FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department

Telecommunication Systems

2013/2014

Laboratory Work 0:
Demonstration of the Java environment
Learning application development

Class 1 – First Application

Integrated Master in Electrical Engineering and Computers

Luís Bernardo
Paulo da Fonseca Pinto

Index

1. OBJECTIVE

Familiarization with the Java programming language and application development in the NetBeans development environment. The work consists in the introduction of part of the code following the instructions set out, learning to use the environment and the set of library classes from the Java language. A project is provided with the start of the work, which is completed in a set of exercises.

2. NETBEANS ENVIRONMENT FOR JAVA

The system includes several Java packages for building GUI applications. The two most popular are the `java.awt.*` and `javax.swing.*`. You can program applications directly in a text editor, but it is convenient to use an integrated development environment for applications that integrates a GUI editor, text editor, compiler and debugger. It was chosen for this class the NetBeans environment due to the vast functionality available, compatibility with Linux, Windows and MacOS, and due to the price (http://www.oracle.com/technetwork/java/index.html).

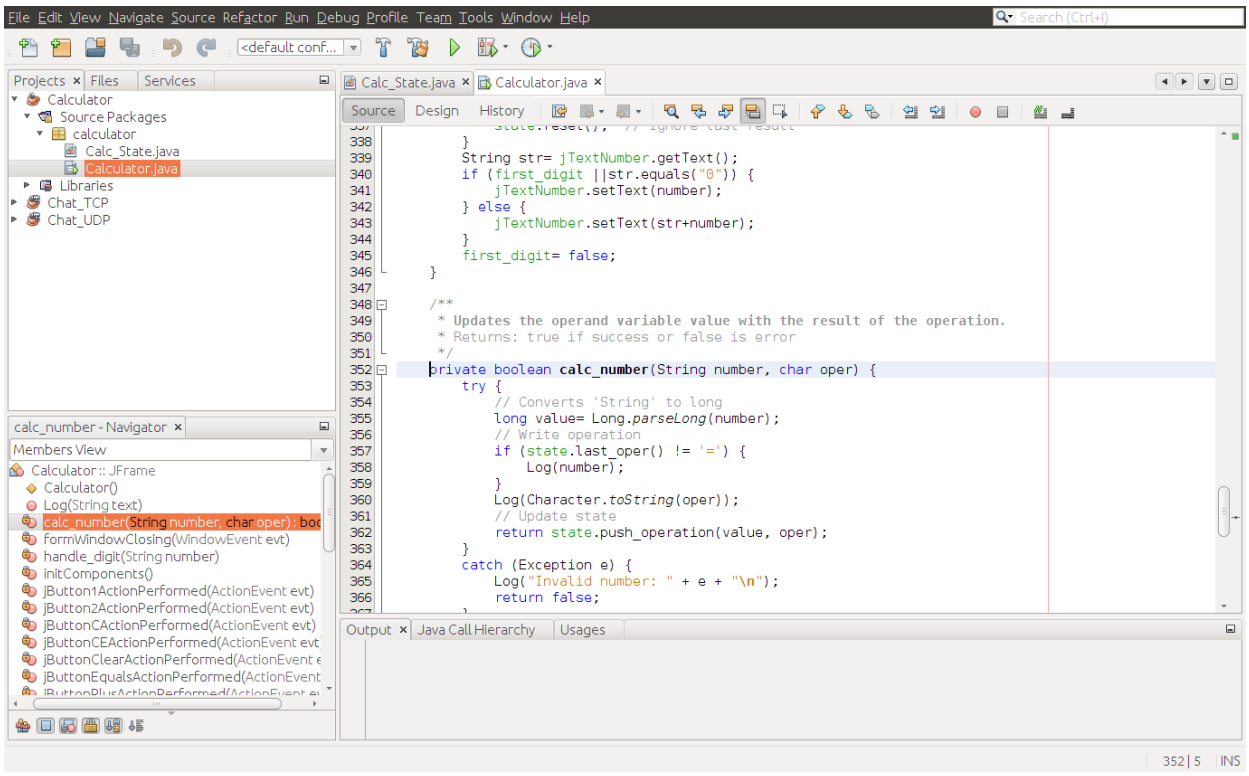
The NetBeans environment contains several windows that support a diverse set of options. There are four main modes of operation: "*Editing*" (text editing), "*GUI Editing*" (graphical editing interface), "*Running*" (running applications), and "*Debugging*" (running step by step, illustrated in the Figure below). Alongside there are several options achievable through menu options, or buttons. Pictured is represented the editing mode, which includes at left and up a list of projects, with the list of files for each project. In the projects that will be implemented in the discipline, there is one file associated with each class. In the "*Projects*" appears the list of packets (*Source Packages*) per project, and for each package comes the list of files: *Calc_State.java* and *Calculator.java*, associated respectively to the classes with the same name: `Calc_State` and `Calculator`.

