

Assignment 1: Road Fighter Car

Submission date: October 18th 2021, 09:00

OBJECTIVE: Introduction to object orientation and to Java; use of arrays and enumerations; string handling with the `String` class; input-output on the console.

Copy Detection

For each of the TP assignments, all the submissions from all the different TP groups will be checked using anti-plagiarism software, firstly by comparing them all pairwise and secondly, by searching to see if any of their code is copied from other sources on the Internet¹. Any plagiarism detected will be reported to the *Comité de Actuación ante Copias* which, after interviewing the student or students in question, will decide whether further action is appropriate, and if so, will propose one of the following sanctions:

- A grade of zero for the TP-course exam session (*convocatoria*) to which the assignment belongs.
- A grade of zero for all the TP-course exam sessions (*convocatorias*) for that year.
- Opening of disciplinary proceedings (*apertura de un expediente académico*) with the relevant university authority (*Inspección de Servicios*).

1. Introduction

In this assignment, we will create a car-racing game inspired by the classic car-racing video games of the 90s — see Figure 1 for a screenshot of such a game — in which a player drives a car along a road, steering round obstacles and overtaking other cars until reaching the finish line. A notable feature of these games is that the player's car remains in the same place on the screen while the road moves towards it.

The following video shows an example of this type of game.

¹If you decide to store your code in a remote repository, e.g. in a free version-control system with a view to facilitating collaboration with your lab partner, make sure your code is not in reach of search engines. If you are asked to provide your code by anyone other than your course lecturer, e.g. an employer of a private academy, you must refuse.



Figure 1: Road Fighter - Nintendo 1991

<https://www.youtube.com/watch?v=ha6vRnauX84>.

The user-interface of our car racing game will be purely text-based with the commands being entered by the user on the command line. The game is represented on the screen as a grid, which we will refer to as the board, in which the player's car is situated in the first column as shown in Figure 2. As is usual in such car racing games, the car remains in the same column of the board while the other elements of the game move towards the car. The car can move up and down in its column to avoid obstacles.

In this first assignment, we create a very simplified version of the game. Figure 2 shows a screenshot of the user interface. We will progressively add more complexity, in particular, adding new types of obstacles and enemies, in later assignments.

```

Super cars 1.0

Level: EASY
Random generator initialized with seed: 366
Distance: 30
Coins: 5
Cicle: 0
Total obstacles: 8
Total coins: 18
Ellapsed Time: 0,00 s

+-----+
|               |
|               |
|               |
|               |
|               |
|               |
|               |
|               |
+-----+
|               |
|               |
|               |
|               |
|               |
|               |
|               |
|               |
+-----+

Command >

```

Figure 2: Screenshot of the user interface

2. Description of the assignment

The game comprises a road of length L and width (number of lanes) W , the value of both depending on the level. Only part of the length of the road appears on the screen at any one time. For example, on the easiest level of difficulty, the road has length 30 and width 3, while the fragment that appears on the screen (which we refer to as the board) is an 8×3 grid (8 columnas, 3 rows). The horizon or *visibility* is the length of the road visible on the screen from the car's position, so in the case of the easiest level, the visibility

is 8.

The player wins the game when the car reaches the end of the road (on the board, when the end of the road reaches the car!), shown on the screen as a finish line, see Figure 3.

```

Distance: 3
Coins: 12
Cicle: 21
Total obstacles: 15
Total coins: 2
Ellapsed Time: 15,40 s
=====
|
|
|
>  ¢  |
|
|
=====
Command >

```

Figure 3: Screenshot showing the finish line.

In the first assignment there are only the following three types of game element:

- **Player.** This is the car that the user operates. It remains in the first column of the board but the user can move it up or down in this column, as long doing so would not cause it to leave the road. The fact that the road moves continually from right to left means that a movement of the car up or down in the first column moves it diagonally up or down from left to right along the road.
- **Obstacle.** These appear randomly in the road and remain in the same position on the road. The car is destroyed if it collides with one of these obstacles.
- **Coin.** The player can collect coins that appear on the road by driving over them and, in future extensions of the game, will be able to use them to buy *power-ups*.

