IIP (E.T.S. de Ingeniería Informática)
Academic year 2019-2020

Lab practice 7. Arrays: an application for managing groups of polygons in the plane

IIP Professors
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Contents

1 Objectives and previous work to be done before the first session of this lab practice

This lab practice summarises all the concepts studied in the subject, in particular the ones corresponding to the unit of **Arrays**.

Before starting the first session students must read all this lab guide in order to know all the classes that will be used in this lab practice, and do the previous work described in this section. It is very important to understand the problem to be solved.

This lab practice lasts three sessions in which all the scheduled activities can be performed. However, we recommend you to advance part of the work at home. During the first two sessions you have to complete classes `Polygon` and `PolygonGroup`. Then, in the last session you can improve the code if needed, complete the program `Test7` and perform test and corrections.

2 Problem description

This lab practice is about the management of groups of polygons located in the plane (2D space).

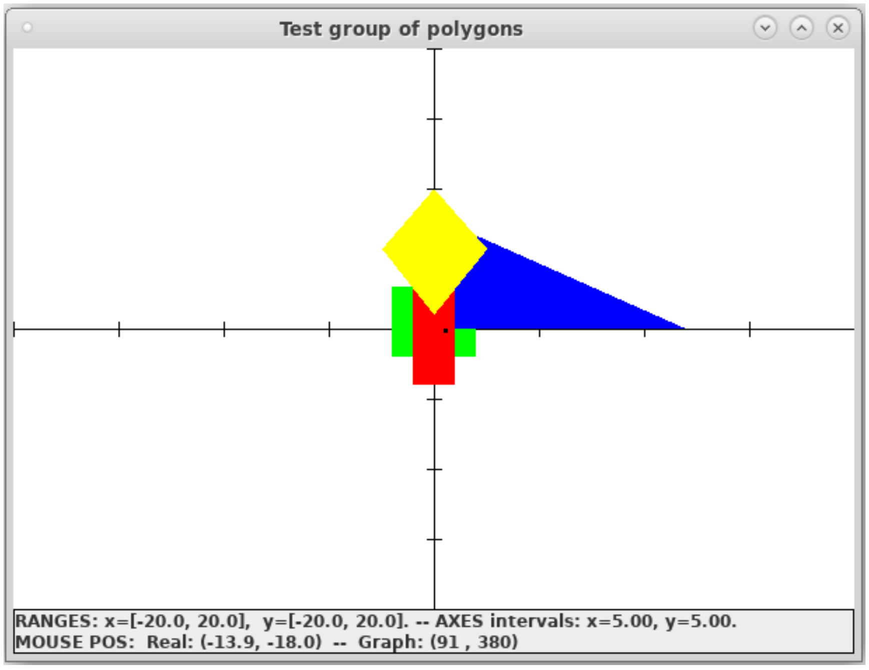
Each polygon is defined by a sequence of points in the two-dimensional space and fill colour. With these data the polygon can be painted in a canvas by using the `Graph2D` library. You are familiar with this library because it has been used in previous lab practices.

A polygon can be moved from one position to another and its fill colour can be changed.

In this lab practice you have to work with groups of polygons. A group of polygons will be stored in an array. Polygons can be added to or removed from the group. Some operations will be applied to all the polygons in the same group, for instance to change the colour of all the polygons in the group, or to move all the group from one location to another in the plane.

It is very common that some figures overlap, it this happens in group of polygons, the oldest ones must be on the bottom and the newest ones on the top. This is easily solved by painting then in the order they appear in the array, the oldest polygon (the first one added) will be at position 0, and the newest one in the last position. Figure 1 shows a group of polygons whose order is: a green rectangle, a blue triangle, a red rectangle, and a yellow quadrangle.

This order can be changed be selecting one of the polygons and pushing it to the bottom, or bringing it to the top, or any other operation.



```
RANGES: x=[-20.0, 20.0],  y=[-20.0, 20.0]. -- AXES intervals: x=5.00, y=5.00.
MOUSE POS:  Real: (-13.9, -18.0)  --  Graph: (91 , 380)
```

Figure 1: Group of polygons.

