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

Lab activity 7. Parking management

Time: three sessions

Contents

1 Context and planning **1**

2 Problem statement **2**

3 `Ticket` class **3**

4 `Floor` class **5**

5 `Parking` class **7**

6 `ParkingManager` class **10**

7 Validation of the classes **10**

1 Context and planning

This lab activity summarises all the concepts of the subject, specially those corresponding to *"Unit 7. Arrays: definition and applications"*. To solve it you can consult, among other materials, Chapter 9 of the book "Empezar a programar usando Java" (3rd edition)[1].

This activity lasts for three sessions where activities proposed in this statement will be developed. In Table 1 you can see the planning of activities for each session, emphasising required homework to cover all the objectives in the planned time.

[1] Chapter 10 in 2nd edition.

	In classroom	Homework
Previous	–	Read this statement
Session 1	1, 2, and 3	Continue the implementation of `Ticket` and `Floor` classes
Session 2	4 and 5	Successfully finish the implementation of `Parking` class
Session 3	6, 7 and 8	Validate all implemented classes

Table 1: Planning of activities

2 Problem statement

It is desired to implement an application that simulates an intelligent parking in which, when a vehicle enters, the parking space to park is indicated, and when the vehicle leaves the parking, indicates which is the fare to be paid. Moreover, the application must offer the possibility of locating the vehicle (i.e., knowing on which parking space and floor is), and must be able to show the parking occupancy (i.e., show the different parking spaces indicating if they are occupied or free). Finally, the parking must have a mechanism to empty all spaces at the end of the day and calculate the corresponding fare for this action (i.e., the accumulated fare for all retired vehicles).

The proposed application must be organised in several datatype classes that would be instantiated to the different objects involved in the application. These classes are:

- `TimeInstant`: keeps data on the hour and minutes that define a timestamp; in this case, is used for representing the enter hour of a vehicle into the parking and its exit hour. This class was implemented in lab activity 4 in the package `labact4` and must be imported in all the other classes.
- `Ticket`: defines parking data of a vehicle that enters the parking: plate number, enter hour, and location (floor and space number).
- `Floor`: represents a floor in a parking (with their corresponding number and free spaces), and its occupancy by the set of tickets associated to the vehicles parked on that floor.
- `Parking`: represents a parking as the occupancy of its floors (with a fixed number of spaces by floor), along with the corresponding fee (in euros per minute).

Apart from that, it is necessary a program class `ParkingManager` that implements the access to the different functionalities of the application.

Activity 1: creation of the *BlueJ* package `labact7`

1. Open *BlueJ* and create in your `iip` project a new package `labact7`; close the project and *BlueJ*.

2. Download in the folder corresponding to package `labact7` the files available in PoliformaT (folder `Recursos/Laboratorio/Práctica 7/English/code`) of IIP.

3. Open again *BlueJ*, re-open your `iip` project and enter in package `labact7`, that must contain (as shown in Figure 1) the code files `Ticket.java`, `Floor.java`, `Parking.java`, and `ParkingManager.java`; apart from that, in the folder you will have two text files (`parkingIIP.txt` and `parkingIIP-Full.txt`), although they are not seen from *BlueJ*.

4. Access to directory `$HOME/asigDSIC/ETSINF/iip/pract7/docEnglish` and open the file `index.html` with any browser. You would be able to examine the documentation (in the usual Java format) of the classes that you must implement.

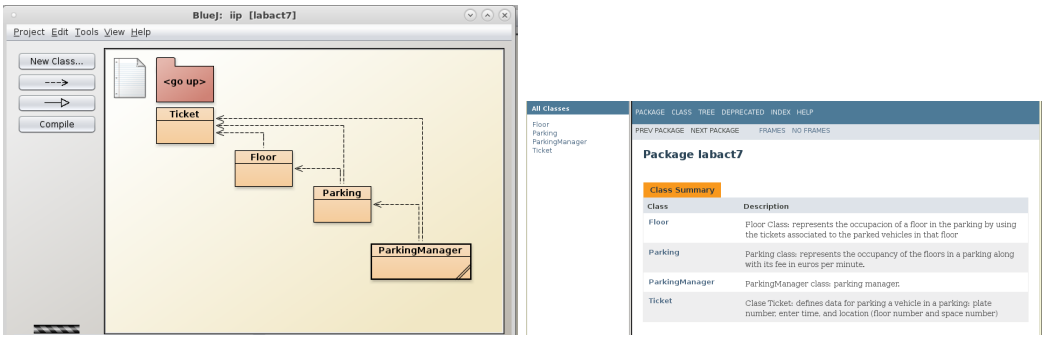


Figure 1: Package `labact7` in *BlueJ* and its documentation.

