IIP (E.T.S. de Ingeniería Informática)
Year 2014-2015

Lab activity 3 - Second Part
Java Classes Development and Design
Data encapsulation and operations

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Contents

1 Objectives and previous work **1**

2 Problem description **1**

3 Lab activities **2**

1 Objectives and previous work

The main objective of this second part is to show the advantadges of encapsulating into an object a set of primitive data items that form a logical unit. Apart from that, this activity will show how encapsulating into a datatype class provides the opportunity of reusing code and developing applications. More specifically, these concepts of Units 3 and 5 will be developed:

- Implementation of a class (as a structure for objects)
- Implementation of constructors, consultors (`get`), modifiers (`set`), and other methods
- Use of the implemented datatype class: reference vars declaration, object creation, and object use (via methods)

2 Problem description

In this second part, the implementation of a class `Hour` is required. This class will encapsulate the data (hours and minutes) and operations described in the first part. In this way, when needed to make calculations on different hours and programs, calling the corresponding methods will be enough.

This will be reflected in the implementation of a new version of the `LabActivity3` class that employs objects and methods defined for the `Hour` class.

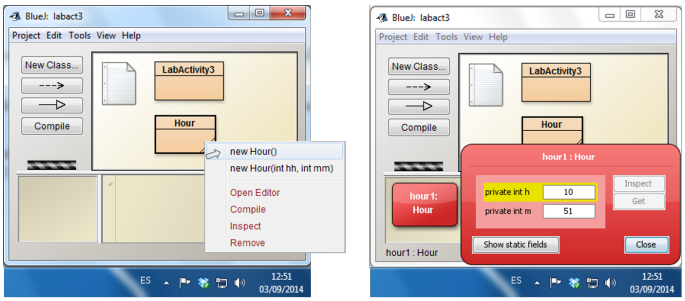

Figure 1: Correct example for object `Hour` creation.

3 Lab activities

Activity 1: development and test of the `Hour` class: attributes and constructors

Open in *BlueJ* the project `labact3` and add to the project the class `Hour.java` that is available in PoliformaT, which contains the skeleton for the class. Objects of this class must have as attributes those necessary to represent an hour, that is, hours and minutes. Thus, attributes must be:

```
private int h;
private int m;
```

The class must include a first constructor method with header:

```
/** Hour corresponding to hh hours and mm minutes.
 *  Precondition: 0<=hh<24, 0<=mm<60
 */
public Hour(int hh, int mm)
```

Apart from this, a default constructor with no parameters must be implemented, which will initialise the attributes to current UTC time. Thus, this method encapsulates the calculations that in the first part of the activity allowed to calculate current UTC hour:

```
/** Current UTC Hour (hours and minutes)
 */
public Hour()
```

The given class skeleton provides the comments for each of these methods, in a way that their documentation will be properly generated. Once this class is edited and correctly compiled, test object generation and correction, such like in the example of Figure 1.

Activity 2: development and test of the `Hour` class: consultors and modifiers

Add to the `Hour` class the constructors and modifiers whose headers are listed below:

```
/** Returns hour from current Hour object */
public int getH()

/** Returns minutes from current Hour object */
public int getM()
```

```
/** Modifies hour of current Hour object */
public void setH(int hh)

/** Modifies minutes of current Hour object */
public void setM(int mm)
```

Before adding more methods, recompile the class and check that all methods are correct. For that, you must create objects (in *BlueJ Object Bench* or *Code Pad*) and check the methods results.

Activity 3: development and test of the `Hour` class: methods `toMinutes`, `toString`, `equals`, and `compareTo`

Add to the class the methods that are described below:

```
/** Returns current Hour object in format "hh:mm"
 */
public String toString()

/** Returns true only if o and current Hour object represent the same hour
 */
public boolean equals(Object o)

/** Returns amount of minutes from 00:00 to current Hour object
 */
public int toMinutes()

/** Compares chronologically current Hour object and hour; result is:
 *      - negative when current Hour is previous to hour
 *      - zero if they are equal
 *      - positive when current Hour is posterior to hour
 */
public int compareTo(Hour hour)
```

When implementing `equals`, remember that the use of the shortcut AND (`&&`) makes important the order of the operands of the comparison between `o` and current `Hour` object:

```
o instanceof Hour  &&  this.h == ((Hour) o).h  &&  this.m == ((Hour) o).m
```

Using this form, second and third operands will be evaluated only when `o` is effectively an `Hour` object. In that case, casting can be applied on `o` and attributes can be properly accessed. The `instanceof` operand can be tested in the *Code Pad* by using tests such like those in Figure 2.