When a group of polygons has been represented, the way for selecting a polygon is by clicking with the mouse on a visible part of the polygon we want to select. In case of clicking on an area belonging to more than one polygon, only the polygon whose overlapped area is visible will be selected, no all the polygons which contain the point where we clicked. For achieving this the code will look for the selected polygon in descending order in relation to their position in the internal array used for storing all the polygons from a group. Figure 1 shows an example where the red polygon has been selected by clicking in the coordinates `(0.6,0.8)`.

In this lab practice you have to develop a small application for manipulating groups of polygons as described above. Figure 2 shows the classes of the application you have to develop. These classes are:

- Class `Polygon`. For manipulating polygons represented in the plane.
- Class `PolygonGroup`. For manipulating groups of polygons.
- Class `Test7`. Test program that creates a group of figures and allow us to perform different actions on it. In order to help students to understand what is done, the program visualises the results on a graphical windows by using the library `Graph2D` from package `graph2D`.

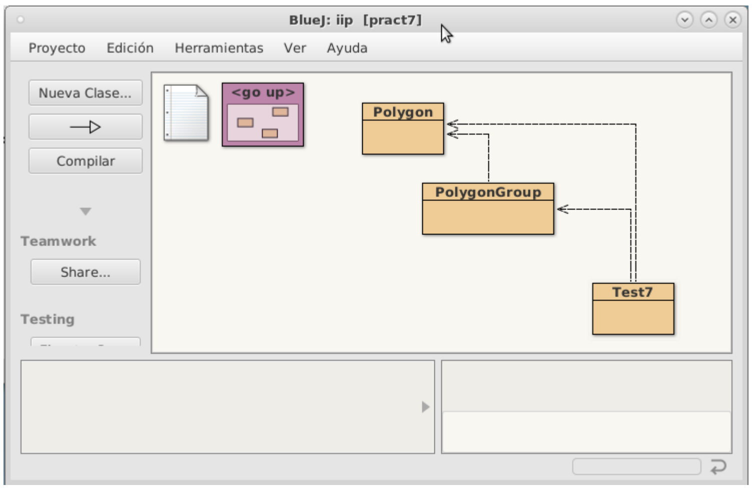


Figure 2: Classes of package `iip[pract7]`.

Activity 1: preparation of the BlueJ package `pract7`

1. Download files `Polygon.java`, `PolygonGroup.java` and `Test7.java` from the folder in *PoliformaT* corresponding to lab practice 7.
2. Open the *BlueJ* project for this subject (`iip`) and create a new package named `pract7`.
3. Add to the package `pract7` the downloaded classes with the option (*Edit - Add class from file*). Check that the first line of all the Java files is `package pract7;`

3 Class `Polygon`

A `Polygon` is defined by a sequence of n vertices, $v_0, v_1, \ldots, v_{n-1}$, so that its sides are the segments $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_{n-2}v_{n-1}}, \overline{v_{n-1}v_0}$. All polygons have a fill colour.

This class must have the following private attributes (instance variables):

- `v`: an array of objects of the class `Point` (this class was implemented in the lab practice 5). This array will store the sequence of n vertices, where each vertex is an object of the class `Point`. The length of `v` must be n, the number of vertices, which is also the number of sides.
- `color`: filling colour, an object of the class `Color` from the package `java.awt`.

Figure 3 shows an example of a `Polygon` with its graphical representation.

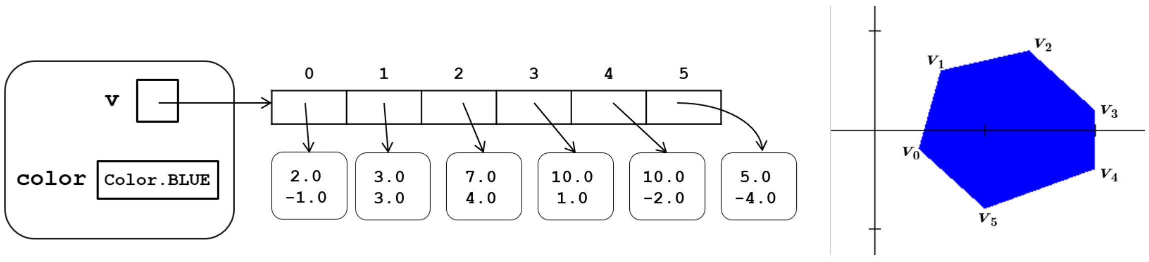


Figure 3: A `Polygon` and its graphical representation.

