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

# *Lab activity 2. Objects, classes, and programs. The BlueJ environment*

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

## Contents



## 1 Context and previous work

In the academic framework, this is the second lab activity of the course, and its main objective is the definition and use of simple classes by using the *BlueJ* integrated development environment (IDE). With this activity the student must get familiar with the basic aspects of *BlueJ*, and employ it from now as working environment for the next lab activities. The student must be able to employ the IDE for:

- creating a *BlueJ* project including a package with the classes that are given at the beginning of the activity,
- editing some of the existing classes,

- creating objects and executing methods on them by using the *BlueJ* Object Bench,
- evaluating expressions by using the *BlueJ* Code Pad,
- generating automatically the documentation for the project,
- validating the correctness of the class or any method of it, by using the *BlueJ* Debugger.

To obtain the maximum performance during the lab sessions, the student must read this report before coming to the lab.

# 2 Developing a *BlueJ* project

As was indicated in lab activity 1 (*Introduction: Linux, Java, and BlueJ*), a *BlueJ* project consists of a set of related classes. *BlueJ* not only allows to develop the project (create, compile, execute, debug, document), but *allows to interact with any method (attribute or object) of any of the classes of the project* as well. Moreover, in a *BlueJ* project, the different class that compose it could be organised in *packages* or *class libraries*, following the class organisation of Java and allowing their posterior reusing from other classes.

The aim of this activity is getting the student familiar with all these concepts using the *BlueJ* environment.

## 2.1 Calling *BlueJ* and creating a project

As explained in the previous lab activity, *BlueJ* can be called from the system menu (`Aplicaciones - Programación - BlueJ 4.2.1`) or from the command line, with or without arguments:

`bluej &` or `bluej projectname &`

Notice that the second option is equivalent to call *BlueJ* without arguments and then, by using the option `Project - Open` from the menu, selecting the project `projectname`. If you want to open in *BlueJ* an existing Java application **which was not developed with *BlueJ***, then call *BlueJ* without arguments and open the existing application (the directory where its classes are stored) with the menu option `Project - Open Non BlueJ...`.

**Note:** *BlueJ* environment language can be changed by using the option `Tools - Preferences - Interface`.

### Activity #1

1. Download the `iip.jar` file available in the PoliformaT folder `Recursos/Laboratorio/Práctica 2/English/Code` in your `$HOME/DiscoW` directory.
2. Call *BlueJ* without arguments.
3. Open the file `iip.jar` with the option `Project - Open ZIP/JAR...`. Then, the file will be decompressed and new files and directories will appear at `$HOME/DiscoW`

`iip` is a *BlueJ* project with a package called `pract2`. In *BlueJ* main window appear the icons of each class of the package (Figure 1) and an icon of a folder with the text `<go up>` that allows to access to the `iip` project. Notice that the icons associated to associated to `.java` files appear with strips when they have not been compiled. The arrows show the use relations between classes. Notice that the icons associated to `.class` files appear with the text `(no source)`. When only the `.class` files are present the classes can be used, but there is no source code to modify.