In each cycle of the game, the following actions are carried out:

1. **Draw.** The current state of the board and that of the game information is sent to the standard output.
2. **User action.** The game reads the user's next command, which may be to move the car, to ask for the help information, etc.
3. **Update.** The elements of the game are updated. Recall that in the current version of the game, the obstacles, once in the game, do not have any behaviour and are therefore not updated.
4. **Remove dead objects.** Any dead objects are removed from the road.
5. **Check end.** It is checked whether or not the game has finished.

After each movement of the car, it has to be checked whether the car has collided with an obstacle or whether the car has picked up a coin.

3. Game elements

In this section, we describe the type of objects that appear in the game and their behaviour. Please consult the `symbols.txt` file distributed with the problem statement to see the concrete characters to be used to represent each game element in the standard output².

Player

- **Behaviour:** Advances, i.e. moves along the road, horizontally or diagonally.
- **Speed:** 1 square per cycle.
- **Resistance:** 0 (it is destroyed on collision with an obstacle).

Obstacle

- **Behaviour:** Remains stationary on the road. The car is destroyed on colliding with it.
- **Resistance:** 1 resistance point (future extensions of the game will contain stronger obstacles which the player will be able to shoot or throw bombs at).

In the first assignment we also have **Coins** that the user can pick up and that will be used in future extensions of the game to buy **power-ups**.

Coin

- **Behaviour:** Remains stationary on the road. When the car drives over a coin, the player's score increases by 1 and the coin disappears.

We recommend you implement the assignment incrementally, in particular, leaving the implementation of the coins until after you have implemented the basic behaviour of the cars and the obstacles.

4. Actions

In this section we describe in more detail the actions that must be executed, in the order specified above, in each cycle.

4.1. Initialisation

At the start of the game, some obstacles are placed on the road, the number of them depending on the level of difficulty, this being one of the input parameters of the game. In this version of the game, there are three levels of difficulty: **TEST**, **EASY** and **HARD** (see Table 1.1).

The level also determines other characteristics of the game, namely:

²Note that the output uses UTF-8 characters so if you did not set the character set of your project to UTF-8 when you created it, as explained in Assignment 0, you can change it in `Project/Properties/Resource/TextFileEncoding`.

- the length of the road,
- the width of the road,
- the visibility (the length of the board in number of columns),
- the probability that an obstacle appears,
- the probability that a coin appears.

Level	length	width	visibility	Obs freq	Coin freq
TEST	10	3	8	0.5	0
EASY	30	3	8	0.5	0.5
HARD	100	5	6	0.7	0.3

Table 1.1: Configuration for each level of difficulty

For example, if the probability of an obstacle appearing is 0.5, statistically, there will be an obstacle every two columns, though since this feature is probabilistic, obstacles may be more, or less, spaced out than this. In the leftmost columns of the road there must not be any obstacles; obstacles can only be placed starting in column number $\text{visibility} / 2$.

If a pseudorandomly-generated decision is taken to place a game element in a pseudorandomly-generated location and it turns out that there is already a game element at that location, e.g. a **Coin** is to be placed on a tile where there is already an **Obstacle**, the new element is simply not added and no action is taken³.

4.2. Draw

On each cycle the current state of the board is sent to the standard output along with the game-state information which comprises:

- the game cycle number (initially 0),
- the distance to the finish line,
- the number of obstacles on the road,
- the number of coins on the road,
- the number of coins acquired by the player,
- the time that has elapsed since the start of the game.

Figure 4 shows an example of the text sent to the standard output on executing the `n` command (see later for an explanation of this command).

The timer is started on the first movement of the car in the game; we will use the `System.currentTimeMillis()` function to implement it.

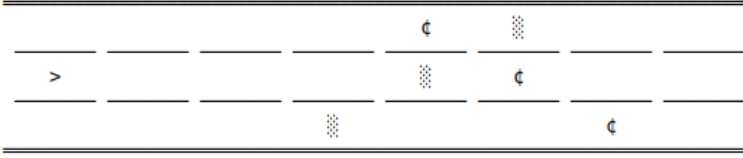
Each command, after being entered, is echoed on the screen with the prefix **[DEBUG]** Executing: . After it has been executed, the prompt (**Command >**), asking the user to enter the next command, is displayed.