When selecting a file, appears in the window below (left) the list of methods (functions), constants and variables declared in that class, allowing you to quickly navigate to the function code, represented in the editor. In the editor, you can easily access help information on an object. For example, if you introduce in the function represented, "`dp.`", a window is open with the list of supported methods in the `DatagramPacket` class and a description of each method.

To compile , or run with ( ) or without ( ) debug, simply use the keys indicated.

The figure below is the text editing window of NetBeans, with a hierarchical view of the code. Marked in blue appears that IDE generated code that can not be modified. The reserved words appear in blue letters, comments with gray letters, and the strings appear in red letters.

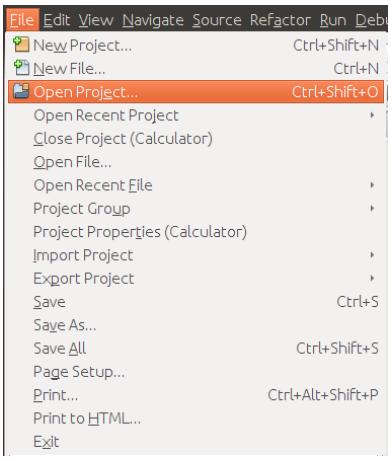
In this document the development of applications with a demonstration of the features of Java and NetBeans environment will be presented, step by step. It is recommended that students follow the description as a way to learn. At the end, for each application it is proposed a set of exercises to be performed by students.



3. FIRST APPLICATION - ADDER

This application will be implemented in two phases:

1. it is performed only minimal functionality of numbers input, sum and registration operations on a window and to a file.
2. It is performed a simplified calculater, adding the support for more operations.



The NetBeans project is provided to the students with the first phase implemented. To access the project, the students must begin by unpacking the file *Calculator.zip* to a local directory. Then you should select "`File`"; "`Open Project…`" (as shown at right), and open the `Calculator` project.

The project code is inside a package with the name `calculator`, and has two files corresponding to the two classes that compose it. The figure at the beginning of the page reproduces what you shall see after opening the project, if you select the file *Calculator.java* and the method `calc_number`.

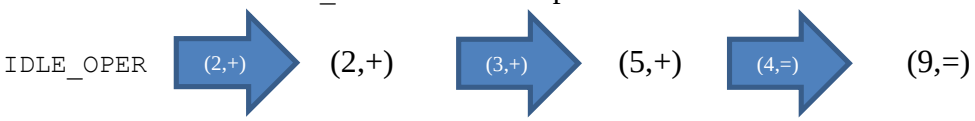
Students should analyze this code following the description that is made in the early part of this document. Then, you should use it to complete the second phase, as exercise.

3.1. SIMPLIFIED ADDER

The aim is to program a calculator that accepts inputs of numbers both by a set of buttons as through an array of characters (*string*) directly on the screen of the calculator, which supports a minimal set of operations ("+" and "=") to create a record of all transactions in a text window, and optionally, in a file.

The implementation will be based on two classes: the class `Calc_State` is used to keep the parcels during the calculation and the `Calculator` class defines the graphical user interface.

The approach used in the class `Calc_State` is to split expressions in a sequence of pairs (*value, operation*) and perform calculations sequentially, one pair at a time, using the method `push_operation`. The class has state the pending parcel (`operand`) and the pending operation (`oper`, of type `char`), which is initialized with `IDLE_OPER`, null operation. For example, the operation "2+3+4=" is accomplished with the composition of three pairs: (2, +) (3, +) (4, =). The evolution of the state in `Calc_State` for each operation will be:



The final result is stored in the value of `operand`.

