IIP (E.T.S. de Ingeniería Informática)
Year 2017-2018

Lab activity 3

Basic elements of language and compiler

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Contents

1 Objectives and previous work

The main objective of this activity is to init the student on the development, compilation, and test of programs. Apart from that, concepts related to Unit 2 (primitive and reference variables) will be used. More specifically:

- Input of data by keyboard and output on the screen
- Use of integer arithmetic operators
- Datatype compatibility and casting
- Operator + (addition for integers, concatenation for `String`)
- Use of the classes `String`, `Scanner`, and `Math`, and use of their documentation

2 Problem description

It is usual to manage in many computer applications data items referring to time: a specific time instant, time between two time instants, ordering time instants, etc.

During this lab activity a program class must be implemented. This class will have to show on the screen data corresponding to a time instant (hour and minutes) in format `hh:mm`. The program will calculate the difference in minutes between two time instants: one inputed by keyboard and the current time, that will be automatically obtained. These two time instants will be presented in the previously described format.

The activities are proposed in order to cover the possible problems that could appear when developing this program.

3 Java code compilation and error detection

Obtaining the executable code (object code) from the high level code (source code) needs that this last one must be syntactically correct; otherwise, translation is not possible and compilation fails. The compiler, from the complete or partial syntax analysis, provides a list of the lines with errors and a message with indications on the error. Since the analysis is quite complex, these indications may not be precise enough to detect the errors at first glance, and sometimes only experience provides the knowledge necessary to interpret these messages.

In any case, compiler messages must be carefully read. Among others, frequent cases are the following:

- Unknown symbol or identifier: could be a typo on the name of the variable, class, or method (a letter is missing, or letter case is not correct); in the case of a variable, maybe was not declared; in the case of a class, maybe the package was not imported
- A syntactically incorrect expression could be caused by a misplaced or missing symbol (parenthesis, braces, brackets, operators, final semicolon, etc.); sometimes the error message must not be interpreted literally; for example, for the following line:

  ```
  int z = x y;
  ```

  what is missing is the `+` symbol to sum `x` and `y`, but the compiler will say that `;` is missing after `x`
- Misplaced braces (`{}`) may cause a bad definition of program blocks, giving errors related to missing or excessive closing braces; for example, in this code:

  ```
  import java.util.*;
  { public class Hello
        public static void main(String[] args) {
            ...
        }
  }
  ```

  the first opening brace is misplaced, and the compiler will say that was expecting a class declaration (not that brace)

As a conclusion, the programmer must check not only the line where the compiler signals the error, but a few previous and posterior lines, in order to put in context the error.

4 Execution errors and program test

A syntactically correct program, without compilation errors, could produce execution errors: logical errors (incorrect results) or runtime errors (produced by *exceptions*, that usually terminate the program execution).

These errors can be produced by several causes. In small programs they are usually produced by an inappropriate solving strategy, cases that where not taken into account, incorrect use of the operations (maybe because of a poor knowledge of their semantics), or even typo errors that do not correspond to syntax errors.

That is the reason why the code must be exhaustively checked with execution examples in order to check that the results are those expected.

5 Coding style

Apart from being syntactically and semantically correct, it is desirable for a code to be written clearly and with order, in order to make it legible and easy to maintain. Something similar happens with regular texts (e.g., newspapers, textbooks, etc.), where their organisation allows to have an easier reading and comprehension of the contents.

This is specially important for professional code, since these programs are usually maintained by different people during their lifetime. Thus, coding style are recommendations of the Java programmers community that, although not mandatory, allow for a better legibility of the programs.

The coding style affects both the code and the comments structure; it was developed by the previous Java developer (Sun Microsystems Inc.) and it is available in:

`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`

Some of them are:

- **Identifiers for classes, methods, and variables**: they must be representative of the meaning of the corresponding entity; methods and variables start with lowercase letter; in case they are composed of several words, second and next words must start with uppercase letter; for example, for a class `MyFirstClass`, for a method `sortNumbers`, for a variable `ageOfPerson`
- **Block structure and indentation**: opening braces for classes and methods (including `main`) must be in the same line that the class or method; closing brace must appear in a single line with the same indentation than the class or method; instructions and other sentences inside a block must have a higher indentation level than the external block; for example:

```
public class Hello {
    public static void main(String [] args) {
        System.out.println("Hello all!");
    }
}
```

- **Expressions**: assignment (`=`) and arithmetical, relational, and logical operators (`+`, `*`, `<`, `&&`, ...) must be preceded and followed by a blank space
- **Line length and line breaking**: very long lines must be avoided; when lines must be broken, they must be broken in a way that allows to recognise expressions and instructions structure; for example:

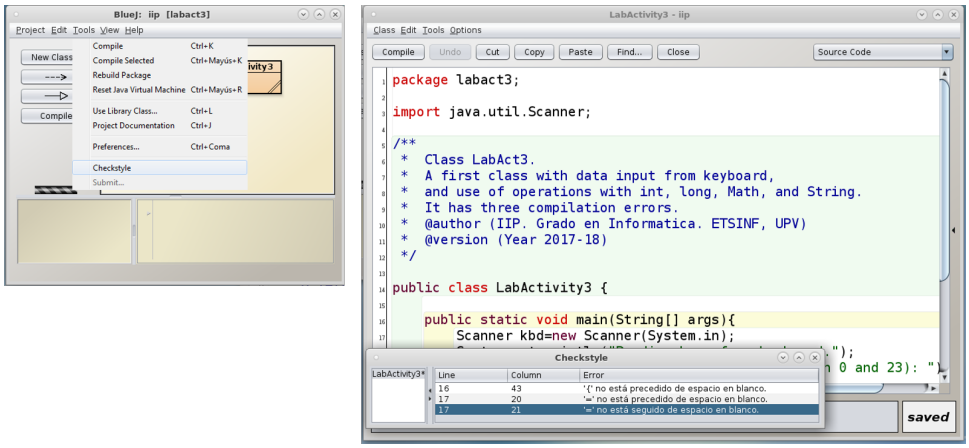

Figure 1: Checkstyle option and possible results.

```
System.out.println("The equation first real root is "
                   + (-b + Math.sqrt(b * b - 4.0 * a * c)) / (2.0 * a));
```

To promote the use of coding style, the extension *BlueJ checkstyle* was installed in the labs. Once compilation errors have been solved, you must select the `Checkstyle` option in the `Tools` menu, which will provide information on the coding style mistakes for the code you have written (see example in Figure 1).

6 Lab activities

Activity 1: creation of the BlueJ package `labact3`

Open *BlueJ* in the project `iip` in your home directory; create package `labact3`; add to the package the Java code `LabActivity3.java`, downloaded from PoliformaT (Recursos - Laboratorio - Práctica 3).

Activity 2: correction of syntax errors in the class `LabActivity3`

The source code of the `LabActivity3` class has three syntax errors (made on purpose) that prevent to compile it. According to compiler error messages, locate the errors and correct them. Figure 2 shows the output when the class is syntactically correct and compiles correctly. This code will be the starting point for solving the problem stated in Section 2.

Activity 3: check style

Employ the `Checksyle` option of the `Tools` menu and apply the suggestions on coding style that the checker provides.

Activity 4: showing the time instant in format `hh:mm`

The program must be modified to show the time instant in the format `hh:mm`, padding with zeros if needed, as can be seen in Figure 3.

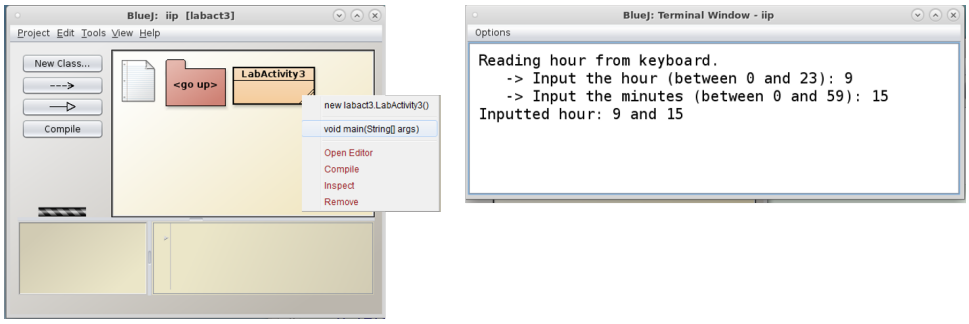

Figure 2: Output for initial `LabActivity3` class after correcting syntax errors.

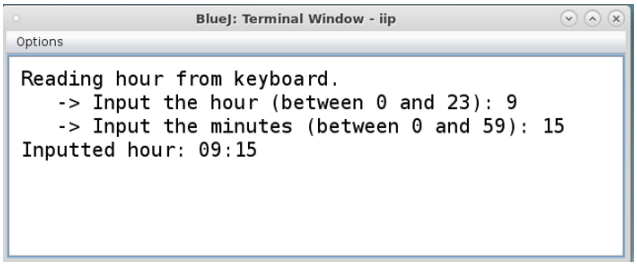

Figure 3: Output of the time instant in format `hh:mm`.