Methods of class `Polygon` are the following ones:

- Constructor `public Polygon(double[] x, double[] y)`: Creates a `Polygon` from array `x` with the abscissas $\mathtt{x}_0$, $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_{n-1}$ of the vertices, and array `y`, with the ordinates $\mathtt{y}_0$, $\mathtt{y}_1$, $\mathtt{y}_2$, ..., $\mathtt{y}_{n-1}$ of the vertices, where $n > 2$. Vertices define a polygon whose sides are segments of line between pairs of consecutive vertices. The polygon is closed by the segment defined from the last point to $(\mathtt{x}_0, \mathtt{y}_0)$. For instance, if `x` is the array `{2.0, 3.0, 7.0, 10.0, 10.0, 5.0}` and `y` the array `{-1.0, 3.0, 4.0, 1.0, -2.0, -4.0}`, the constructor must create the object shown by Figure 3.

  Blue is the default colour: `Color.BLUE`.

- Methods `public Color getColor()` and `public void setColor(Color nC)`, getter and setter, respectively, for attribute colour.

- Methods `public double[] verticesX()` and `public double[] verticesY()` that return an array with abscissas or ordinates of the polygon.

- Method `public void translate(double incX, double incY)`, that moves all the vertices of the polygon from its current position to the new one calculated by adding `incX` to the x-axis coordinates and `incY` to the y-axis coordinates of all the vertices.

- Method `public double perimeter()`, that returns the perimeter of the polygon.

- Method `public boolean inside(Point p)`, that checks if `Point p` is inside the polygon by using the `ray casting algorithm` discussed in appendix A.

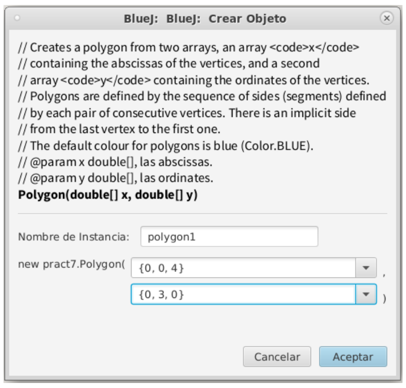
  If the point is inside returns `true`, otherwise returns `false`.

Activity 2: implementation and test of the class `Polygon`

Declare the attributes and implement the methods of the class `Polygon`. Notice that for using classes `Point` and `Color` it is necessary importing them, the following two lines are one of the possible ways for importing them.

```
import java.awt.Color;
import pract5.Point;
```

Methods will be checked as you complete its code. First, check the constructor by creating a triangle with vertices $(0, 0)$, $(0, 3)$ y $(4, 0)$, and inspecting the created object, as Figure 4 shows.

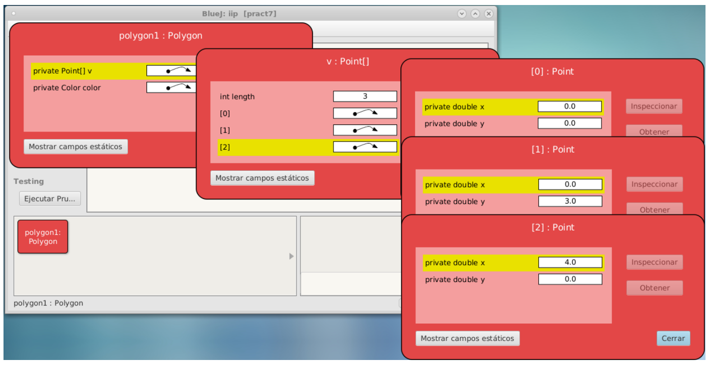


BlueJ: BlueJ: Crear Objeto

```
// Creates a polygon from two arrays, an array <code>x</code>
// containing the abscissas of the vertices, and a second
// array <code>y</code> containing the ordinates of the vertices.
// Polygons are defined by the sequence of sides (segments) defined
// by each pair of consecutive vertices. There is an implicit side
// from the last vertex to the first one.
// The default colour for polygons is blue (Color.BLUE).
// @param x double[], las abscissas.
// @param y double[], las ordinates.
Polygon(double[] x, double[] y)
```

Nombre de Instancia: polygon1

new pract7.Polygon({0, 0, 4} , {0, 3, 0})