3 `Ticket` class

The `Ticket` class defines the parking data for a vehicle in a parking by the following private instance attributes:

- `plate`: `String` that indicates the plate number of the associated vehicle.
- `enterHour`: `TimeInstant` that indicates the time when the vehicle entered in the parking.
- `floor` and `space`: integer numbers that represent the location of the vehicle in the parking, that initially have value -1 and, after parking the associated vehicle, take the values of the floor and space number, respectively, where the vehicle is parked on.

This class has the following methods:

- Constructor `Ticket(String, TimeInstant)`: creates a `Ticket` object associated to a vehicle, with the plate number and enter time given as parameters, that access to the parking and, consequently, has no assigned location.
- A consultor method for each attribute: `getPlate()`, `getEnterHour()`, `getFloor()` and `getSpace()`.
- Modifier methods for floor and space: `setFloor(int)` and `setSpace(int)`; they will be used when parking the vehicle.
- A `toString()` method, that overrides that of `Object`, that returns a `String` with the data of the ticket in the following format:
  - When the ticket has not assigned location:
    `"Plate: PLATE - Enter time: ENTERHOUR"`
  - When it has it assigned:
    `"Plate: PLATE - Enter time: ENTERHOUR - Floor: FLOOR - Space number: SPACE"`

Activity 2: implementation and test of `Ticket` class

- Review in the `Ticket` class the definition of the attributes and methods that are given.
- Complete the constructor method following its specification.
- Complete the `toString()` method following its specification.
- By using the Code Pad or the Object Bench of *BlueJ*, create a `Ticket` object of a vehicle with plate number 1234AC that enters the parking at 10:25. Check that the result is that shown in Figure 2.

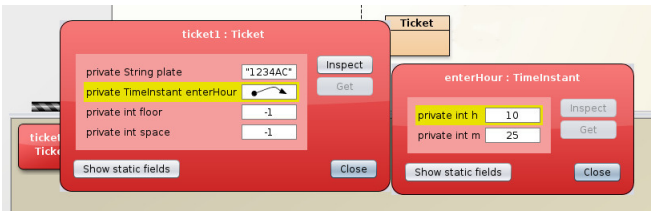


Figure 2: `Ticket` object for a vehicle with plate 1234AC that enters the parking at 10:25.

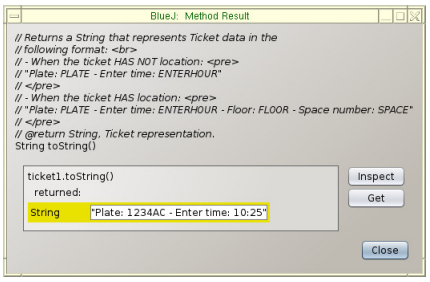


Figure 3: Result of the `toString()` method for a `Ticket` of a vehicle with plate 1234AC, enter time 10:25, without assigned location.

- Execute each of the consultors and check the correct result.
- Execute the `toString()` method and check that the result is that shown in Figure 3.
- Execute methods `setFloor(int)` and `setSpace(int)` to update the floor and parking space of the created `Ticket` with values 2 and 5 respectively. Check that the result is that shown in Figure 4.

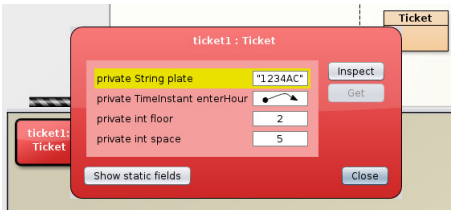


Figure 4: `Ticket` object of a vehicle situated in floor 2 and parking space 5 of the parking.

- Execute the `toString()` method and check that the result is that shown in Figure 5.

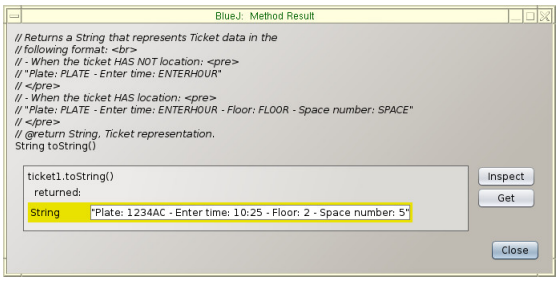


Figure 5: Result of the `toString()` method for a `Ticket` of a vehicle with plate 1234AC, enter time 10:25, situated on floor 2 and space 5 of the parking.