3.1.1. Class `Calc_State`

The source code of class `Calc_State` is shown in the box below. It includes the definition of a constructor (run when invoking the operation `new Calc_State()`) that starts the state of the object, a method that restores the initial state (`reset()`) and the declaration of two private variables (not accessible directly). It also includes a constant (`Calc_State.IDLE_OPER`), which defines a state with no pending operation. Two methods are defined to access the variables' values: `last_oper()` and `operand()`. The main method is `push_operation(new_operand, new_operation)`, which performs calculations updating the state of the object, returning `true` on success, or `false` otherwise.

```
public class Calc_State {
    public static final char IDLE_OPER= ' ';    // Constant

    public Calc_State() {     /* Constructor */
        operand= 0;
        oper= IDLE_OPER;
    }

    public char last_oper() {  /* Returns the last operation */
        return oper;
    }


    public long operand() {    /* Returns the current operand */
        return operand;
    }

    public void reset() {     /* Resets the state to the default values */
        operand= 0;
        oper= IDLE_OPER;
    }

    /* Adds a new pair (operand, oper) to the state */
    public boolean push_operation(long new_operand, char new_oper) {
        switch (oper) {    // Calculate the pending operation
            case IDLE_OPER: // First operand
                operand= new_operand;
                break;
            case '+':   // Calculate pending addition
                operand += new_operand;
                break;
            case '=':
                break;
            default:    // Error - operation not supported
                reset();
                return false;
        }
        // Memorizes pending operation
        oper= new_oper;
        return true;
```

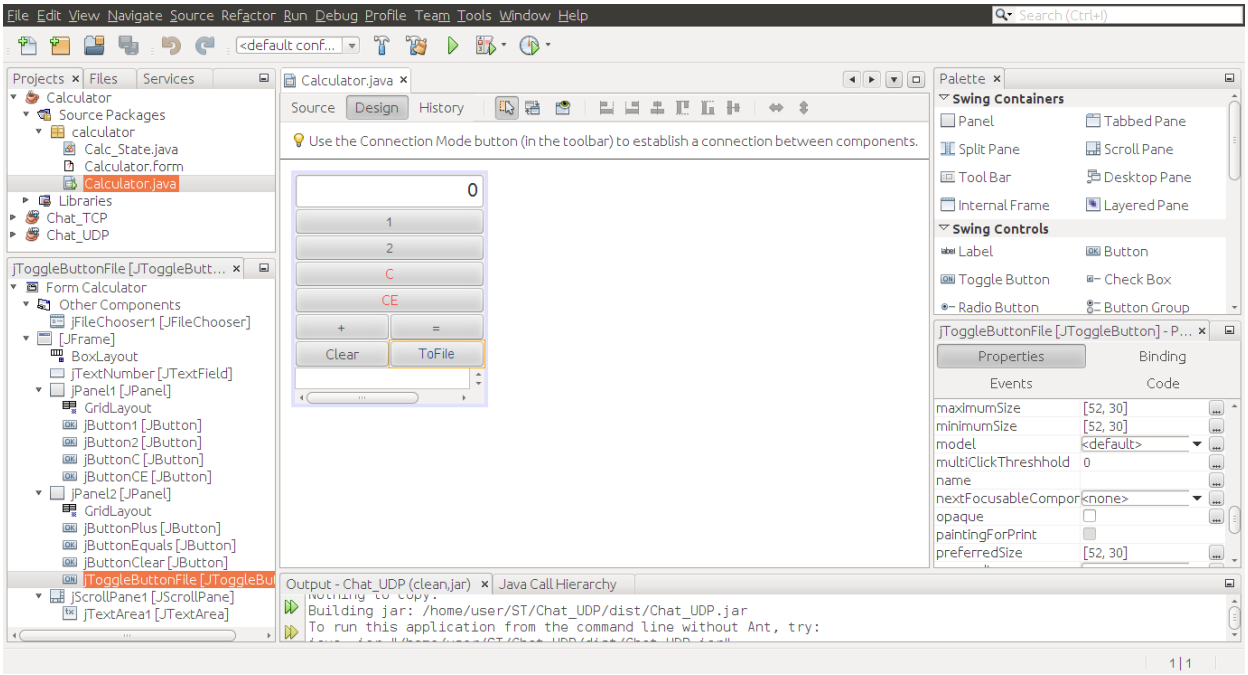
```
        }

        // Class local variables
        private long operand;
        private char oper;
    }
```