³There were three options for the game design here: try again (taking care of possible infinite loops), choose randomly only from those tiles that are empty, or do nothing; we have chosen the third option, which is the simplest to implement.

```

[DEBUG] Executing: n
Distance: 29
Coins: 5
Cicle: 1
Total obstacles: 14
Total coins: 12
Ellapsed Time: 0,00 s

```



```

Command >

```

Figure 4: Output on executing the n command.

Recall that the length of the board, called the visibility, depends on the level and that the other game elements move across the board from right to left while the car stays in its column.

4.3. Update

In the first assignment, the only updating to be done is that of the position of the car, which can move stay still or move up or down in its column of the board (advancing either horizontally or in diagonal along the road). We also have to check that a vertical movement in its column of the board would not lead to it leaving the board (coming off the road).

4.4. User command

On seeing the prompt, the user can enter one of the following commands

- `info`: Displays information about each of the elements of the game.
- `q`: Moves the car upwards on the board (diagonally upwards on the road).
- `a`: Moves the car downwards on the board (diagonally downwards on the road).
- `none`: Moves the car horizontally on the road.
- `reset`: Restarts the game in the initial configuration.
- `test`: Changes the game mode to test, in which the elapsed time is not printed; this mode is explained in more detail below.
- `exit`: Exits the game after displaying the message "Game Over".
- `help`: Displays information about each of the existing commands, each on a new line using the following format: `<command_name> <command_parameters>: <command_description>`

```

[DEBUG] Executing: help
Available commands:
[h]elp: show this help

```

```
[i]nfo: print the game element info
[n]one | []: just update the game
[q]: move car up
[a]: move car down
[e]xit: exit the game
[r]eset: reset the game
[t]est: activate test mode
```

Observations concerning the commands:

- The command can be written in upper-case letters, lower-case letters or any mixture of the two.
- The first letter of each command can be used instead of the whole word.
- The empty command, i.e. simply pressing *return*, has the same effect as the none command.
- If the command does not exist (perhaps due to being badly written) or cannot be executed in the current state, a suitable error message is displayed.
- After the execution of a command that doesn't change the state of the game (info, test, exit, help), or after an error, the board should not be displayed.

The game ends when the player's car reaches the finish line or collides with an obstacle. A message should be displayed to distinguish between these two cases:

```
[GAME OVER] Player wins!
New record!: 4.85 s
```

or:

```
[GAME OVER] Player crashed!
```

Regarding the former message, in this first version of the game the current record is not stored between games so a new record is set on every game.

Observations concerning the probabilistic behaviour

A pseudo-random number generator produces a pseudo-random sequence of values, where the next value is calculated from the previous value or values; it is used by asking for the next value in the sequence, each time a random value is required. A *seed* for a pseudo-random number generator is the initial value with which to start the calculation of the pseudo-random sequence of numbers. For a given seed, a pseudo-random number generator produces exactly the same pseudo-random sequence of numbers.

Pseudo-random number generators in computing languages, such as the one encapsulated by the Java class `Random`, provide the programmer with the option of choosing the seed or not. In the former case, this ensures that the behaviour of a program involving pseudo-randomness is reproducible, since two different executions of such a program that use the same seed will produce exactly the same outcome, thus facilitating programming tasks such as debugging. In the latter case, the system generates a seed as randomly as

it can, ensuring that the behaviour of a program involving pseudo-randomness is different on each execution. For example, the Java class `Random` has a one-argument constructor which accepts a seed and a no-argument constructor, which generates its own seed⁴.

In programs involving pseudo-random behaviour, it is customary to offer the user the option of passing a seed for this behaviour to the program when starting it. This option is implemented in the code that we provide (see the file `SuperCars.java`). From inside the program, program parameters are obtained via the array of strings that is the argument of the `main` method. Observe that in the code for the `main` method contained in the file `SuperCars.java`, the `seed` parameter is optional. Observe also that the other parameter that is passed to the program on start-up (and that is not optional) is the `level`.

Clearly, in order for passing a seed to guarantee that the pseudo-random behaviour be reproducible, it is crucial that all pseudo-random behaviour of the application use that same seed. The simplest way to accomplish this is to ensure that the application create only one random-number generator (so, in our case, a single object of the class `Random`) and that all the pseudo-random behaviour of the application use this same generator.

