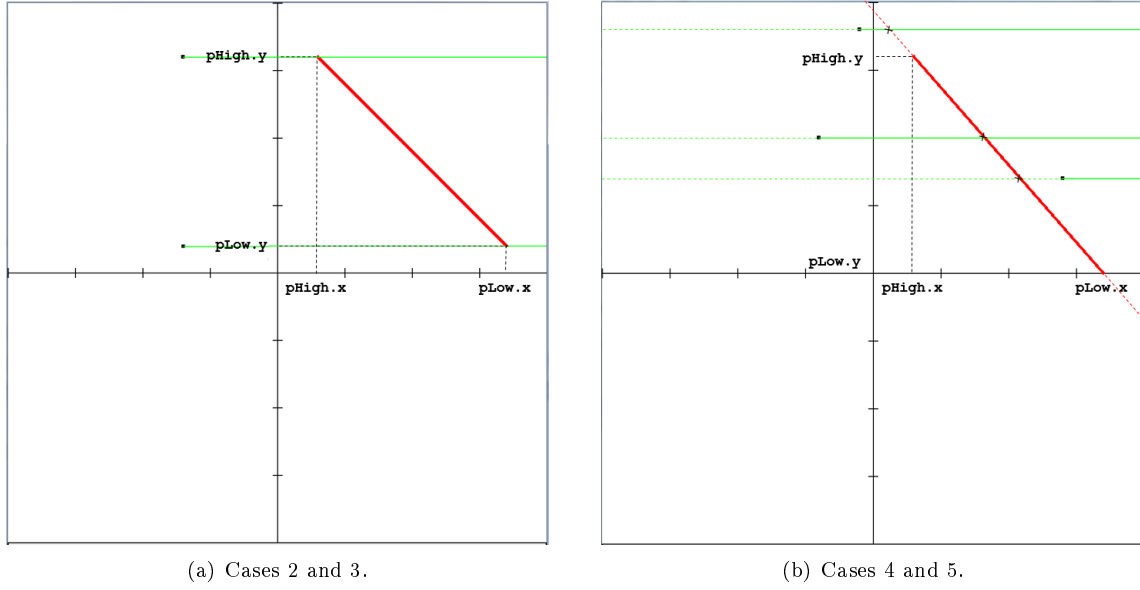


Figure 2: Crossing points of one segment by different rays.

The five cases are summarised in Figure 3.

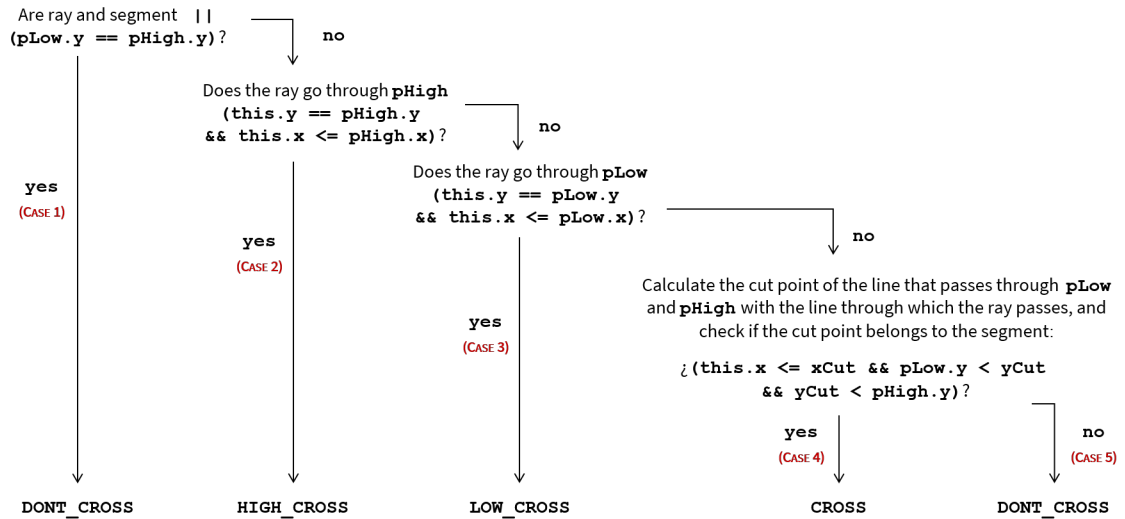


Figure 3: Case analysis for cross method.