3.1.2. Class `Calculator` – Graphical interface

The design of the GUI is done using the "*Design*" mode, represented in the figure below. To develop this application several types of graphical components will be used (selected in the "*Palette*", at right in the picture) that are added to a `JFrame` window:

- 7 objects **Button** (from folder '*Swing Controls*') {for buttons 0-1, +, =, C, etc.}
- 1 object **ToggleButton** (folder '*Swing Controls*') {for the button with state 'ToFile'}
- 1 object **Text Field** (folder '*Swing Controls*') { Calculator display }
- 1 object **Text Area** (folder '*Swing Controls*') { Window with operation registry }
- 2 objects **Panel** (folder '*Swing Containers*') { Groups of buttons }
- 1 object **Scroll Pane** (folder '*Swing Containers*') { Slidding bars for *TextArea* }

  Besides, a component of type **File Chooser** (folder '*Swing Windows*') to choose the name of the file to store the calculation log.



Graphic objects were arranged according to the figure above, and the names of objects (e.g. the *Text Field* (type `JTextField`) was named `jTextNumber`) and the text of the buttons were edited, as shown in the "*MembersView*" in the bottom left of the picture above. In addition, the following properties were configured:

- object `JFrame`
  - ✓ `Layout` was set to `BoxLayout` with `Axes`= '*Y Axes*'; // Vertical organization of the panels
- objects `jPanel1` and `jPanel2`
  - ✓ `Layout` was set to `GridLayout`, with a grid size (`Columns`=3; `Rows`=4) and (`Columns`=3; `Rows`=2) respectively for panels 1 and 2.
  - ✓ `MaximumSize`= [210,116] and [210,60] respectively for panels 1 and 2, to limit the vertical growth of the panels with the buttons.
  - ✓ `PreferredSize`= [31,116] and [104,60] respectively for panels 1 and 2.

The fonts and colors of a few buttons were changed. You can see all the values of the properties of the various objects in NetBeans.

Finally, the event handler functions of all buttons were created, double-clicking on each button, which creates an empty function and associates it to the `actionPerformed` event. A function was associated to the event `windowClosing` of the `JFrame` object so as to intercept the termination of the window.

After creating the GUI, you should program the `Calculator` class, which performs the GUI. This class was automatically generated by NetBeans and extends the `javax.swing.JFrame` inheriting all its variables and methods. It also defines the `main` function, automatically generated to open the window when the application starts.

To facilitate interaction with the *Text Area* (`jTextArea1`) and enable the concurrent writing to the terminal and to a file, we defined a function, `public void` **`Log`**`(String str)` presented below. In addition, the "*Clear*" button clears the contents of the *Text Area*:

```
private void jButtonClearActionPerformed(java.awt.event.ActionEvent evt) {
  jTextArea1.setText("");
}
```

3.1.3. Class `Calculator` – Reading numbers

The number is stored in the text box `jTextNumber`, and a Boolean variable is also used to store whether it is the first digit.

```
private boolean first_digit;       // If it is the first digit of a number
```

The `handle_digit` method was created to concatenate the string digit number in the text box every time you press a number button:

```
private void handle_digit(String number) {
  if ((state.last_oper() == '=') && first_digit) {
    // If the result from the last addition is not used
    Log("");        // Add na empty line to the output
    state.reset();  // Ignore last result
  }
  String str= jTextNumber.getText();    // get the string
  if (first_digit ||str.equals("0")) {  // Ignore leading zeros
    jTextNumber.setText(number);        // Sets the string
  } else {
    jTextNumber.setText(str+number);    // Concatenate digit
  }
  first_digit= false;   // The next digit is not the first!
}
```

The functions generated in NetBeans graphical environment use the previous function to handle the numeric buttons (eg '1'):

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
  handle_digit("1");
}
```

The button '**C**' handler function removes the last digit of the number. The method `jTextNumber.getText()` returns a string with the contents of the box, whereas the method `jTextNumber.setText(str)` modifies the string shown to the value of `str`.