5. Implementation

We start this section by observing that the quality of the implementation proposed here is relatively low, one of the main reasons for this being that it does not stick to the **DRY (Don't Repeat Yourself)** programming principle. The duplication of code in different parts of a program makes it less maintainable, less readable and less testable; modifying such a program is considerably more complicated and error-prone. The reason that the DRY principle is not correctly followed in the proposed implementation is that in this first assignment, we do not want to use *inheritance* and *polymorphism*, two basic tools of object-oriented programming. In the second assignment, we will refactorise the code, improving it by introducing these tools, thereby converting it into a *bona fide* object-oriented program.

5.1. Implementation details

Executing the class `es.ucm.tp1.SuperCars` starts the application, for which reason you are advised that all classes developed in the assignment should be inside the `es.ucm.tp1` package (possibly inside other packages that are inside this package). Recall that the Java naming convention is that package names begin with a lower-case letter while class names begin with an upper-case letter.

To implement the assignment, you must use, at least, the following classes:

- **Coin, Obstacle:** These classes encapsulate the behaviour of the game elements. They contain one or two attributes to store their position⁵ and an attribute containing (a reference to) the unique instance of the class `Game` that exists in any execution. There is no need for more than one object of the class `Game` so we will refer to this unique instance as *the game object* or simply *the game*. These game elements will need to invoke methods of the game to know whether or not they can carry out certain actions. Each of these classes also contains a static attribute which is a counter of how many instances of that class exist in the game at any one time. The game will access these attributes when it needs this information.

⁴In the code that we provide (see the file `SuperCars.java`), we generate a seed, instead of using the no-argument constructor, in order to facilitate leaving the creation of the `Random` object to the `Game` class.

⁵Whether this requires one or two attributes depends on whether or not you decide to implement a `Position` class. If you do, note that objects of this class should be immutable.

- **Game:** This class encapsulates the logic of the game. It contains an attribute to store the cycle counter and another to store (a reference to) the unique instance of the class `Player` that exists in any execution. There is no need for more than one object of the class `Player` so we will refer to this unique instance as *the player object* or simply *the player*. Among the methods of this class is an `update` method to update the elements of the game.
- **Player:** This class represents the player's car. It contains attributes to store the position, the number of coins the player has acquired etc. In the update phase of the game cycle, the car modifies its positions and then checks whether it has collided with any obstacles or driven over any coins.
- **ObstacleList, CoinList:** Each of these classes contains an attribute to store an array of the respective game elements, together with some methods for the management of these elements. In the first assignment, **the use of basic Java arrays is obligatory**; in subsequent assignments, we will instead use the library class `ArrayList`.
- **Level:** This is an `Enum`, which in Java is a kind of class. It contains attributes for each of the characteristics of the game that depend on the level.
- **GamePrinter:** This class contains attributes to store the dimensions of the board and an attribute to store (a reference to) the game. It has a `toString` method that builds the string which is sent to the standard output after executing a command that changes the state of the game.
- **Controller:** This class contains an attribute to store (a reference to) the game and an attribute (customarily called `in`) to store an instance of the library class `Scanner`. The latter object is used to read from the standard input the commands entered by the user via the keyboard. The `run` method of this class contains the main loop of the game, in which, while the game is not finished, it reads the next command from the standard input, calls the corresponding method of the game object and then sends the textual representation of the state of the game to the standard output, if it has changed after executing that command. There is no need for more than one instance of this class so we will refer to this unique instance as *the controller object* or simply *the controller*.
- **SuperCars:** This class contains the `main` method of the application. This method reads the values of the program parameters (`level` and `seed`, where the second is optional), creates an instance of each of the classes `Game` and `Controller`, passing the former as a parameter to the latter, and invokes the controller's `run` method to start the game.

Observations concerning the implementation

We also provide you with some code to help you get started.

The rest of the information needed to implement the assignment will be provided by the lecturer during the lectures and lab classes. The lecturer will give indications of which aspects of the implementation are considered obligatory in order to accept the assignment as correct and which aspects are left to the students' judgement.