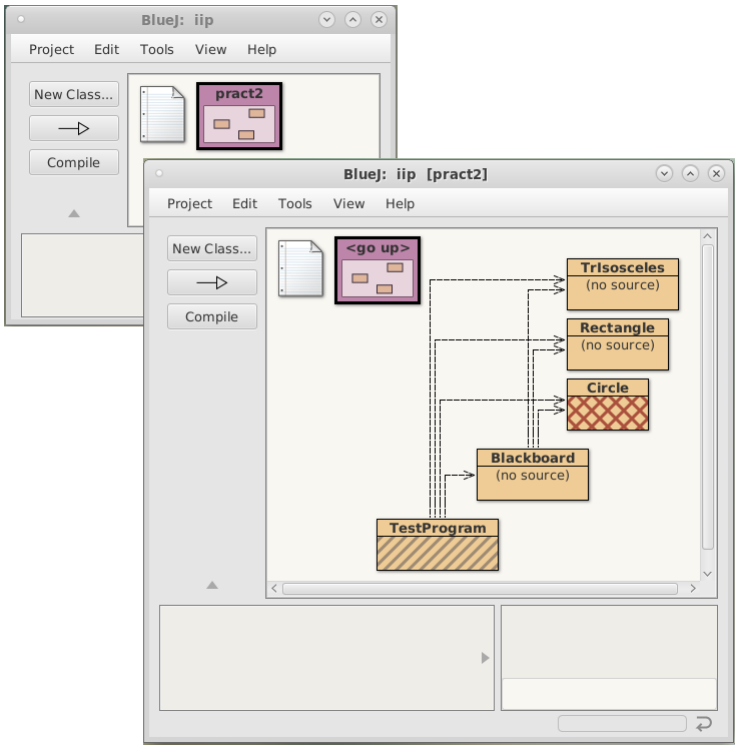



Figure 1: Project `iip` including package `pract2`.

## 2.2 Edition and compilation

For editing a class, in our case `Circle`, the option `Open Editor` from the menu class can be selected. As an alternative, double click on the class icon can be used. A class can be compiled from the editor or, *when all the project is going to be compiled*, from the tool bar of the main window of *BlueJ*. *BlueJ* performs a precompilation of the code. If small syntactic errors are detected the editor shows these errors with a red mark at the left side and a small underlined. For example, if the compiler detects that any semicolon is missing (Figure 2). A black box with a message appears if the cursor is located on the underlined word or when the button with the text "`Errors`" (right part of the information zone) is clicked. After precompilation you can compile the class and more errors could be detected.

## Activity #2

1. Check that the first line of classes `Circle` and `TestProgram` include the compiler directive that indicates that are classes pertaining to a package:

   ```
   package pract2;
   ```

2. Compile the class `Circle` and correct the compilation errors.

3. Compile the class `TestProgram`. Check what happens in the central part of the *BlueJ* main window.

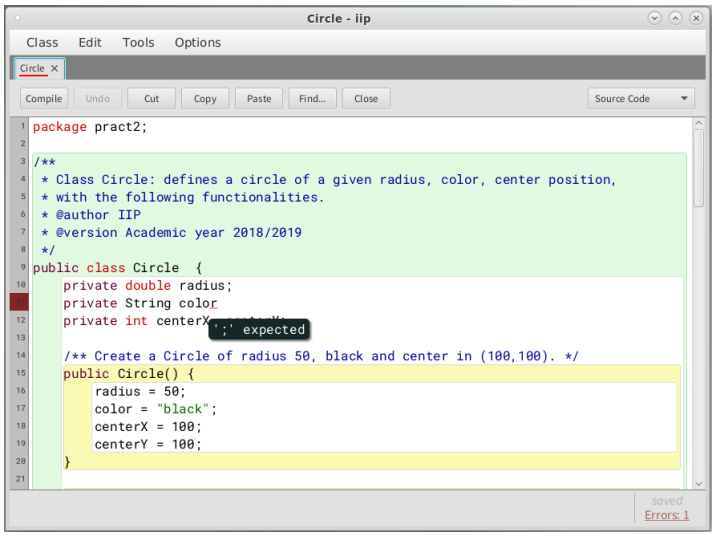



Figure 2: Compilation of class `Circle`.

One of the features offered by the *BlueJ* editor is the code autocompletion, which is activated by pressing `Ctrl - Space`.

### Activity #3

Although the `TestProgram` class has no compilation errors, indeed has a logical error: the programmer wanted to show the perimeter of the created circle on the screen, but actually shows a wrong value. In this activity, that error must be corrected.

1. Edit the `TestProgram` class and complete the instruction that shows on the screen the perimeter of the object `Circle c`. Write `c.` and press `Ctrl-Space`. Check that, as shown in Figure 3, a list of all the possible methods appears, along with a short description of each of them. Select the suitable one and press `Enter` to make *BlueJ* to autocomplete it.

Then, from the option `Tools - Checkstyle` in the project view, check if the code of some class has style errors and correct them if it is the case (Figure 4).

## 2.3 Class execution: calling `main`

As was said before, when a class is clicked by using the mouse right button, the class menu appears. From the operations that shows this menu, the call to the `main` method must be highlighted, since *allows to execute a class from its menu.* In the case of the `TestProgram` class, no arguments are necessary for main (only the braces that appear by default are enough).

In Figure 5(a) and 5(b) are shown, respectively, the pop-up menu for the class `TestProgram`, and the call window for the `main` method of the `TestProgram` class.