Recompile class and test the new methods. For example, for `equals` and `compareTo` you can create three objects `hour1`, `hour2`, and `hour3` for hours 00:00, 12:10, and 12:10, respectively, and check that:

- `hour2` and `hour3` are equal
- `hour1` is previous to `hour2` (negative result of `compareTo`)
- `hour2` is posterior to `hour1` (positive result of `compareTo`)

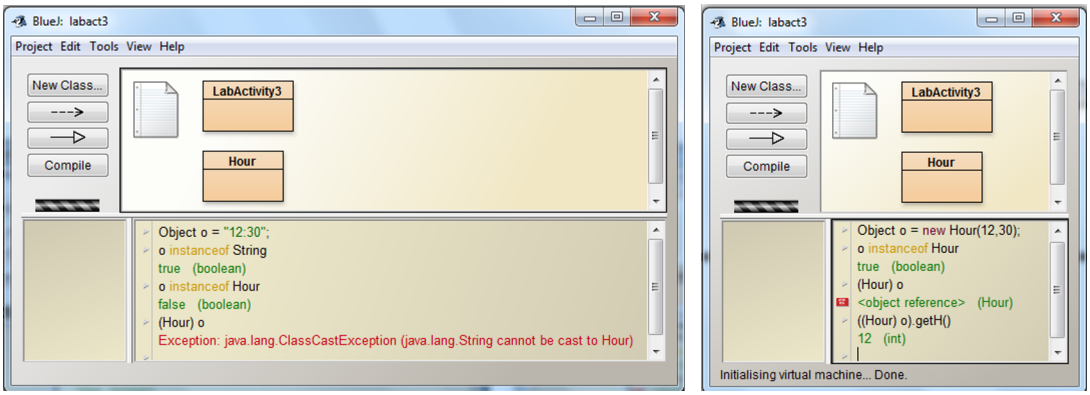

Figure 2: Behaviour for the `equals` method.

Activity 4: generating documentation for `Hour` class

Generate class documentation by passing from the edition (implementation) mode to the interface mode as shown in Figure 3.

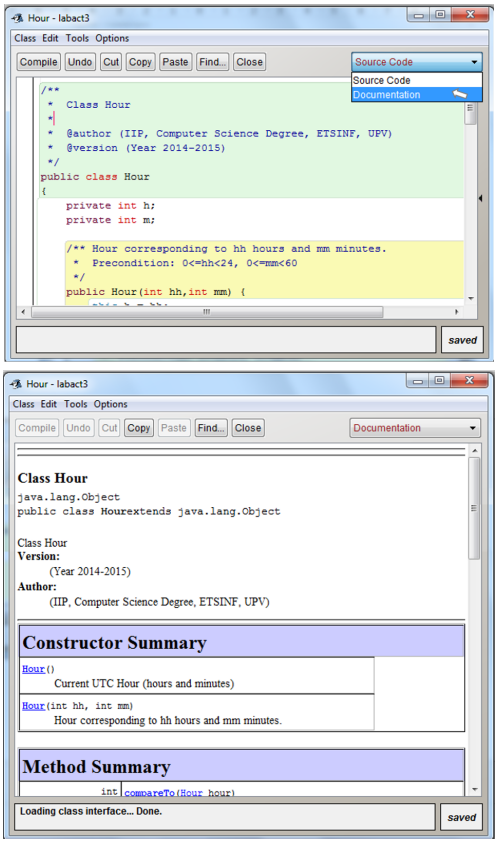

Figure 3: Documentation generation for class `Hour`.

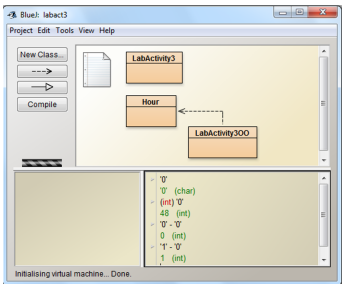

Figure 4: Transforming char digits into its numerical value.

Activity 5: implementation of class `LabActivity300`

Add to the project a new program class `LabActivity300` that solves the same problem than `LabActivity3` but by using `Hour` objects. That is, `LabActivity300` will be a transliteration of `LabActivity3` where:

- All hours (that inputted via keyboard or the current UTC time) must be stored into `Hour` objects
- The hour must be printed on the screen in the format `"hh:mm"` by using the `toString()` method of the `Hour` objects
- Difference in minutes between hours must be calculated similarly, but by employing the consultor methods or the `toMinutes` method of the `Hour` objects

Extra activity: expansion of class `Hour`: method `valueOf`

This activity is optional and can be developed in the lab is there is enough time. This activity allows to reinforce concepts on the `char` and `String` datatypes.

It is proposed to add to the `Hour` class the following method:

```
/** Returns an Hour from its textual description in format "hh:mm".
 */
public static Hour valueOf(String hhmm)
```

This method, given a `String` which represents an hour in format `"hh:mm"`, calculates and returns the corresponding `Hour` object. It is a `static` method (is not applied to any object) that only works with the given `String`.

The method must calculate the integer values that are stored into `hhmm` and then create the `Hour` object that corresponds to that hour. For calculating these values, you must take into account that:

- Characters in position 0 and 1 (`hhmm.charAt(0)` and `hhmm.charAt(1)`) correspond to tens and units of the hour, while those in positions 3 and 4 correspond to tens and units of the minute
- Although `char` and `int` are compatible (`char` are numbers in the range [0,65535]), codes for characters between `'0'` and `'9'` do not have the numerical values between 0 and 9; but since they are consecutive, the expression `d - '0'` gives the numerical value for `char d` if it stores a character that represents a digit (i.e., 0 for char `'0'`, 1 for `'1'`, etc.), as you can check on the examples seen in Figure 4