4 `Floor` class

The `Floor` class allows to represent a floor in the parking with an associated number, and that has a set of spaces that can be free or occupied. It is defined by the following private instance attributes:

- `numFloor`: integer that indicates floor number.
- `spaces`: array of `Ticket` objects of a given size (given in the constructor when the object is created) that represents the occupancy of the parking spaces on the floor. The number of each space corresponds with its position on this array, in a way that `spaces[i]` keeps the `Ticket` associated to the vehicle parked on space `i`, or it is `null` when the space is free (with $0 \leq$ `i` $<$ `spaces.length`).
- `freeSpaces`: integer that indicates the number of free parking spaces on the floor (initially, the whole number of spaces).

Methods for that class are the following:

- `Floor(int, int)` constructor: creates an object `Floor` given a number for the floor and number of spaces, with all parking spaces free (see Figure 6).

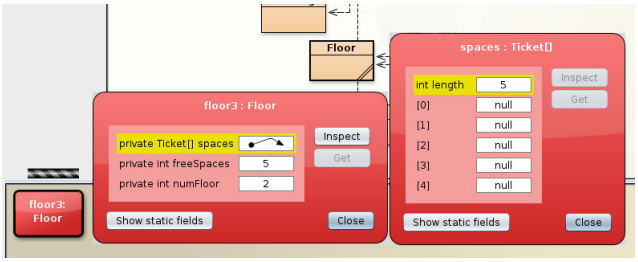


Figure 6: `Floor` object for floor 2 in a parking, with 5 parking spaces (all free).

- A consultor method for the number of floor, and a consultor method of the number of free spaces: `getFloor()` and `getFreeSpaces()`.
- A method `isFull()` that returns `true` when floor has no free spaces, and `false` otherwise.
- A method `firstFree()` that returns the number of the first free space (i.e., that with lowest number) on the floor, or `-1` when no free spaces available. This method must employ the `isFull()` method.
- A method `park(Ticket)` that, given a `Ticket` without assigned location, simulates the parking process on that floor of the associated vehicle (see Figure 7) as follows: if there are free spaces (textttfirstFree() returns different from -1), assigns the `Ticket` parameter to the first free space (that of lowest number), and updates the number of space in the `Ticket`[2] and the number of free spaces on the floor. This method must employ the `firstFree()` method.
- A method `searchTicket(String)` that checks if, given a plate number, the corresponding vehicle is on the floor, returning the associated `Ticket` if found, or `null` otherwise.
- A method `leave(int, TimeInstant)` that simulates a vehicle leaving from the floor (see Figure 8), given the number of space `spc`, occupied (`spaces[spc] != null`), and a `TimeInstant` that represents the exit time (posterior to that vehicle's enter hour); this method must return the number of minutes that passed from the vehicle entrance until the given exit time. When the vehicle leaves, the `spaces` array and the number of free spaces must be updated.

[2]Notice that number of floor is not updated here, but in the `Parking.park` method.

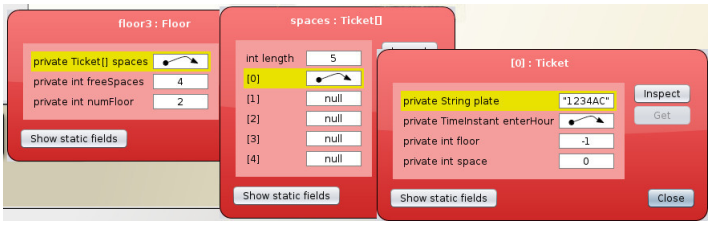
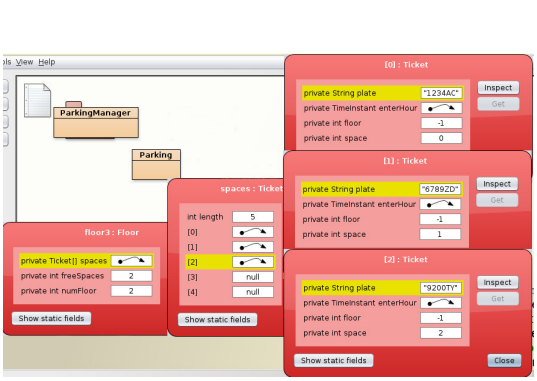
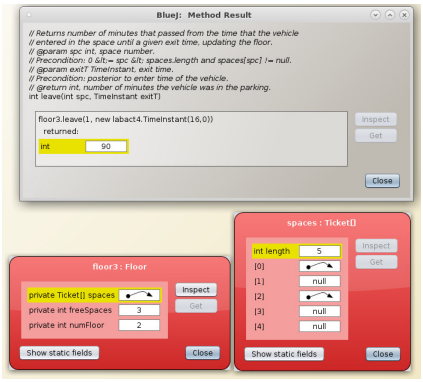