5 Lab activities

Activity 1: create the pract5 BlueJ package

1. Download to Downloads directory the `Point.java` and `RayTest.java` files, from PoliformaT (in the folder *Recursos - Laboratorio - Práctica 5*).
2. Open the *BlueJ* iip project for the subject.
3. Create (by using *Edit - New package*) a new package `pract5` and open it (double click).
4. Add to the package `pract5` `Point` and `RayTest` classes by using *Edit - Add Class from File*. Check that first line for both codes is `package pract5;`, which determines that they belong to that package.

Activity 2: installation of Graph2D graphic library

For drawing segments and ray crossings, it is available the graphical library for plotting points, lines and other elements in a 2-dimensional space. It is a library developed in the context of subjects IIP and PRG with the purpose to facilitate students of the first academic year to graphically represent some results in an easy way.

The graphical library is in the class `Graph2D` of the package `graph2D` you can find in the JAR file `graphLib.jar` available in *PoliformaT* in the folder *IIP:recursos/Laboratorio/Librería gráfica*. You have to load this library in *BlueJ* as follows:

1. Put the file `graphLib.jar` in the folder of the project iip, for instance in `${HOME}/DiscoW/iip`
2. Then, in *BlueJ* select the menu *Tools* and then enter in *Preferences*, choose the tab labelled as *Libraries* and add the file `graphLib.jar`.

After that you must restart *BlueJ* for loading the recently added library. Figure 4 shows an example.