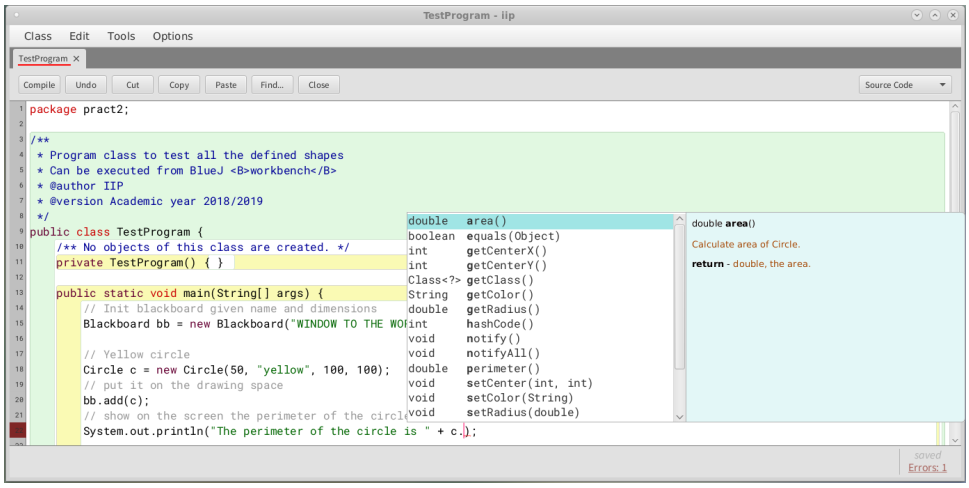



Figure 3: Code autocompletion in *BlueJ* editor.

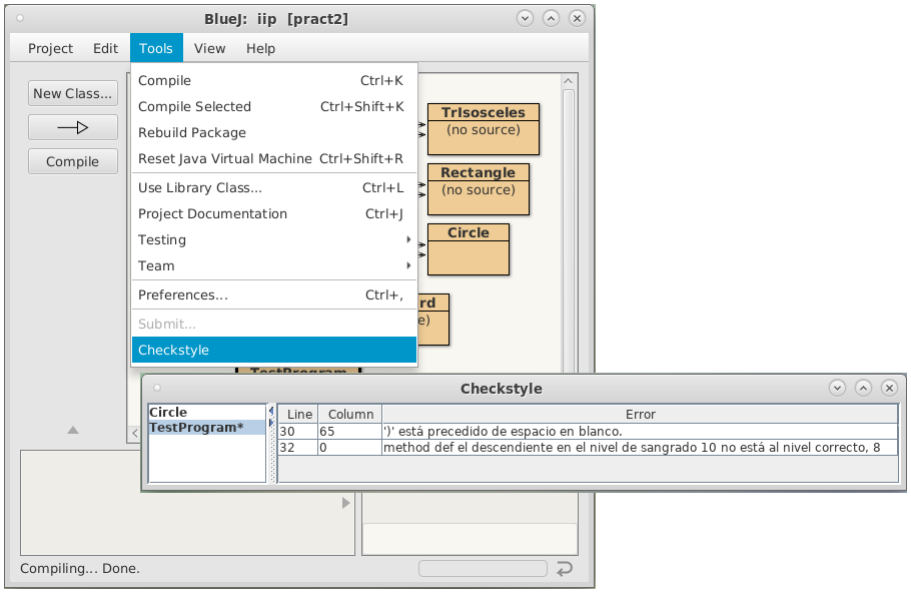
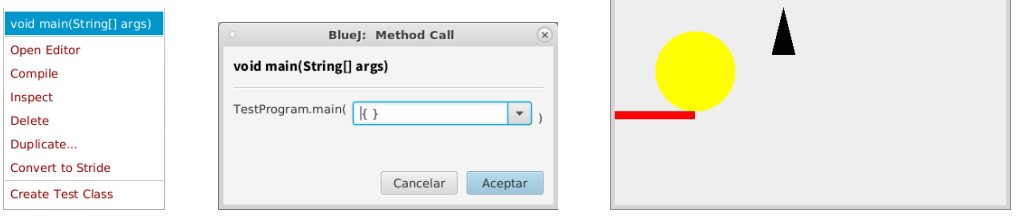



Figure 4: Checking code style in *BlueJ*.