Figure 7: Result for the method `park(Ticket)` when parking a vehicle on floor 2 of a parking with 5 free spaces.



(a) `Floor` object of the floor 2 of a parking, with 3 occupied spaces (of the total 5).



(b) Result of leaving, at 16:00, the vehicle on space 1 that entered at 14:30.

Figure 8: Result for the method `leave(int, TimeInstant)`.

- A method `emptyFloor(TimeInstant)` that, given a `TimeInstant` that represents an empty time (posterior to the enter time of all the vehicles on the floor), takes out all the vehicles and returns the total number of accumulated minutes for all the vehicles that were on the floor until the given exit time. This method must use `leave(int, TimeInstant)` method.
- A method `toString()` (that overrides that of `Object`) that returns a `String` with the occupancy of a floor. For example, if the floor is number 2 and, from the 5 spaces that it has, numbers 0, 1, 3, and 4 are occupied, the resulting `String` would be: `" 2 X X X X "`, as can be seen in Figure 9. The same `String`, but substituting spaces by `-` for a clearer view of the format, is `"--2---X---X-------X---X-"`. This method is already implemented.

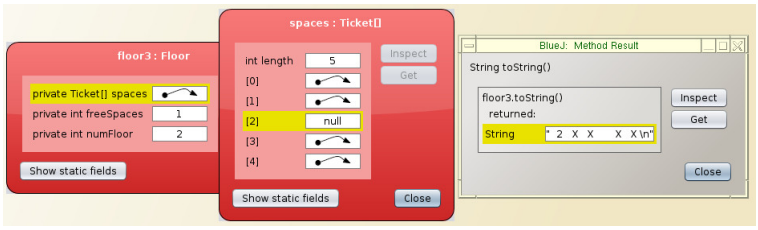


Figure 9: Result of method `toString()`.

Activity 3: implementation and test of the `Floor` class

Implement the attributes and methods for the `Floor` class. When finished, you can test it by using the Code Pad or Object Bench of *BlueJ*. More specifically, check the behaviour of the

constructor and the methods `park(Ticket)`, `searchTicket(String)`, `leave(int, TimeInstant)`, and `emptyFloor(TimeInstant)`, as shown in Figures 6, 7, and 8.

5 `Parking` class

The `Parking` class represents a parking as a set of floors with free and occupied spaces, along with the corresponding fee (euros per minute). It is defined according the following attributes:

- `FLOOR_SPACES`: public constant class attribute, with value 5, that indicates the number of spaces per floor.
- `floors`: private instance attribute, array of `Floor` objects with a given size (defined when the object is created in the constructor), that represents the floors in the parking. The number of each floor corresponds with its position in this array, in a way such that in `floors[i]`, the `Floor` of number `i` is stored, with $0 \leq$ `i` $<$ `floors.length`.
- `fee`: private instance attribute, real positive, that represents the parking fee (in euros per minute).

The behaviour of the objects in this class is given by the following methods:

- `Parking(int, double)` constructor: creates a `Parking` object given number of floors and fee, that initially is an empty parking (see Figure 10).