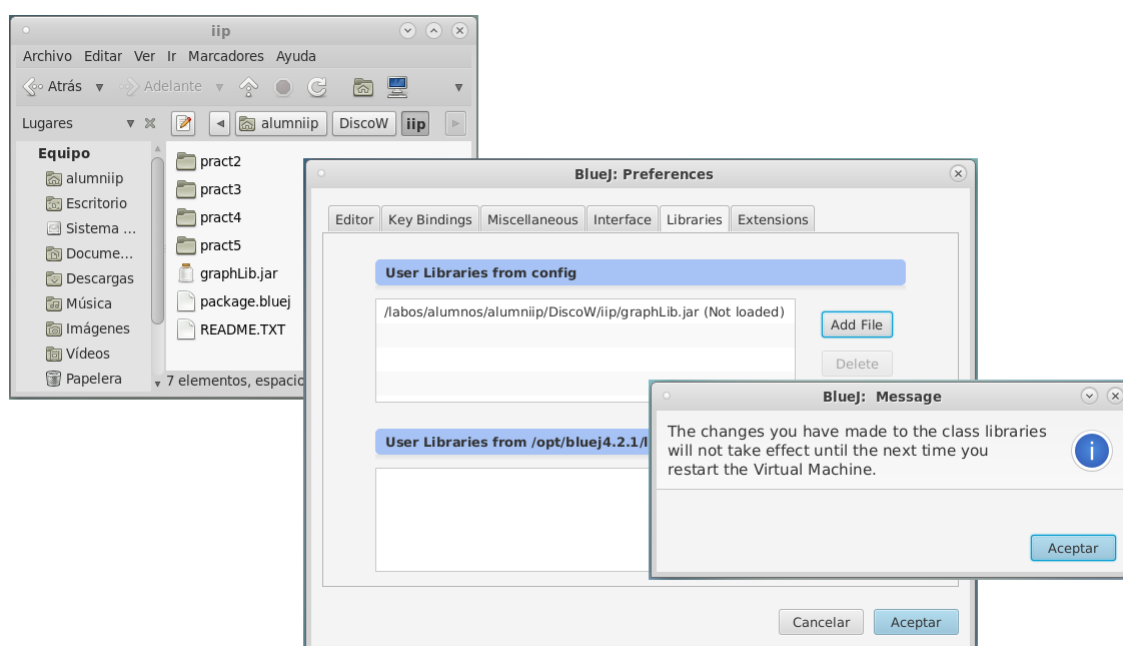


Figure 4: Installation of the graphical library in *BlueJ*.

The documentation of the class `Graph2D` is available in the file `docGraph2D.zip` you can download from *PoliformaT*. Extract the contents of this file in the folder of the project iip in order to have access to the documentation when needed, in particular to file `Graph2D.html`.

Activity 3: complete the Point data-type class

Complete the `Point` class in package `pract5`. You must complete where comments indicate. When finished, the class must include:

1. The object and class attributes described in Section 3.
2. A default constructor for creating a point in the origin, i.e. coordinates (0.0,0.0), and a generic constructor with two parameters, one for each attribute.
3. The `get` and `set` methods for each object attribute.
4. The `distance` for returning the Euclidean distance between the current point `this` and another passed as parameter.
5. The `move` method for updating the coordinates of the current point `this` to the values given as parameters.
6. The `equals` method for checking whether two objects of the class `Point` are identical, i.e., all their attributes have the same value.
7. The `toString` method for returning an object of the class `String` representing the current point `this` using the typical format: (x,y).
8. The `cross` method that implements the case analysis described in Figure 3 for computing if the ray parallel to x -axis and starting from the current point to $+\infty$ goes through the segment defined by the two points given as parameters.

Activity 4: complete the RayTest class - standard output

The *program* class `RayTest` is the class where the method `cross` from the class `Point` is tested. For performing the tests, the following methods must be implemented.

- Method `main` where three vertex will be declared and constructed:

```
Point vert1 = new Point(3.0, 16.0),  
      vert2 = new Point(3.0, 0.0),  
      vert3 = new Point(17.0, 0.0);
```

which are the vertex of the triangle shown in Figure 1, so that the segment delimited by `vert2` and `vert3` is parallel to rays and allow us to check the **CASE 1** described in the case analysis. The other two segments can be used to test the remaining cases. Additionally, some points are created for testing:

```
Point p1 = new Point(-1.0, 18.0), p2 = new Point(2.0, 16.0),  
      p3 = new Point(-4.0, 10.0), p4 = new Point(14.0, 7.0),  
      p5 = new Point(1.0, 0.0), p6 = new Point(-5.0, -6.0);
```

With these points we can test all the possible cases enumerated in Table 1.

In the method `main` you have to do a minimum of five calls to method `cross` in order to test all the possible cases of the algorithm for checking if a ray starting from one point goes through a segment. In the provided code you have the first example for testing **CASE 1**. For each possible case the output should be something similar to the following example, where the tested cases are: **1a**, **2b**, **3b**, **4b** and **5e** from Table 1:

```
Crossing of segment (3.0,0.0) to (17.0,0.0) from (-1.0,18.0) : DONT_CROSS  
Crossing of segment (3.0,16.0) to (17.0,0.0) from (2.0,16.0) : HIGH_CROSS  
Crossing of segment (3.0,16.0) to (17.0,0.0) from (1.0,0.0) : LOW_CROSS  
Crossing of segment (3.0,16.0) to (17.0,0.0) from (-4.0,10.0) : CROSS  
Crossing of segment (3.0,16.0) to (17.0,0.0) from (14.0,7.0) : DONT_CROSS
```

Table 1: Test cases for `p.cross(u, v)`.