## Activity #4

1. Execute the `main` method of the class `TestProgram`. The result must be similar to that that is shown in the Figure 5(c), and terminal must show a message similar to that in Figure 6.

When a program requires data to be inputted by keyboard or must show any result on the screen (Figure 6), automatically appears a text terminal. If it does not appear, choose the `Show terminal` option on the menu `View`. Note that, in version 4.2.1 of *BlueJ*, when you have to enter data from the keyboard, it is done in an additional line at the bottom of the terminal window, if the program does not need an input, it appears dimmed.




(a) Class menu. (b) `main` method call. (c) Result of the execution

Figure 5: Execution of the class `TestProgram`.

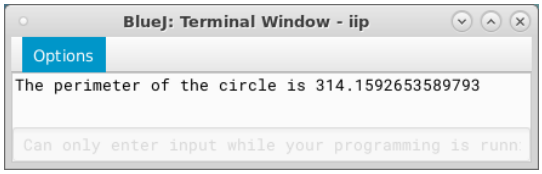



Figure 6: *BlueJ* terminal window.

If the `main` of `TestProgram` is executed again, you can verify that the message is rewritten in the terminal window. So that, in each new execution, the result of the previous execution is deleted from the terminal, you must select, from the menu `Options` of the terminal window, the option `Clear screen at method call`, as in the Figure 7. For future lab activities, also select the `Unlimited buffering` option.

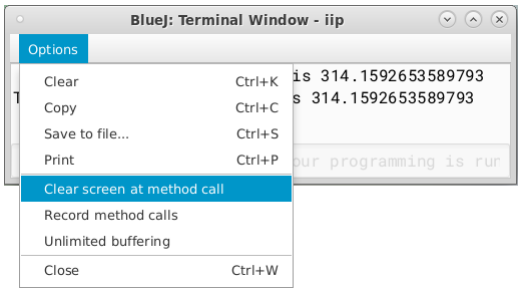



Figure 7: Options in *BlueJ* terminal window.

## 2.4 *Tools* option. Documentation generation

From the different utilities for the `Tools` option, the most important is `Project Documentation`. This utility generates the subdirectory `doc` with the documentation on the classes in `html` format. Notice that documentation for an individual class can be generated during class edition, by selecting `Documentation` instead of `Source code` or selecting `Tools - Toggle Documentation View`.


Figure 8: Documentation for project `pract2`.

### Activity #5

1. Generate the project documentation and consult it. The result must be similar to that which appears in Figure 8. Clicking, for example, in the link of the `Circle` class you can check its documentation.

## 2.5 *Help* option

The utility `Java Class Libraries` allows to access the Java documentation. In the default instalation this help is in `https://docs.oracle.com/en/java/javase/11/docs/api/`. It could be changed by a local directory, modifying the option in `Tools - Preferences - Miscellaneous`.

Other utilities from `Help` allow to consult the *BlueJ* manual, as well as accessing its web site `www.bluej.org`.

# 3 Using the Object Bench

One of the most interesting features of the *BlueJ* IDE is its capability to interact with isolated objects from any class and execute the methods defined on these objects; thus, the functionality of a class can be checked before writing any application that uses it.

## 3.1 Class operations

To access to the possible operations, the class icon must be selected and clicked with the mouse right button. Then, it appears a list of the constructors of the class, along with other operations permitted by the IDE - such as removing or compiling the class (Figure 9).