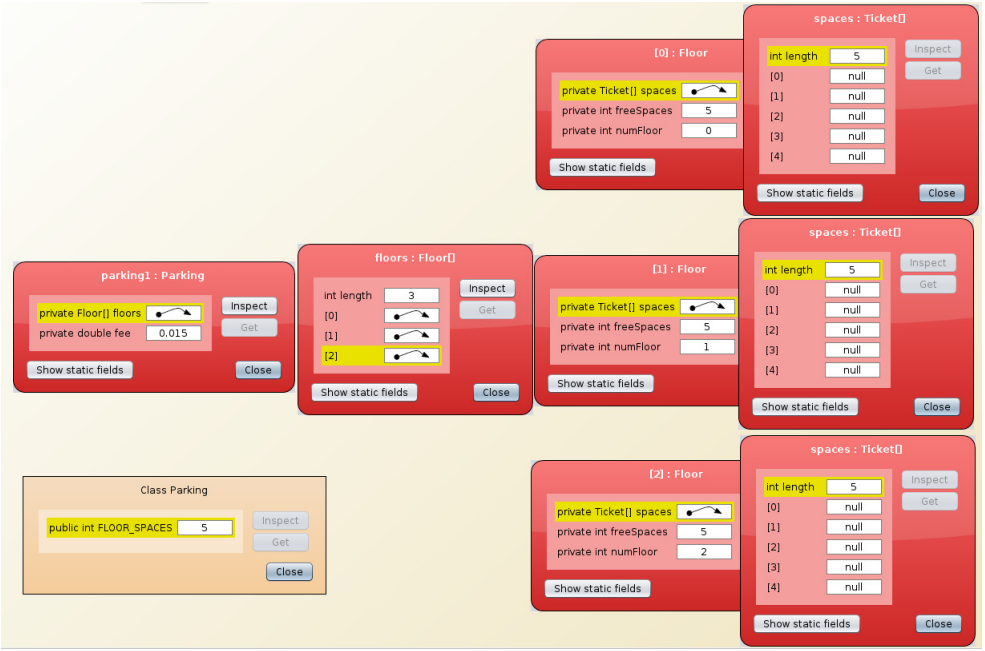
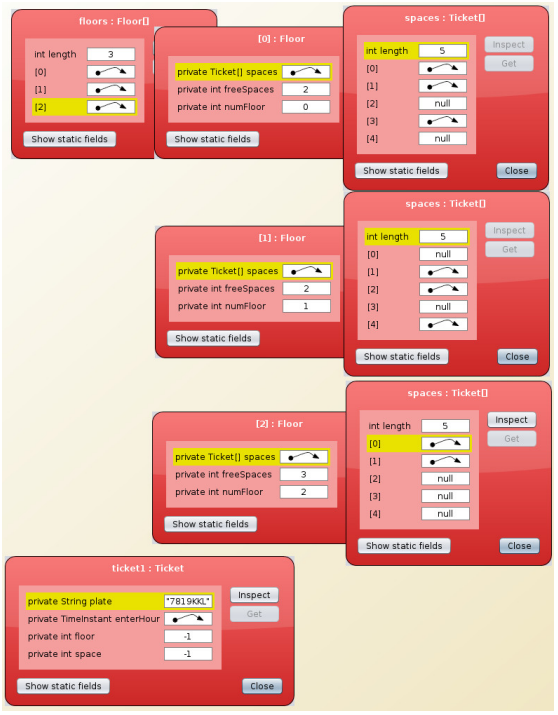


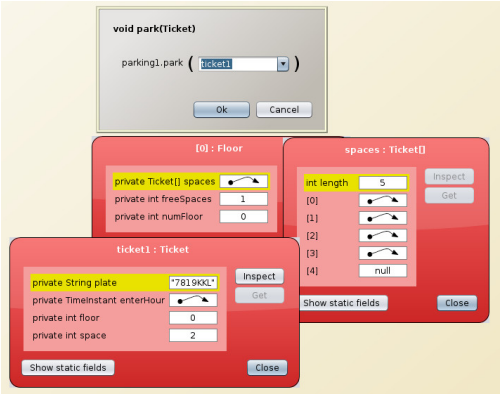
Figure 10: `Parking` object that represents a parking with 3 floors, 5 spaces per floor (all them free), and a fee of 0.015€/minute.

- `Parking(String)` constructor: alternative constructor that reads from a text file, in a specific format, data occupancy for the parking in a given moment, initialising with those data items the instance attributes of the created `Parking` object. If you open with any text file editor the file `parkingIIP.txt` (or `parkingIIP-Full.txt`), you will see that first and second line keep, respectively, the number of floors and the fee of the parking, and the following lines the data for the parked vehicles (number of floor, plate, enter hour and minutes). This method is fully implemented since file management is taught in PRG (second semester). It is used here for making test without having to input manually data in each execution.

- A consultor method for the number of floors and a consultor method for the fee: `getNumFloor()` and `getFee()`.
- A modifier method to update the fee: `setFee(double)`.
- A method `isFull()` that returns `true` when the parking is full; otherwise, returns `false`. The parking is full when all the floors all full; thus, this method uses the method `isFull()` of the `Floor` class.
- A method `park(Ticket)` that, supposing that the parking has free spaces and given a `Ticket` without previous location, simulates the parking process for the associated vehicle in the first space (that of lowest number) on the first floor (that of lowest number) with free spaces (see Figure 11). The process is: once the floor with lowest number with free spaces is found, update with that number the number of floor of the `Ticket` and then call the `park(Ticket)` method for that floor in order to make the parking.