CASE	Examples	u	v	p
1	a	vert2	vert3	cualquiera
2	a	vert1	vert2	p2
	b	vert1	vert3	p2
3	a	vert1	vert2	p5
	b	vert1	vert3	p5
4	a	vert1	vert2	p3
	b	vert1	vert3	p3
5	a	vert1	vert2	p1
	b	vert1	vert2	p4
	c	vert1	vert2	p6
	d	vert1	vert3	p1
	e	vert1	vert3	p4
	f	vert1	vert3	p6

If you like you can implement several test examples per case.

Notice that in this example the result of method `cross` is printed as an `String` from (`DONT_CROSS`, `HIGH_CROSS`, `LOW_CROSS`, `CROSS`) instead of as an integer. For obtaining the corresponding string from the integer returned by the method `cross` you have to use the method `crossToString` that must be completed according to the following description.

- Method `crossToString` must use the control flow instruction **switch** and the **constants defined in the class `Point`**. You must appropriately update the value of the local variable `res`.

Once this class is completed and tested, you can improve your code by implementing a static method `showCross` that, given three `Point` that represent a point and the ends of a segment, it includes the call to the `cross` method and the output that is performed for each of the tested cases.

Activity 5: complete the `RayTest` class - graphical output

By using the library `Graph2D` from package `graph2D`, all the test cases will be plotted graphically. This will facilitate students to visually check the results obtained in the previous activity.

Notice that for using this library the class `Graph2D` must be imported at the beginning of class `RayTest` by means of the compiler directive for importing:

```
import graph2D.Graph2D;
```

After importing a package or class it is already possible to declare and create objects of the imported classes. In the provided code you can find an example where an object of the class `Graph2D` is created with a viewport from $(-20.0, -20.0)$ as lower left corner to $(20.0, 20.0)$ as upper right corner. The window size in pixels is 600×600 , the background colour is white, and the title is `RAY TEST`.

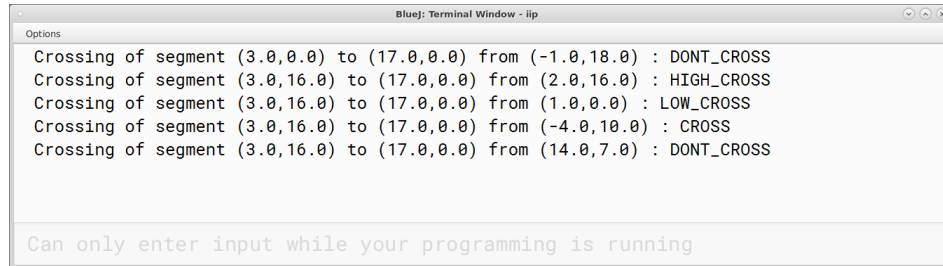
Next, the segment \overrightarrow{uv} is painted with the method `drawLine` as a red line from u to v with a line width of 3 pixels.

Finally, in the provided code, the static method `drawRay` from class `RayTest` is used for plotting the points to be tested and the corresponding rays, using methods `drawPoint` and `drawLine` respectively. The code of this static method is:

```
/** Draws the point p and its ray in the canvas gd. */
private static void drawRay(Graph2D gd, Point p) {
    gd.drawPoint(p.getX(), p.getY(), Color.BLACK, 4);
    gd.drawLine(p.getX(), p.getY(), 20, p.getY(), Color.GREEN, 1);
}
```

In this activity students must complete the plot of all the testing rays for checking the crossings with the segment \overrightarrow{uv} . As in the case where the results were sent to the standard output, you can find an example for **CASE 1** in the provided code.

The result of the graphical output of method `main` for the following test cases **1a**, **2b**, **3b**, **4b** and **5e** from Table 1, is the one shown by Figure 5(b).