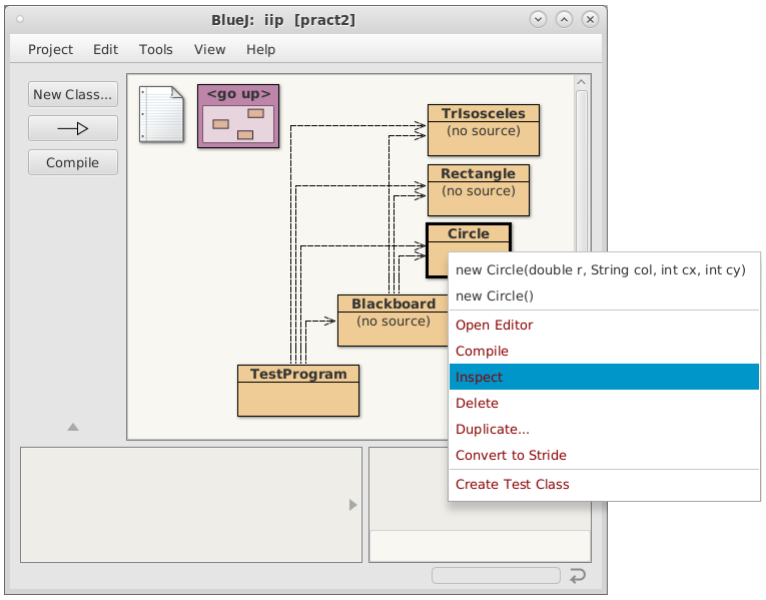



Figure 9: Menu for the class `Circle`.

## 3.2 Object creation

To create an object, a constructor must be selected from the pop-up menu, and follow the dialogue box which gets opened (Figure 10(a)). Specifically, a name for the object is demanded. When the object is created, it appears in the bottom left corner of the *BlueJ* main window, in a zone known as *Object Bench* (Figure 10(b)).

## 3.3 Object methods execution

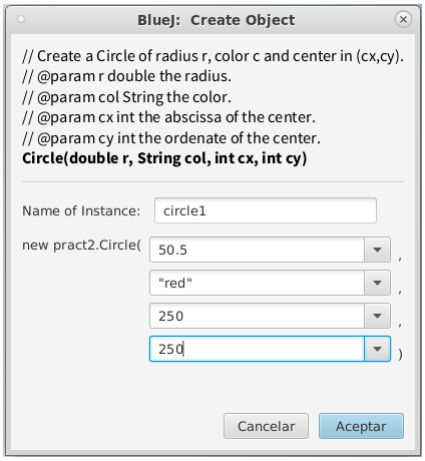
When the created object is clicked with the right button, the methods that can be executed on it can be accessed (Figure 10(b)). To execute any of them it must be selected. When the object inherits methods from other classes, they appear in submenus.

## 3.4 Watching the state of the object

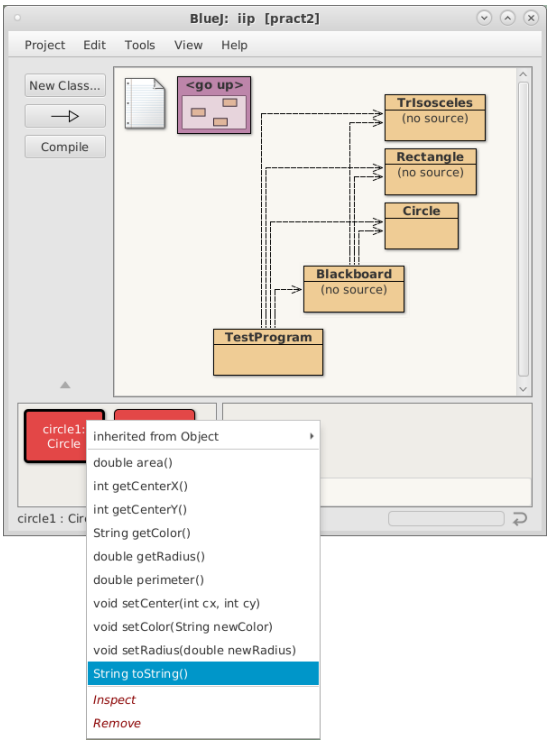
To debug the designed methods the `Inspect` option can be used. This operation allows to know the values of the attributes of the objects (Figure 10(c)).

### Activity #6

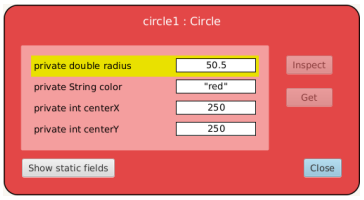
1. Create an object of the `Circle` class with radius 50.5, color red, and centre in $(250, 250)$.
2. Inspect the values of the created object.
3. Execute the `toString()` method defined in the `Circle` class on the created object.
4. Modify the radius of `Circle` to 30.0.
5. Execute again the method `toString()`.
6. Create an object of the class `Blackboard` with title "Drawing" and size $500 \times 500$. Do not close the created object.
7. Inspect the values of the attributes of the created object.
8. Add the `Circle` object to the `Blackboard`.