BlueJ: iip [pract7]

polygon1 : Polygon
private Point[] v
private Color color

v : Point[]
int length 3
[0]
[1]
[2]

[0] : Point
private double x 0.0
private double y 0.0

[1] : Point
private double x 0.0
private double y 3.0

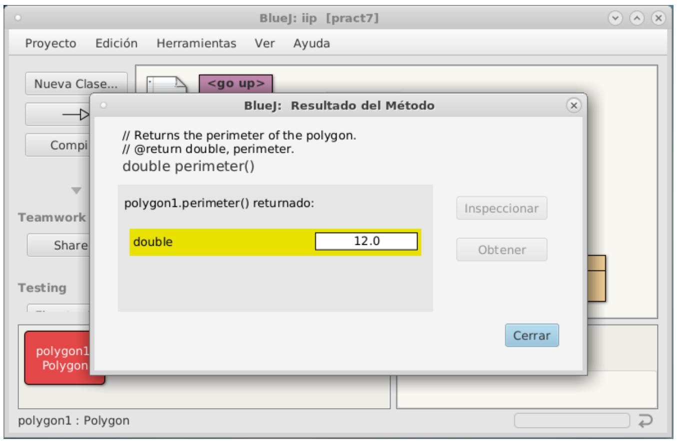
[2] : Point
private double x 4.0
private double y 0.0

Figure 4: Creation and inspection of a `Polygon` in *BlueJ*.

Next, all the instance methods must be completed and tested. In particular **`verticesX`**, **`verticesY`**, **`translate`** and **`perimeter`**, because these methods manipulate the vertices of the polygon. You can see an example in Figure 5.

In the case of method `translate`, if you move it `1.0` in x-axis and `1.0` in y-axis, then you have to check that methods `verticesX` and `verticesY` return arrays `{1.0, 1.0, 5.0}` and `{1.0, 4.0, 1.0}` respectively.

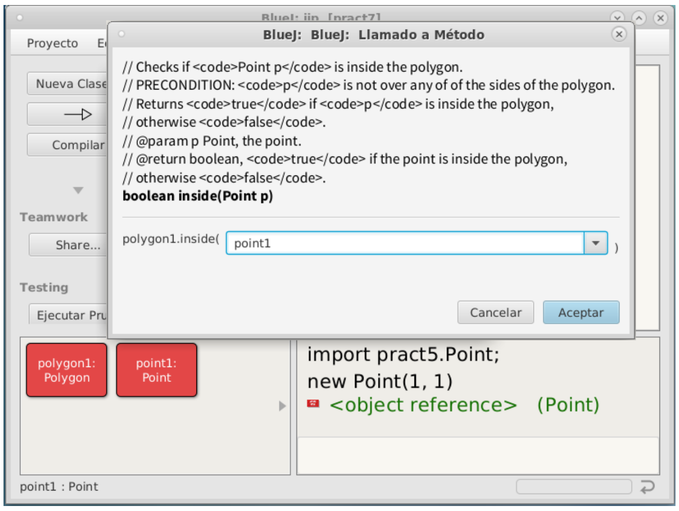
For testing method `inside` you have to import in the code zone the class `Point` from package `pract5`, then create a `Point` with coordinates $(1, 1)$ and drag up it to the object workbench for using it as parameter to the method, as Figure 6 shows. As the point is inside the triangle, the result must be `true`. Repeat the test with point $(4, 5)$, an external point, and with point $(-1, 3)$, an external point for which the ray goes through vertex $(0, 3)$ of the triangle.



```
// Returns the perimeter of the polygon.
// @return double, perimeter.
double perimeter()

polygon1.perimeter() returnado:
double    12.0
```

Figure 5: Testing method `perimeter` on triangle from Figure 4.



```
// Checks if <code>Point p</code> is inside the polygon.
// PRECONDITION: <code>p</code> is not over any of of the sides of the polygon.
// Returns <code>true</code> if <code>p</code> is inside the polygon,
// otherwise <code>false</code>.
// @param p Point, the point.
// @return boolean, <code>true</code> if the point is inside the polygon,
// otherwise <code>false</code>.
boolean inside(Point p)

polygon1.inside( point1 )
```

```
import pract5.Point;
new Point(1, 1)
<object reference>   (Point)
```

Figure 6: Testing method `inside` on triangle from Figure 4 and `Point (1.0,1.0)`.