(a) `floors` array of a `Parking` and `Ticket` of the vehicle to be parked.



(b) Result of parking in the first free space (space 2 of floor 0).

Figure 11: Result of the method `park(Ticket)`.

- A method `park(Ticket, int)` that, supposing that there are free spaces in the parking and given a `Ticket` without assigned location and a preferred floor, simulates the parking process of the vehicle associated in that floor if there are free spaces in that floor; otherwise, it must park in the closest floor with free spaces, as Figure 12 shows, in order to guarantee a space in a floor as closest as possible to the preferred floor. Once the floor is found, the number of floor of the `Ticket` object must be updated and the method `park(Ticket)` for that floor must be called in order to make the parking. In more details, the search strategy to be implemented described in Figure 12, applied when the preferred floor `pref` has no free spaces, consists of searching first the previous floor `pref - 1` and, if no free spaces are present, search in the posterior floor `pref + 1`. In case that floor has no free spaces, the search is repeated for floors `pref - 2` and `pref + 2`, and so on, until finding a floor with free spaces.

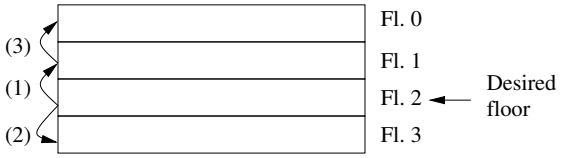


Figure 12: Search strategy to be implemented in method `park(Ticket, int)` in order to obtain a floor with free spaces.

- A method `searchTicket(String)` that checks if, given a vehicle plate, is in the parking (see Figure 13), returning the `Ticket` associated to the vehicle if found, or `null` otherwise. This method must employ the method `searchTicket(String)` of `Floor`.

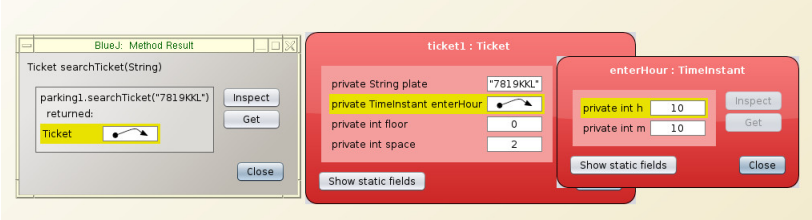


Figure 13: Result of the method `searchTicket(String)` in the `Parking` object of Figure 11(b) in order to locate a vehicle with plate 7819KKL.

- A method `leave(Ticket, TimeInstant)` that simulates the exit of a vehicle from the parking, given a valid `Ticket` (with assigned location and with its vehicle in the parking), and a `TimeInstant` that represents the exit time (posterior to enter time of the vehicle), and returns the fare to be paid (see Figure 14). Since taking out a vehicle from the parking is taking it out from the floor where it is, this method must employ the method `leave(int, TimeInstant)` of `Floor`.

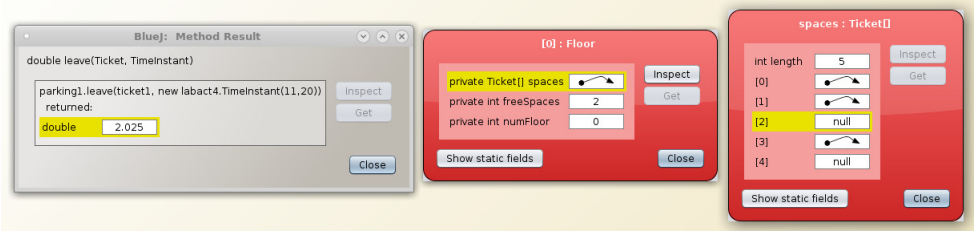


Figure 14: Result of the method `leave(Ticket, TimeInstant)` in the `Parking` object of Figure 11(b) when taking out, at 11:20, the vehicle with plate 7819KKL.