(a) Creating a `Circle` object.




(b) Object Bench and menu for `Circle` object.




(c) Inspecting the `Circle` object.

Figure 10: Creating an object in the Object Bench of *BlueJ*, menu, and inspection state of the object.

# 4 Using the Code Pad

The *BlueJ* Code Pad is situated on the right bottom corner, besides the Object Bench (Figure 11). In case it is not shown, it can be shown by selecting the option `View - Show Code Pad`.

This zone can be used to input Java expressions or instructions; when `Enter` is pressed, each line is evaluated and the result value will be shown, followed by its type (between parenthesis), or an error message when the expression is incorrect. Objects from the Object Bench can be used as well.

Some results of expressions are objects instead of primitive values. In this case, the object is shown as a reference to an object (`<object reference>`), followed by the type of the object, and a small icon which represents the object is shown by the result line. That icon can be used to work with the result object, since it can be dragged to the Object Bench.

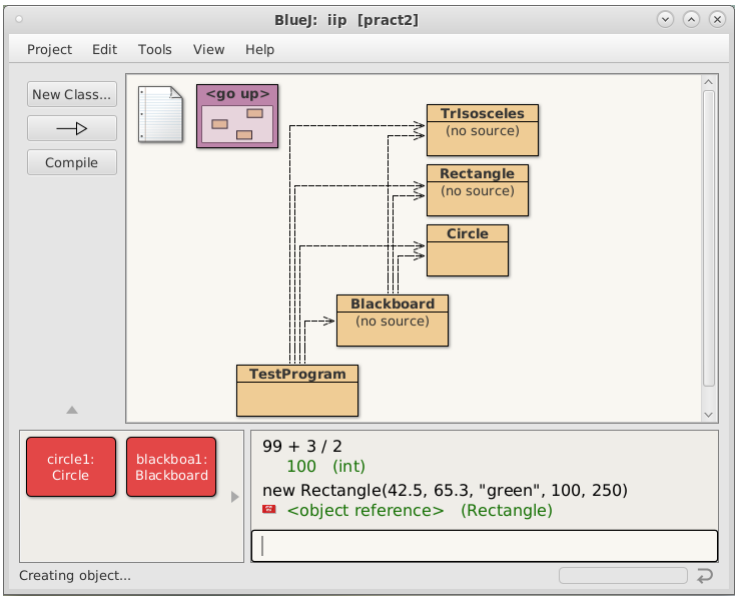



Figure 11: *BlueJ* Code Pad

In this case, the object will be situated on the Object Bench, where it will be available for future method calls (by using its pop-up menu or by using the Code Pad).

## Activity #7

1. What are the results of evaluating the following expressions in the *BlueJ* Code Pad?

| | | | |
|---|---|---|---|
| 1 | `8 % 3` | 6 | `9 / 2` |
| 2 | `(int) 98.67` | 7 | `9.0 / 2.0` |
| 3 | `Math.round(98.67)` | 8 | `9 / 2.0` |
| 4 | `Math.sqrt(121)` | 9 | `9 / (double) 2` |
| 5 | `Math.sqrt(-5)` | 10 | `9 / 0` |

2. Define the integer variables `x` and `y` with values 4 and 6, respectively. Write an arithmetic expression to calculate the following algebraical expression:

$$\frac{x^2 - y}{x} \tag{1}$$

3. Define the integer variables `a`, `b`, and `c`, with values 2, -7, and 3, respectively. Write an arithmetic expression to calculate the following algebraical expression:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

4. Write on the Code Pad an instruction that shows on the screen (terminal window of *BlueJ*) the radius of `circle1`, the `Circle` located in the Object Bench from the Activity #6.

5. Execute on the Code Pad the `toString()` method on `circle1` and drag the icon of the resulting `String` object to the Object Bench.
6. Create a new `Circle` object in the Code Pad and drag its icon to the Object Bench. Give to it the name `circle2`.
7. Write on the Code Pad an expression that returns the color of `circle2`.
8. Execute on the Code Pad the `toString()` method on `circle2`. The resulting `String` object can be dragged as well to the Object Bench.