Note also that in a *Problem-Based Learning* approach, the student is required to search for the knowledge they need to solve the problem at hand and to apply this knowledge to

solving the problem *before* the pertinent information and solutions is presented in lectures. Many studies have shown that knowledge is more easily absorbed and retained if it is acquired in this way, by the student working under the lecturer's guidance but independently. Moreover, perhaps the most important ability to be acquired at university is how to learn independently.

6. Submission

The assignment must be submitted as a single compressed (with zip) archive via the Campus Virtual submission mechanism not later than the date and time indicated at the start of this document⁶. The zip archive should contain at least the following⁷:

- A directory called `src` containing the Java source code of your solution,
- a file called `students.txt`, containing the names of the members of your group,

Do not include the files generated on compilation, i.e. the `.class` files, which, if you created your project following the procedure indicated in Assignment 0, should be in a separate directory called `bin`. Optionally, you can also include:

- a directory called `doc` containing the API documentation in HTML format generated automatically from the Java source code of your solution using the `javadoc` tool.

The contents of any `doc` directory you may include will not contribute to the grade in this assignment but may do so in subsequent assignments. Note that using `javadoc` involves adding comments to your code in the `javadoc` format, otherwise the HTML that you generate will contain almost no information.

7. Testing

Together with the instructions for the assignment, you have a directory of program traces. In the directory there are files following one of two nomenclatures:

- `easy_s5_0.txt`: the input for test case 0 with level `easy` and seed 5.
- `easy_s5_0_output.txt`: the expected output for this input.

To take the input from a given input text file and send the output to an output text file in Eclipse, you can configure the *redirection* of the standard input and standard output / standard error in the **Common** tab of the **Run Configurations** window as shown in Figure 5.

To obtain exactly the same output as the tests, you should take into account that in generating them:

- We have used the method `Random.nextDouble()` to know whether to add an object or not.
- If `frequencyElementLevel` is the frequency of that element appearing when playing at the given level as provided in Table 1.1, then we add the element to the table if `Random.nextDouble() < frequencyElementLevel`.

⁶To generate the zip file, you may find it helpful to use the Eclipse option `File > Export`.

⁷You may also include the project information files generated by Eclipse

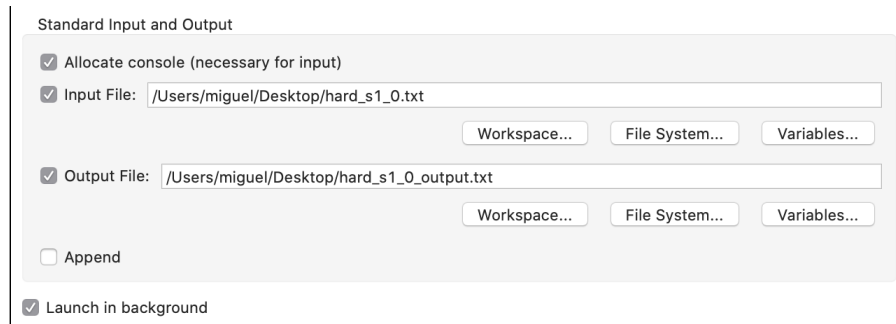


Figure 5: Redirection of the standard input, error and output.

- As stated in Section 4.1, elements can only added starting at `visibility / 2`.

In Listing 1.1, we provide you with pseudocode for adding new game elements:

- `tryToAddObstacle` and `tryToAddCoin` add the appropriate element if it is possible.
- `getVisibility()` returns the current visibility.
- `getRandomLane()` returns an integer chosen at random from the set $\{0, \dots, \text{number_of_lanes} - 1\}$ in order to add the element in that lane of the road.

```
for (int x = getVisibility () / 2; x < roadLength; x++) {
    tryToAddObstacle(new Obstacle(game, x, getRandomLane()), level.obstacleFrequency());
    tryToAddCoin(new Coin(game, x, getRandomLane()), level.coinFrequency());
}
```

Listing 1.1: Pseudocode for adding game elements

There are many free programs to visually compare files and thereby check that the output of your program coincides with the expected output for each of the test cases we provide. In particular, there is one integrated in Eclipse: select the files you wish to compare, press the right button and then select **Compare With > Each other** in the pop-up menu, see Figure 6 and Figure 7.