4 Class PolygonGroup

A `PolygonGroup` is defined by a sequence of polygons. The order of polygons is the order in they were added into the group, from oldest to newest. All groups of polygons have a maximum capacity, i.e., they can contain a predefined maximum number of polygons.

Attributes of class `PolygonGroup` are the following ones:

- `MAX`: public constant and class attribute of type `int` for indicating the maximum capacity of the groups of polygons. It must be set to 10.

- `group`: private attribute (instance variable). An array of objects of the class `Polygon` whose length must be `MAX`. Polygons will be stored in this array from position 0 to `size-1`. Non used positions of the array must be `null`.
- `size`: private attribute (instance variable). Size of the group, i.e., the number of polygons stored in the array.

Figure 7 shows an object of the class `PolygonGroup` corresponding to the group of polygons represented by Figure 1.

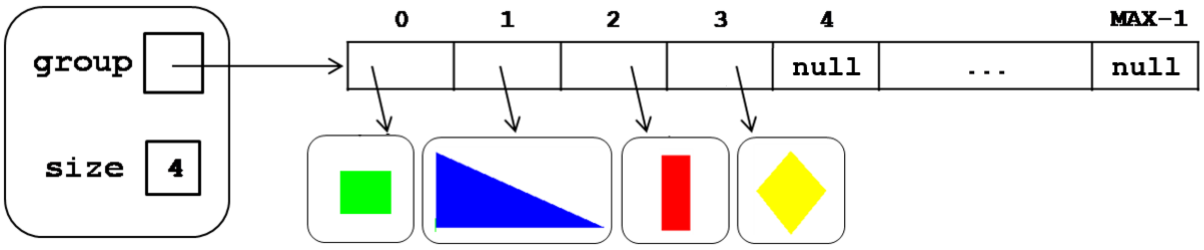


Figure 7: A `PolygonGroup` with size equal to 4.

Methods of class `PolygonGroup`:

- Constructor `public PolygonGroup()`: creates an empty group of polygons by setting `size` to zero and creating the array `group` with size set to `MAX`. All the elements of the array are set to `null` by default, remember what the operator `new` does.
- Method `public boolean add(Polygon pol)`, for adding a polygon into the group. The new polygon must be set to the first free position in the array, what means it will be on the top. The method returns `true`. If the group was full (`size` == `MAX`), then the new polygon must not be added into the group. In this case, this method returns `false`.
- Method `public int getSize()`, getter for the size.
- Method `private int search(Point p)`, that searches in descending order the first polygon containing `p`. Returns the position of the found polygon or -1 if the point is not inside any of the polygons of the group.

  This method will be used in methods `remove`, `toFront`, `toBack` and `translate` for finding the position in the array of the polygon selected by point `p`.
- Method `public boolean remove(Point p)`, that removes the polygon selected by point `p` from the group and it returns `true`. If none of the polygons of the group are selected by the point `p`, this method returns `false`.

  Notice that the implementation of this method must ensure that the remaining polygons in the group are arranged in consecutive positions of the array, from position 0 to `size-1`. And that all position in the array from `size` to `MAX-1` must be set to `null`.

  The following three methods do not do anything if none of the polygons of the group are selected by the point `p`.
- Method `public void toFront(Point p)` sets the the selected polygon on the top of the group, i.e. brings it to the front.
- Method `public void toBack(Point p)`, sets the the selected polygon on the bottom of the group, i.e. sends it to the back.

- Method `public void translate(Point p, double incX, double incY)` moves the selected polygon from its current position in the plane to the new position calculated by adding `incX` to the abscissas and `incY` to the ordinates of all the vertices.