```
private void jButtonCActionPerformed(java.awt.event.ActionEvent evt) {
  String str= jTextNumber.getText();    // Get the string
  if (str.length() > 1) {               // Remove the last digit
    str= str.substring(0, str.length()-1);  // substring from 0 to length-1
    jTextNumber.setText(str);           // Set new string
  } else {      // Deleted all digits
    first_digit= true;
    jTextNumber.setText("0");
  }
}
```

3.1.4. Class `Calculator` – Calculation

The object `state`, of class `Calc_State`, will be used to perform the calculation,

```
private Calc_State state;          // Calculation state
```

initialized in the constructor of class `Calculator`:

```
state= new Calc_State();           // Allocates a new object Calc_State
```

The `calc_number` method was created to process the string in the text box, validating it, and to invoke the operation `push_operation` on the state. Meanwhile, it controls the writing of operations in the log. Note the use of the `try – catch`: if the number is valid, invokes the method and returns `true`; otherwise it runs the code in `catch`, returning `false`.

```
private boolean calc_number(String number, char oper) {
  try {
    long value= Long.parseLong(number);        // Converts 'String' to long
    if (state.last_oper() != '=') {
      Log(number); // Write number
    }
    Log(Character.toString(oper));  // Write operation
    return state.push_operation(value, oper);    // Update state
} catch (Exception e) {
    Log("Invalid number: " + e + "\n");
    return false;
  }
}
```

The method that handles the '+' button only has to invoke the previous operation and prepare the reading of the following number.

```
private void jButtonPlusActionPerformed(java.awt.event.ActionEvent evt)
  calc_number(jTextNumber.getText(), '+');
  first_digit= true;
}
```

The method that handles the button '=' is similar to the above, but additionally, it has to write the result of the operation.

```
private void jButtonEqualsActionPerformed(java.awt.event.ActionEvent evt)
  if (calc_number(jTextNumber.getText(), '=')) {
    Log(Long.toString(state.operand()));   // Writes the result of the operation
    jTextNumber.setText( Long.toString(state.operand()) );   // Update the number
  }
  first_digit= true;   // Wait for first digit of next number
}
```

3.1.5. Class `Calculator` – Writing to file

A escrita no ficheiro vai requerer duas variáveis adicionais: o descritor de ficheiro `f` e o objeto para escrita `os`. O ficheiro está ativo quando `os!=null`; caso contrário não se escreve no ficheiro.

Writing the file requires two additional variables: the file descriptor `f` and the writing object `os`. The file is active `os!=null`; otherwise no writing is done to the file.

```
private File f;                    // File object
private BufferedWriter os;         // File writting objects
```

The method `start_writing_to_file` opens a window allowing the user to choose the filename you want to save the log of operations, opening then the file writing object `os`. It returns `true` on success, or `false` if the file is not open.

```
private boolean start_writing_to_file() {
  try {
    // Open FileChooser dialog window na wait for filename selection
```

```
      if (jFileChooser1.showSaveDialog(this) == JFileChooser.APPROVE_OPTION) {
        f= jFileChooser1.getSelectedFile();   // Get the filename selected
        Log("Writing to: "+f+"\n");
        os= new BufferedWriter(new OutputStreamWriter(
                          new FileOutputStream(f), "8859_1"));  // Open writing device
        return true;   // Success - writing to file
      }
    } catch (Exception e) {  // Error during file selection or opening
      System.err.println("Error selecting output file: "+e);
    }
    return false;
  }
```

The method `stop_writing_to_file` does the opposite: it closes the file and writing objects and frees the memory, seting them to `null`.

```
  private void stop_writing_to_file() {
    try {
      if (os != null) {  // If the writing device is open
        os.close();  // Close file writing device
        os= null;    //     and free memory setting references to null
        f= null;
        Log("Stopped writing\n");
      }
    } catch (Exception e) {
      System.err.println("Error: "+e);
    }
  }
```

The control of writing to the file is done via the toggle button "*ToFile*." The following function handles the events of the toggle button, starting or finishing writing to the file.