This can be solved by employing `String` class methods, creating from integer variables for hour and minutes a `String` which stores it in format `"hh:mm"`, and showing that `String` on the screen.

This can be done by concatenating `"0"` previously to the value of hour (and respectively for minutes), obtaining a `String` with two or three characters (e.g., for `h=9`, `"0" + h` will obtain `"09"`, but for `h=10` it will obtain `"010"`), and then obtaining the substring with the last two characters in order to obtain the two needed digits. The needed `String` methods are:

- `int length()`, that gives the number of characters n of the `String` objects, whose characters go from position 0 to $n-1$
- `String substring(int b)`, that returns the substring from position `b` to the end of the `String` (in this problem, from index $n-2$)

Thus, a `String hh` variable can be obtained with this process, and the same must be done for obtaining `String mm`, and then show on the screen the result for the concatenation `hh + ":" + mm`.

The program must be checked with several examples, like those shown in Table 1.

Activity 5: calculating current time (UTC)

The program must be completed to calculate current time (that at the moment of the program execution) hours and minutes. This can be done by calling the method:

```
System.currentTimeMillis()
```

Table 1: Test cases.

Hour	Minutes	"hh:mm"
0	0	"00:00"
9	5	"09:05"
9	35	"09:35"
19	5	"19:05"
19	35	"19:35"

As you can see in `System` class documentation, returns the number of milliseconds from 00:00 of Jan 1st 1970 in UTC (Greenwich time, in Spain usually UTC+1 in winter timezone and UTC+2 in summer timezone, except for Canary Islands).

The method returns a `long` value (`int` has not enough range for storing it). Since for calculations no units lower than a minute are used, the following Java assignment provides a `long` value with the number of minutes from the reference date, by using the quotient of the integer division of the milliseconds by 60×1000 (number of milliseconds of a minute). That value is stored in the `long` variable `tMinTotal`:

```
long tMinTotal = System.currentTimeMillis() / (60 * 1000);
```

Thus, `tMinTotal` stores the number of minutes in UTC from midnight (00:00) of January 1st 1970. From these minutes, only those pertaining to the current day are of our interest, that can be calculated as the remainder of the `tMinTotal` value by the number of minutes of a day (24×60). The number of minutes of the current day are smaller than the previous value and can be stored in an `int` variable

The next instruction shows how casting is applied to make the type conversion, since the final value can be stored into an `int` variable, and prevents the compilation error for mismatching types:

```
int tMinCurrent = (int) (tMinTotal % (24 * 60));
```

Figure 4 shows several examples on the casting from `long` to `int`, depending on the range of the converted value.

From the value `tMinCurrent` and by using simple arithmetic operations, the current hour and minutes can be obtained and the proposed problem can be solved.

Finally, the program must show as well the difference in minutes between the current time instant and that inputed from keyboard (consult `Math` class documentation for the methods that allow to obtain the absolute value of a number), expressed in hours and minutes. Execution result must be similar to that presented in Figure 5.

Extra activity: expansion of class `LabActivity3`

This activity is optional and can be developed in the lab if there is enough time. This activity allows to reinforce concepts on the management of `String` and on the syntax of `boolean` expressions.