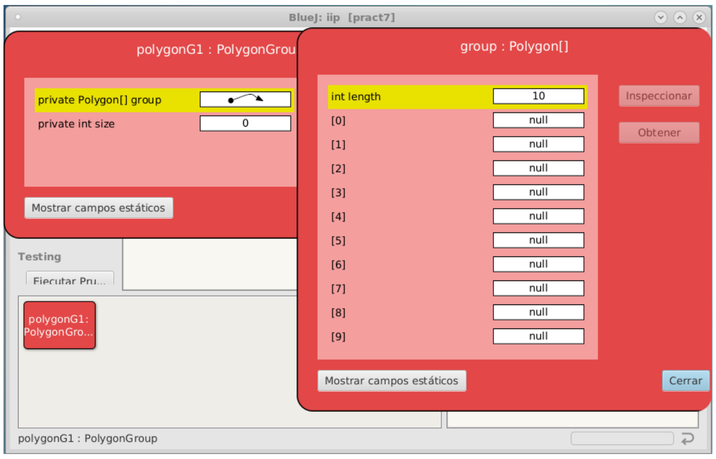
- Method `public Polygon[] toArray()` returns an array of size equal to the size of the group (not the size of the internal array) containing all the polygons arranged as they are in the group.

Activity 3: implementation and test of class `PolygonGroup`

Declare the attributes and implement the methods of class `PolygonGroup`. Notice that classes `Point` and `Color` must be imported for using them. As in the class `Polygon`, the following two compiler directives must be used.

```
import java.awt.Color;
import pract5.Point;
```

You must test methods as you complete them. First test should be to create and inspect an empty group of polygons, as Figure 8 shows.


BlueJ: iip [pract7]

polygonG1 : PolygonGrou

private Polygon[] group	●⟶
private int size	0

Mostrar campos estáticos

Testing

Ejecutar Pru...

polygonG1: PolygonGro...

group : Polygon[]

int length	10
[0]	null
[1]	null
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null

Inspeccionar

Obtener

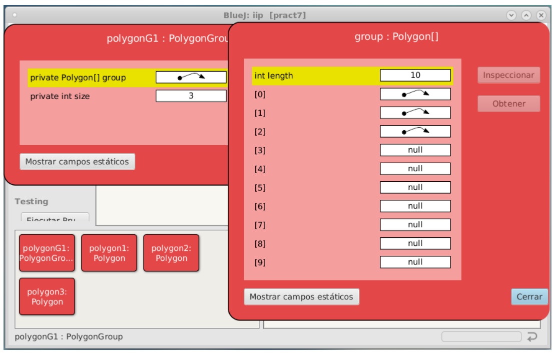
Mostrar campos estáticos

Cerrar

polygonG1 : PolygonGroup

Figure 8: Test of the constructor of the class `PolygonGroup`.

For testing methods **`add`**, **`size`**, **`toArray`**, **`search`** and **`remove`**, three identical triangles will be created with coordinates $(0,1),(2,2)$ y $(0,3)$ (array of abscissas `{0.0, 2.0, 0.0}` and array of ordinates `{1.0, 2.0, 3.0}`). These three 100% overlapping triangles must be added to the group step by step, and you have to inspect the object of the class `PolygonGroup` as you add each triangle. Figure 9 shows the state you have to observe after adding the three polygons.


BlueJ: iip [pract7]

polygonG1 : PolygonGrou

private Polygon[] group	●⟶
private int size	3

Mostrar campos estáticos

Testing

Ejecutar Pru...

polygonG1: PolygonGro...

polygon1: Polygon

polygon2: Polygon

polygon3: Polygon

group : Polygon[]

int length	10
[0]	●⟶
[1]	●⟶
[2]	●⟶
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null

Inspeccionar

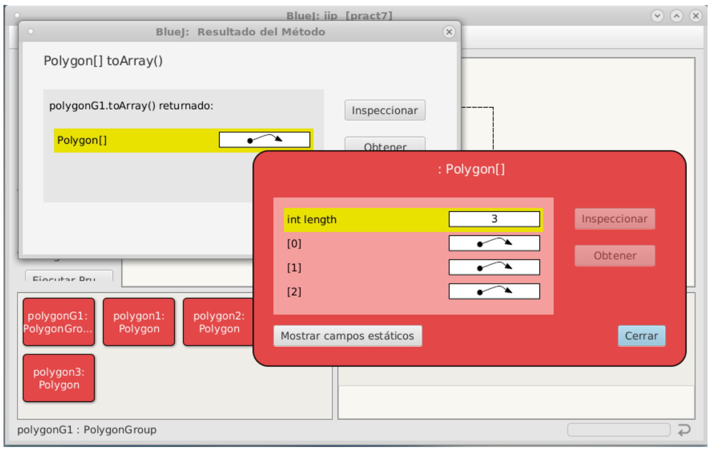
Obtener

Mostrar campos estáticos

Cerrar

polygonG1 : PolygonGroup

Figure 9: Testing method **`add`** of `PolygonGroup` and inspecting the result.