# 5 Using the Debugger

The *BlueJ* debugger is a simple tool, but very useful when validating the functionality of a class (i.e., its `main` method) or any other method of the class. Basically, it allows to:

- watch the execution of any method, i.e., *obtaining its trace for a given set of values*, maybe step by step, or maybe only for some of the code lines;
- inspect the call sequence associated to the call of the method on execution;
- check pass of parameters and the values of the local variables of the method on execution.

To obtain this, the debugger has the following functionalities:

1. **Establish breakpoints**. The execution state of a method can be viewed only when its execution is stopped in a line of its code. The Debugger provides this functionality to allow to stop the execution of the method in a specific line, that is, *establishes breakpoints*.

   In *BlueJ*, breakpoints are established in the *breakpoint area* of the editor, on the left of the text. When clicking in this area by a code line selected for stopping the execution, a small stop sign appears, which marks a breakpoint in that line. When the execution of a method arrives to a line with this mark, the execution gets interrupted and subsequently appear:

   (a) *the editor window*, in which the next line to the breakpoint line appears highlighted, since is the *next line to be executed*;

   (b) *the debugger window*; with the information items and buttons that are presented below.
2. **Step-by-step execution**. When the execution is stopped, it can be restarted step-by-step (instruction by instruction), which allows to follow the code and watch how the execution progresses (i.e., allows to *make a trace* of the code).

   To perform a step-by-step execution in *BlueJ* it is enough with clicking repeatedly on the `Step` button of the Debugger window. Each click produces the execution of an only code line, and then the execution stops again.

   To finish this process and return to normal execution, the breakpoint must be erased by clicking on it, and then the button `Continue` on the Debugger window must be clicked.
3. **Inspection of variables and pass of parameters**.

   By watching the *BlueJ* Debugger window, we can observe the call sequence associated to the method on execution, check the pass of parameters, and inspect the values that have its local variables.

   To inspect any variable or parameter you must double click on it on the Debugger window.

## Activity #8

1. In the `main` method of the `TestProgram` class, create a breakpoint in the lines in which the objects of type `Circle`, `Rectangle`, and `TrIsosceles` are created.
2. Execute the `main` method and observe what happens when, after reaching the breakpoint, the `Step` button of the Debugger window is clicked.
3. To inspect the variables of the `main` method, double click on them in the Debugger window.

In the Figure 12 the *BlueJ* Debugger is shown after reaching the last breakpoint during the execution of the `TestProgram` class.

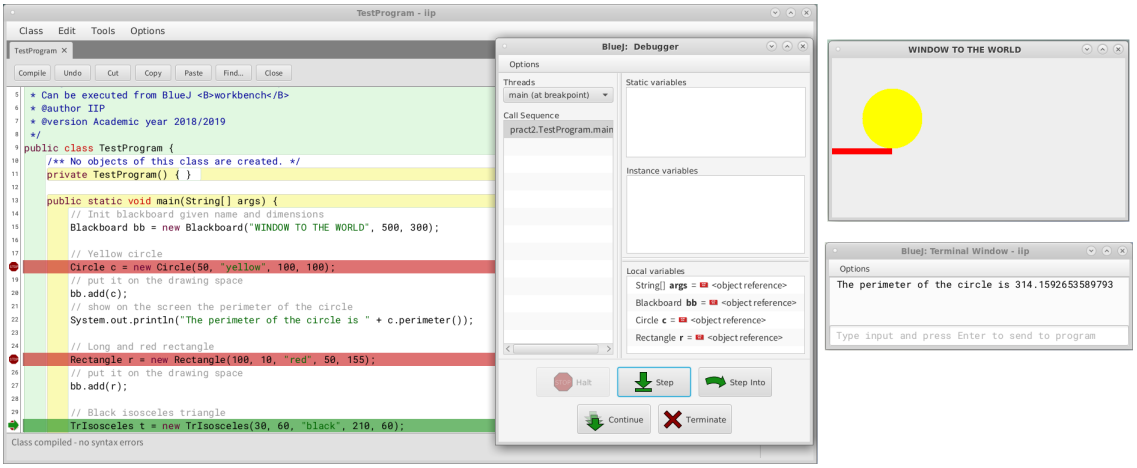



Figure 12: *BlueJ* Debugger.

## Activity #9

Write a class program, similar to the class `TestProgram`, that shows a figure formed by circles, rectangles and triangles.