- A method `emptyParking()` that makes the parking empty, taking out all the vehicles, supposing that time is 23:59, and returning the total fare in euros to be paid for all the retired vehicles. Since emptying the parking is emptying all its floors, this method must employ the method `emptyFloor(TimeInstant)` of `Floor`.
- A `toString()` method, that overrides that of `Object`, that returns a `String` representing the occupancy of the parking with the following format: a first line that identifies the spaces corresponding to each floor, followed by as many lines as floors that represent the occupancy of each floor (result of the `toString()` method for `Floor`). For example, for a parking with 3 floors and 5 spaces per floor, where occupancy is: spaces 0, 1, and 3 of floor 0; spaces 1, 2, and 4 for floor 1; spaces 0 and 1 for floor 2 (situation of the `Parking` object in Figure 11(a)),

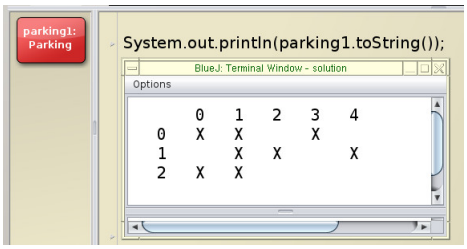


Figure 15: Result of the `toString()` method for the `Parking` object in Figure 11(a).

the result `String` is the one shown in the terminal window of *BlueJ* in Figure 15. Below is shown that `String` but changing blank spaces by `-` in order to clarify the used format:

```
------0---1---2---3---4-
--0---X---X-------X-----
--1-------X---X-------X-
--2---X---X-------------
```

Activity 4: `Parking` class implementation and test

Implement the attributes and methods for the `Parking` class. When finished, you can test it by using the Code Pad or Object Bench of *BlueJ*. In particular, check the behaviour of the constructor and the methods `park(Ticket)`, `park(Ticket, int)` (pay special attention to the case where preferred floor has no free spaces), `searchTicket(String)`, `leave(Ticket, TimeInstant)`, `emptyParking()`, and `toString()`.

You can create your own file in order to define a situation of the parking or employ the text files given (`parkingIIP.txt` and `parkingIIP-Full.txt`) in order to create a parking with a specific occupancy (by using the `Parking(String)` constructor) in order to make fast tests. For example, the situation in Figure 11(a) is the result of creating a `Parking` object from the data in file `parkingIIP-Full.txt` and executing the methods `searchTicket(String)` and `leave(Ticket, TimeInstant)` for the vehicles with the following plates: 7819KKL, 8199BBD, 1123DXC, 7264BPR, 67890BD, 54321EF, and 98796GH.

6 `ParkingManager` class

The `ParkingManager` class is a menu-driven application that offers six different options: park a vehicle, make a vehicle leave, search for a vehicle, show occupancy, make empty the parking, and finishing the application.

Activity 5: test of the `ParkingManager` class

Execute the `main` method of `ParkingManager` as is given and check the behaviour of each option. Modify the code in order to employ other initial configurations (empty parking or full parking by using the `parkingIIP-Full.txt` in the parking creation with `Parking(String)` constructor). In Figure 16, an example of execution of `ParkingManager` with the data in `parkingIIP.txt` is shown.

7 Validation of the classes

When you teacher consider it as convenient, you will have available in PoliformaT unit test for the different classes that you have implemented, in order to validate their implementation. In that case, you will have to add the test units to the `labact7` package and it must appear something similar to what you have in Figure 17.

In general, to make tests correctly execute, you must:

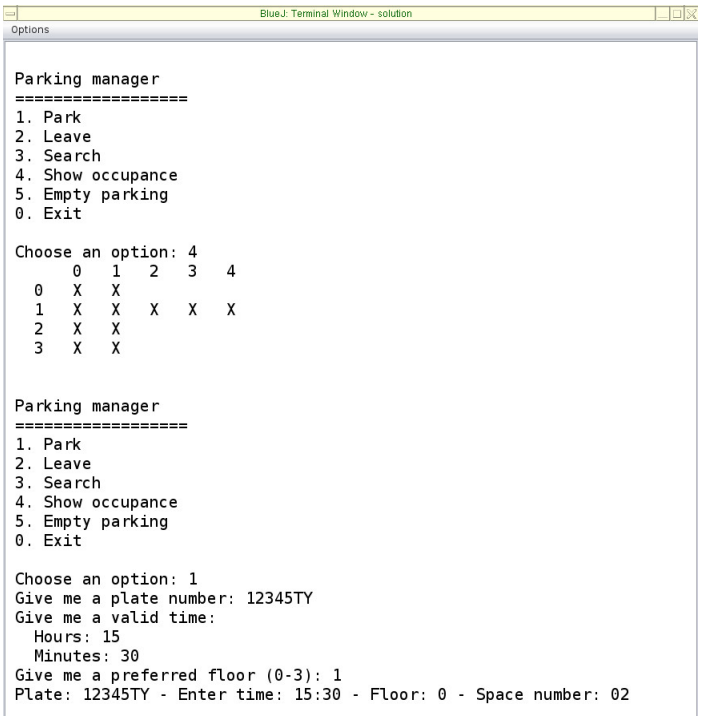


Figure 16: Execution example for `ParkingManager`.