BlueJ: Resultado del Método

Polygon[] toArray()

polygonG1.toArray() returnado:

Polygon[]

: Polygon[]

int length	3
[0]	
[1]	
[2]	

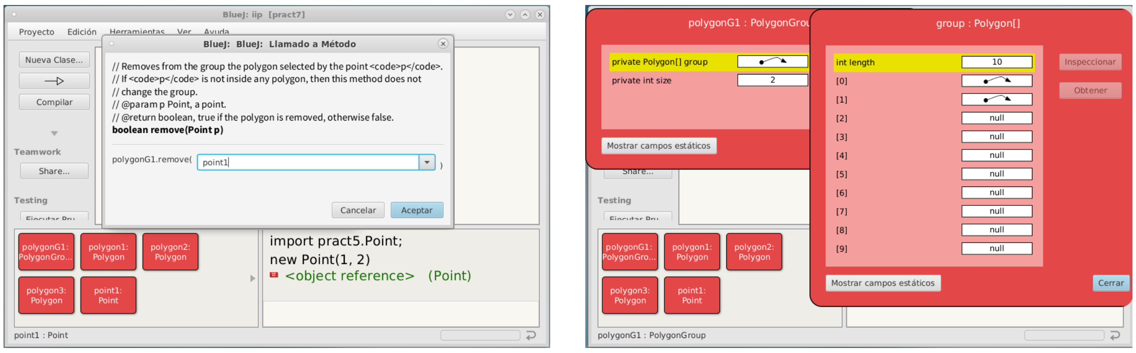
polygonG1 : PolygonGroup

Figure 10: Testing method `toArray` of `PolygonGroup`.

Next, check that the method `toArray` returns an array with three polygons (see Figure 10).

Next, three removal operations must be performed for removing the first polygon containing the point $(1, 2)$. The group must be inspected after each removal operation. As the three triangles are identical, so all three contain the point $(1, 2)$, you must check that the first removed triangle is the one at position 2 in the array, then the one at position 1, and finally the one at position 0. After the three removal operations the group must remain empty.

Figure 11 shows the state of the group after the first removal operation.



BlueJ: BlueJ: Llamado a Método

```
// Removes from the group the polygon selected by the point <code>p</code>.
// If <code>p</code> is not inside any polygon, then this method does not
// change the group.
// @param p Point, a point.
// @return boolean, true if the polygon is removed, otherwise false.
boolean remove(Point p)
```

polygonG1.remove(point1)

```
import pract5.Point;
new Point(1, 2)
<object reference>   (Point)
```

point1 : Point

polygonG1 : PolygonGroup

private Polygon[] group	
private int size	2

group : Polygon[]

int length	10
[0]	
[1]	
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null

polygonG1 : PolygonGroup

Figure 11: Test of search and removal operations of a polygon: call to method `remove` of `PolygonGroup` and inspection of the result.

After these preliminary tests, you can carry out a more exhaustive set of tests for the methods of this class by using the program `Test7` explained in the following section.

5 Class `Test7`

Class `Test7` is a program-class that performs several tests on a group of polygons and facilitates the graphical representation of the actions. For doing it this class has the method `drawGroup` that paints a group of polygons on a canvas by using the `Graph2D` library. This class also uses the method `nextMousePressed` of the class `Graph2D` for getting the coordinates of the point inside the canvas you (the user) click with the mouse. Thanks to that it is possible to select a polygon of the group.

In this class it is created a group of polygons, as Figure 1 shows, for performing the tests.

Activity 4: completing and testing method `main` of class `Test7`

This class includes methods `menu(Scanner)` and `submenu(Scanner)` for allowing the user to choose among different operations to test on the group of polygons. You have to read carefully method `main` and complete the code in order to execute the corresponding method of the class `PolygonGroup` according to the selected action.

Next, run the program as many times as needed in order to carry out all the possible tests thanks to the different options of the menu.

6 Validation of the classes

When your teacher thinks it is the moment, (s)he will leave available in *PoliformaT* some test classes for validating your code. For that you will have to download the corresponding files into the folder corresponding to package `pract7` and re-open your project `iip` in *BlueJ*.