```
  private void jToggleButtonFileActionPerformed(java.awt.event.ActionEvent evt) {
    if (jToggleButtonFile.isSelected()) {   // If it is selected, start writing
      if (!start_writing_to_file()) {
        jToggleButtonFile.setSelected(false); // If failed, set button off
      }
    } else {        // If it is not selected
      stop_writing_to_file();
    }
  }
```

The writing of the operations to file was held at `Log` function, which writes to the *TextArea*, to the terminal, and to the file, if it is open:

```
  public void Log(String text) {
    jTextArea1.append(text+"\n");      // Write to the TextArea
    System.out.println(text);          // Write to the terminal

    if (os != null) {                  // If file is open, write to file
      try {
        os.write(text + "\n");
      } catch (IOException ex) {       // Close file, if write failed
        jTextArea1.append("Error writing to file: "+ex+"\n");
        stop_writing_to_file();
      }
    }
  }
```

The writing of the contents in the file only occurs when the file is closed. To ensure that nothing is lost when you close the application, the following function has been added to handle the `windowClosing` event of `JFrame` object, triggered when the window closes.

```
  private void formWindowClosing(java.awt.event.WindowEvent evt) {
    // Main window is closing - close file if it is open
    stop_writing_to_file();
  }
```

3.1.6. Class `Calculator` – Starting and stopping the application

The application startup occurs from the `main` function, entirely created by *NetBeans* when creating the project. In this function, and within a task managed by the graphical environment, a new object of type `Calculator` is created and is made visible.

```
public static void main(String args[]) {
  java.awt.EventQueue.invokeLater(new Runnable() { // Create and display the form
            public void run() {
                new Calculator().setVisible(true);
            }
  });
}
```

The initial value of the variables of the `Calculator` class is defined in the constructor:

```
public Calculator() {
  initComponents();     // Start graphical window; created by NetBeans
  first_digit= true;        // Waiting for first number
  state= new Calc_State();  // Create object with operand and operation state
}
```

During operation of the calculator, it is possible to restore the initial state (except for the recording file) through the button CE:

```
private void jButtonCEActionPerformed(java.awt.event.ActionEvent evt)
  state.reset();
  Log("");                // Add an empty line
  first_digit= true;
  jTextNumber.setText("0");
}
```

3.2. COMPLETE CALCULATOR – EXERCISES FOR THE FIRST CLASS

The second phase of this project aims to complete the calculator previously started, completing the set of keys ('0' to '9') and adding a new operation ('*'). This second phase will be carried out by the students during the first class of the discipline.

3.2.1. Numerical buttons '0' to '9'

As a first exercise, students should modify the GUI, so as to have all the numerical buttons. You should open the file "*Calculator.java*" selecting the "*Design*" in the editor. Then the eight *Button* objects should be added to the form, in the first panel, changing its name and text, according to the figure on the right.

In a second phase, you should create the functions (methods) for the treatment of the new buttons, filling the corresponding code. To solve this problem, it is suggested reading section 3.1.3.

3.2.2. Multiplication operation

As a second exercise, the aim is to add the multiplication operation to the calculator. This exercise has three distinct phases:

- Add button '*'
- Program multiplication without considering precedence rules
- Program multiplication considering precedence rules

The first phase is similar to the previous exercise: add a button to the list of operations. The second and third phases require the definition of a method for handling the '*' button and reprogram the class `Calc_State` so as to make it recognize the multiplication operation. The

second phase should ignore the precedence of multiplication face to addition and calculate the operations sequentially, i.e. "2+2*2" should give the result 8.

The third phase will again focus on the class `Calc_State` but requires adding more variables to the class, to store up to two pending operations. In the case of “(2,+)(2,*)(2,=)” you need to save (2,+), (2,*) and the calculations should only be done after receiving (2,=), yielding the correct result (6).

3.2.3. Timing

If you arrive at this stage and still have at least 20 minutes, it is suggested a last exercise: if the user does not press any key for 10 seconds in the middle of a calculation, the calculator should display the partial result as if he had pressed the button '='. Do not forget to reset the clock every time the user presses a key.