1. There is an alternative for generating the format `"hh:mm"`, taking into account the following facts:

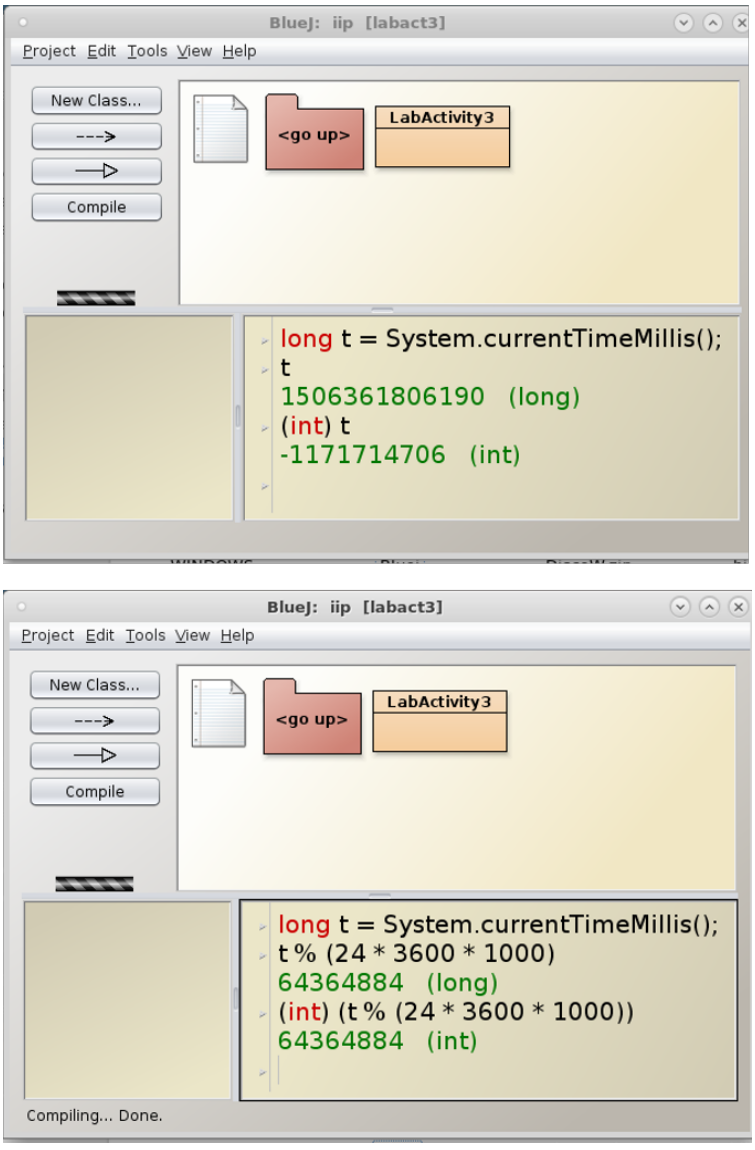

Figure 4: Casting examples.

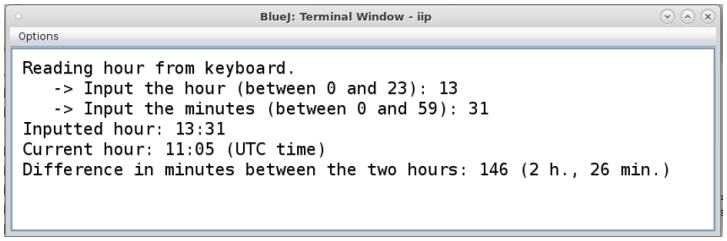

Figure 5: Final output of the program.

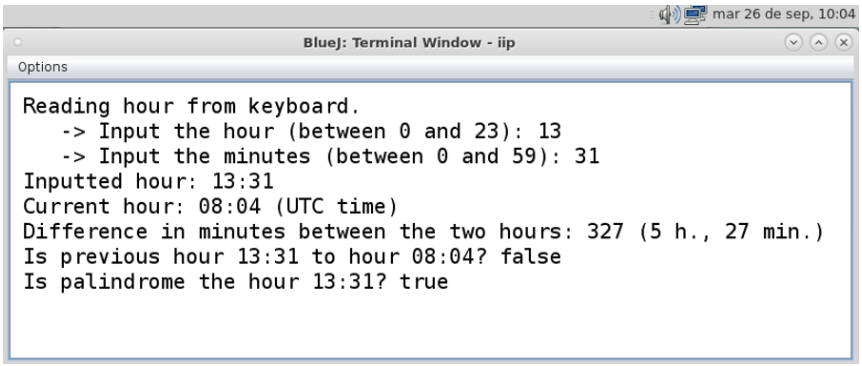

Figure 6: Result for the class extension.

Table 2: Test cases for class extension.

Hour	Minutes	`"hh:mm"`	Palindrome?
0	0	`"00:00"`	`true`
2	20	`"02:20"`	`true`
13	31	`"13:31"`	`true`
19	21	`"19:21"`	`false`
13	35	`"13:35"`	`false`
19	35	`"19:35"`	`false`

- Given the integer variable `h` with an hour, with values between 0 and 23, `h / 10` obtains the digit for the tens (0 when `h` < 10) and `h % 10` the digit for the units
- Both values are calculated as `int`, but when the expression `h / 10 + "" + h % 10` is evaluated, since `+` appears to operate with a `String` (`""`), they are converted into a `String`; since `""` is used, the result is a `String` where the values for the digits appear without separation (they are joined by the empty string)

A similar approach can be used for minutes, and the final time instant can be obtained by using this method.

2. Add to the `main` method the code in order to show the following results on the screen:
   - Boolean value that says if the inputed time instant is previous to the current time instant
   - Boolean value that says if the inputed time instant in its `String` format `"hh:mm"` is a palindrome (its reading is the same from left to right than from right to left); check `String` class documentation to choose the suitable methods; the program can be tested with the cases presented in Table 2.

   Resulting messages must be similar to those shown in Figure 6.