In general, for passing all the tests, your code must use the same identifiers for all the attributes and methods of the classes proposed in this document and in the comments inside the `.java` files. Data-types and qualifiers must be strictly the same are indicated here or in the comments of the code.

Activity 5: validation of the class `Polygon`

1. Method `cross` from class `Point` must be tested in order to use it with guarantees in the method `inside` of the class `Polygon`. To test it you have to use the validation test `PointUnitTest.class` in the package `pract5`.

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 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.

Activity 6: validation of the class `PolygonGroup`

For testing the class `PolygonGroup` you have to select option *Test All* from the pop-up menu. Remember that pop-up menus appear when clicking the right button of the mouse. In this case by clicking it when the mouse pointer is over the icon of the *Unit Test* corresponding to the class to be tested.

All methods of the class `PolygonGroup` will be tested. It is required that classes `Point` and `Polygon` run correctly. They will do it if they are validated as indicated by the previous activities.

A Method `inside`

As described in lab practice 5, the *ray casting algorithm*, based on the Jordan theorem, allows us to easily check if a point is inside or outside a polygon. The procedure is the following: a ray is thrown from the point for counting how many times the ray crosses the perimeter of the polygon, i.e., the number of segments crossed by the ray. If the counter is an even number, then the ray went in and out the same number of times, so the point is outside the polygon, otherwise the point is inside the polygon because the ray went out one time more than the number of times it went in.

Figure 12 shows an example of how a ray thrown from a point inside the polygon crosses the perimeter an odd number of times.

The method `inside` can be implemented by checking all the sides of the polygon and accounting the number of crossed sides. For this method to work correctly, the special case when a ray crosses a vertex must be considered. As a vertex belongs to two sides of the polygon, in this case we have to take into account the following possibilities:

- As Figure 12 shows, vertex v_8 of the polygon connects two sides or segments, one segment in its lowest point and the other segment in its highest point. In this case the ray crosses the perimeter once, so the algorithm must account this case as one cross.
- But, in the case of vertices v_3 and v_6 of the same polygon, these vertices connect two segments by the highest/lowest ends of both segments. This case must not be accounted because it is similar to add 2.
- Notice that if the ray is goes through a side parallel to the x-axis, nothing must be accounted.

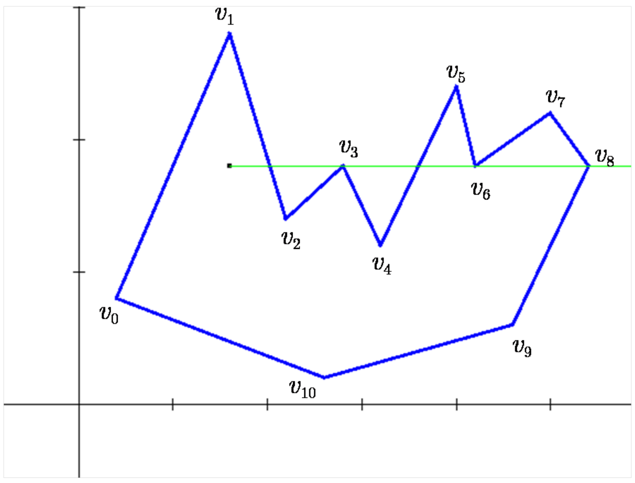


Figure 12: An example of the ray casting algorithm.

Hint: in order to simplify the algorithm for taking into account all the possibilities described above, we can decide to account one cross when the ray crosses one side at its lowest end and not to account when the ray crosses one side at its highest end.

The following code snippet accounts the number of crossings to the sides of a polygon by a ray thrown from a point `p` by using the method `cross` of the class `Point` and following the described criterion.

```
int nCuts = 0, n = v.length;
int cross;
for (int i = 0; i < n - 1; i++) {
    cross = p.cross(v[i], v[i + 1]);
    if (cross == Point.CROSS || cross == Point.LOW_CROSS) { nCuts++; }
}
cross = p.cross(v[n - 1], v[0]);
if (cross == Point.CROSS || cross == Point.LOW_CROSS) { nCuts++; }
```

So that, the implementation of the method `inside` of the class `Polygon` must perform this counting, and return `true` if `nCuts` is an even number, and `false` otherwise.