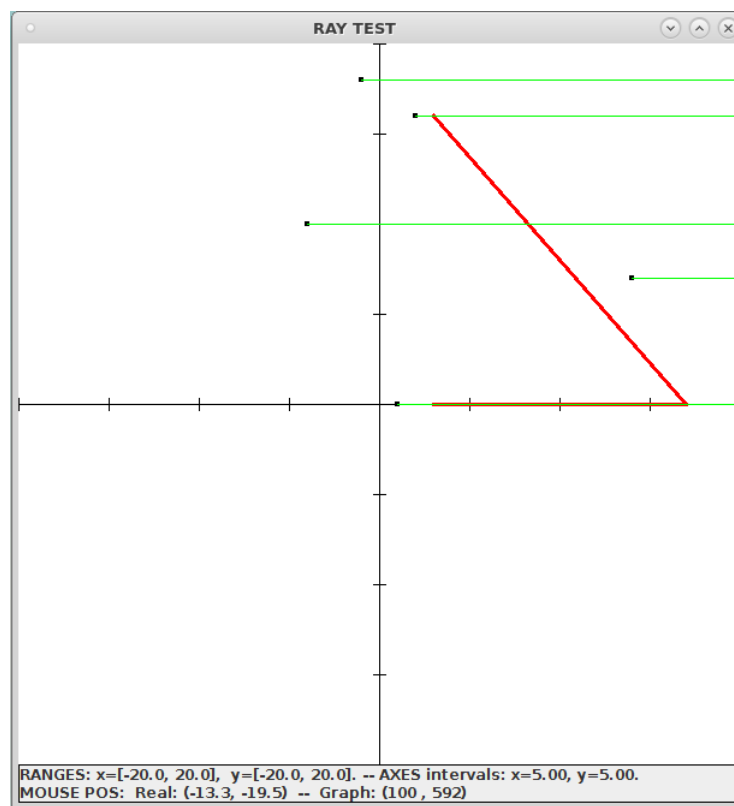
```

Options
Crossing of segment (3.0,0.0) to (17.0,0.0) from (-1.0,18.0) : DONT_CROSS
Crossing of segment (3.0,16.0) to (17.0,0.0) from (2.0,16.0) : HIGH_CROSS
Crossing of segment (3.0,16.0) to (17.0,0.0) from (1.0,0.0) : LOW_CROSS
Crossing of segment (3.0,16.0) to (17.0,0.0) from (-4.0,10.0) : CROSS
Crossing of segment (3.0,16.0) to (17.0,0.0) from (14.0,7.0) : DONT_CROSS

Can only enter input while your programming is running

```

(a) Standard output.



(b) Graphical output.

Figure 5: Result of running the method `main` for test cases **1a**, **2b**, **3b**, **4b** and **5e** from Table 1.

Activity 6: checking style for the Point and RayTest classes

Check that the implementation of `Point` and `RayTest` fulfils the coding style directives by using the `Checkstyle` of *BlueJ*, and correct the different warnings.

Activity 7: validate the class Point

When your teacher thinks it is the moment, (s)he will leave available in *PoliformaT* a test class for validating the code of your `Point` class. In general, for passing all the tests, your code must use the same identifiers for all the attributes and methods proposed in this document. Data-types and qualifiers must be strictly the same as indicated here or in the comments of the code. You must follow these steps:

1. Download to `pract5` directory the `PointUnitTest.class` file, and reopen the *BlueJ* iip project.
2. Select the option *Test All* from the pop-up menu that appears when you click on the icon of the class *Unit Test* with the right button of the mouse. As usual, a set of tests will be run for checking the behaviour of the methods of the `Point` class. You will see errors or warnings if the obtained results by using your implementation are not the expected ones.
3. Tests passed will be marked with the symbol ✓ in the window *Test Results* of *BlueJ*. Methods that do not pass a test will be highlighted with symbol X. You can select with the mouse lines marked with symbol X for obtaining a short description about the error.
4. If after correcting the errors and compile the class again, the icon in the *Unit Test* is crossed out with squares, then close and re-open the *BlueJ* project.