Other possibilities are *Beyond compare* (see <https://www.scootersoftware.com/>) or *DiffMerge* (see <https://sourcegear.com/diffmerge/>). Take care with the order of instructions in your program since it can have a significant effect on the output. If you detect an error in the output of any of the test cases provided please let the lecturer know ASAP so that we can correct it.

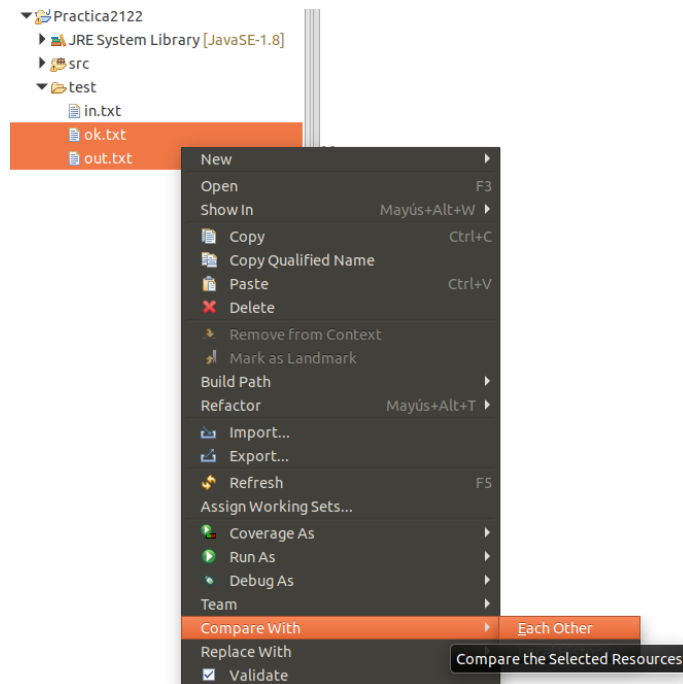


Figure 6: How to compare two text files in Eclipse.

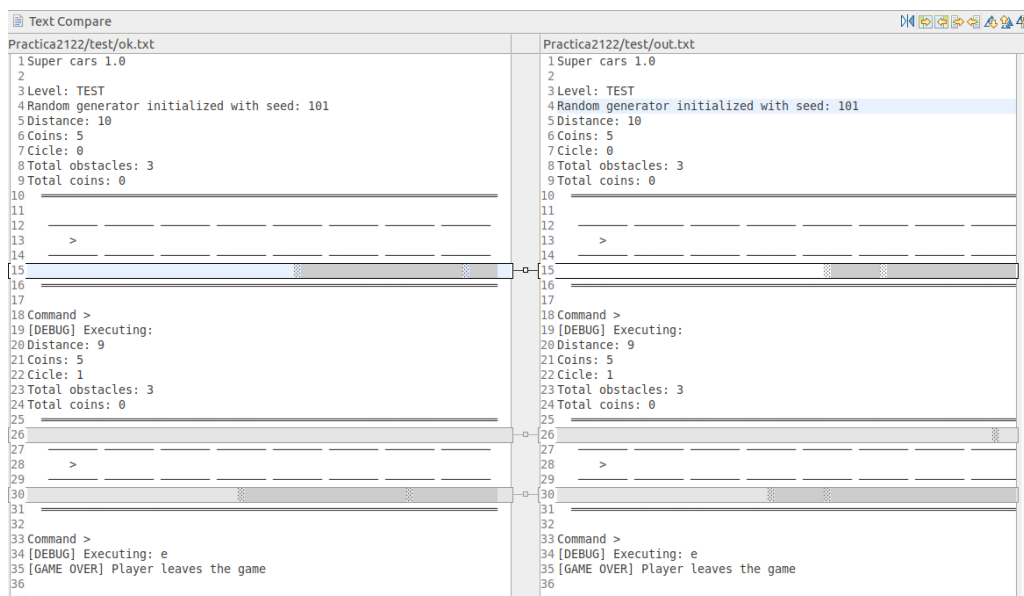


Figure 7: Output of the Eclipse file-comparison tool.