- Employ the names for the attributes and methods proposed in the statement and the documentation of the `.java` files provided to you, following the features and restrictions about modifiers and parameters proposed in the statement for each of them.
- In the methods that return a `String` result, follow the format and text indicated in the statement.

Activity 6: validation of the `Ticket` class

1. To employ the test for this activity, you must have validated the `TimeInstant` class (you have available the validation test `Lab4TestUnit` from lab activity 4), since it is used in the `Ticket` class. Any error in the `TimeInstant` class probably would cause errors in the test for `Ticket` class and in any other that employs `TimeInstant`.
2. Choose the option *Test All* from the pop-up menu activated by right-clicking the test icon. A set of tests would be executed on the methods of the tested class, comparing obtained and expected results.
3. As happened in previous lab activities, when the methods are correct, in *Test Results* window of *BlueJ* they will be marked with ✓ (green color). If any of the method does not work correctly, in *Test Results* windows the method will be marked with `X`. If any of the tests is selected, the lower part of the window displays an indicative message about the cause of the error.
4. If after correcting errors and recompiling the class the test unit icon appears stripped, you must close and re-open the *BlueJ* project.

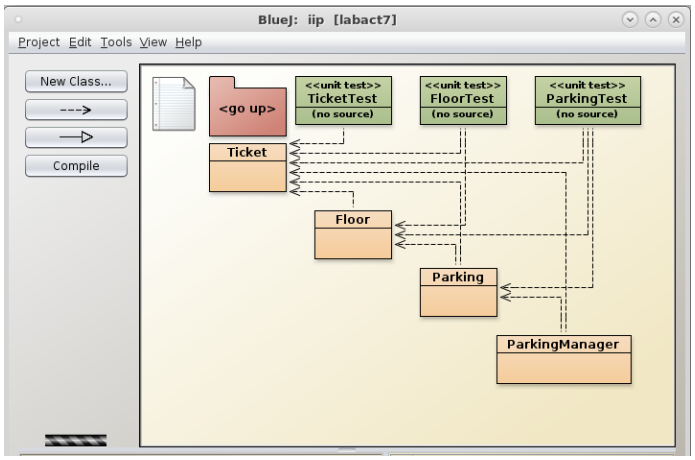


Figure 17: `labact7` package with test units.

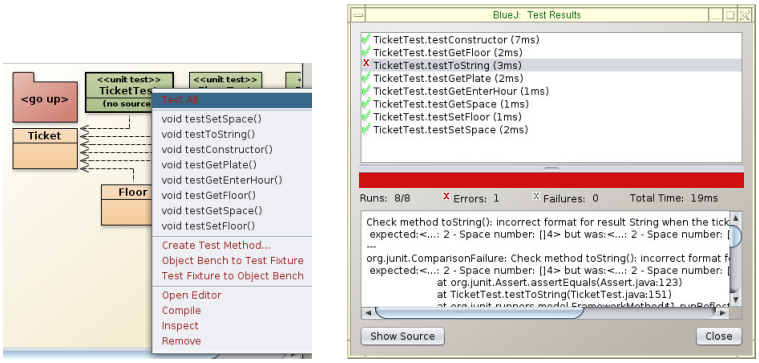


Figure 18: Execution of the unit test `TicketTest` and its *Test Results* window in *BlueJ*.

Activity 7: validation of the `Floor` class

To use the test of the `Floor` class, employ the option *Test All* from the pop-up menu that appears when right-clicking the icon of `FloorTest`. As you did in the validation for the classes of Activity 6, a set of test on the methods of `Floor` will be executed, comparing the obtained and the expected result. This test needs that the implementations of `TimeInstant` and `Ticket` are correct. Thus, first you have to pass the tests for those classes.

Activity 8: validation of the `Parking` class

Validate the `Parking` class by executing the `TestParking` test unit. This test supposes that the classes implemented until this moment are correct. Thus, you must pass successfully the tests for those classes before testing this last one.