Assignment 3: Road Fighter v 3.0

Fecha de entrega: 13 Diciembre 2021, 9:00

Objetivo: Exception handling and file I/O

1. Introduction

In this assignment we extend the functionality of the *Road Fighter* game of the previous assignment in two ways:

- **Exception handling**: errors that may occur during the execution of the application can be more effectively dealt with using the exception mechanism of the Java language. As well as making the program more robust, this mechanism enables the user to be informed about the occurrence of an error in whatever level of detail is considered appropriate, while at the same time providing a great deal of flexibility in regard to where the error is handled (and the error message printed).

- **File handling**: a useful addition to the application would be the facility to save the state of a game to file and load the state of a game from file. To this end, we add two new commands, one to write to a file and the other to read from a file. The use of the Command pattern introduced in the previous assignment greatly facilitates the addition of new commands.

2. Exception Handling

In this section, we present the exceptions that should be handled by the application and give some information about their implementation, as well as providing a sample execution.

In the previous assignment, the `execute` method of each command simply invoked a method of the `Game` class which implemented the functionality of that command (or at least, it should have!). You will have observed that there are circumstances in which a command may fail, either in its parsing or in its execution. The execution of the `grenade` command, for example, will fail if the player does not have enough coins, or if the position chosen by the user at which to place the grenade is already occupied by another game object or is not on the visible part of the board.

In the previous assignment, the occurrence of such an error when implementing the functionality of the `grenade` command was likely communicated back to the execute method of the GrenadeCommand class via a boolean return value (or, if not, by some ad-hoc mechanism involving specific methods in the Controller class, obliging the game to know about the controller). This use of a boolean value to communicate the occurrence of an error is very limiting, since such binary communication cannot include any indication of the reason for the occurrence of the error, nor can it include any other data about the error which may be required to handle it adequately.

In this assignment, we deal with these issues using the Java exception mechanism. An exception mechanism provides a flexible communication channel between the location in the code where an error occurs and the location in the code where that error is handled, along which any required data concerning the occurrence of the error can be sent from the former to the latter[1]. In many cases, the data concerning the occurrence of the error that is transmitted from one code location to another via an exception consists simply of an error message, and the handling of this error consists simply of sending that message to the standard output to be displayed on the screen. In the general case, however, more data about the error and its context may be transmitted between code locations and the error-handling may require more complex actions than simply printing a message to the screen.

In particular, in this assignment, the exception mechanism enables us to ensure that all messages to be printed to the standard output can be printed from the controller. It should be pointed out that file handling inevitably involves exception handling, particularly when reading from files, which is why these two topics are often introduced to students at the same time. We will deal with these input-output exceptions in the second half of this document.

2.1. Types of exception

First, we discuss handling exceptions thrown by the system, that is, those that are not explicitly thrown by the programmer (most of which are also created by the programmer, though the programmer can also explicitly throw system exceptions). You should at least handle the following exception:

- NumberFormatException, which is thrown when an attempt is made to parse the String-representation of a number and convert it to the corresponding value of type int or long or float, etc., in the case where the input string does not, in fact, represent a number and cannot, therefore, be so converted.

Regarding programmer-defined exceptions (which, clearly, can only be thrown by the programmer), you should create the following three exception classes: GameException, CommandParseException and CommandExecuteException, where the latter two exceptions inherit from the former and are to be used when an error occurs on parsing a command / executing a command respectively. These two exceptions are to be considered high-level exceptions. You should also create low-level exceptions but these should not reach the controller, that is, they should be caught in the methods of the command classes and

[1]The error code mechanism of C and C++ is somewhat primitive in comparison, though it is also much less computationally costly, which is why C++ retains it as well as having an exception mechanism (though this exception mechanism is less type-checked and more difficult to use than the Java equivalent).

then wrapped in one of the two high-level exceptions. An example of an error that should produce an exception that will reach the controller wrapped in a CommandParseException is that provoked by a `grenade` command with an incorrect number of arguments. An example of an error that should produce an exception that will reach the controller as a CommandExecuteException is an attempt to add a grenade in a position on the board that is already occupied by another game element. Any exceptions thrown by the system during parsing, e.g. NumberFormatException, should also be wrapped in a CommandParseException.

In the Road Fighter application, the exception mechanism can be used to ensure that all printing to the standard output or standard error is done from the controller (in the case of error messages, in a catch clause of the run method), with the exception of the execute method of the HelpCommand and later perhaps other commands. It will now be much easier to use more specific error messages than in the previous assignment.

2.2. Implementation

Summarising, you need to make the following changes to your application:

1. Define new (high-level) exception classes: GameException with subclasses CommandParseException and CommandExecuteException as well as some low-level exceptions, see below. As an example of a CommandParseException, consider the following version of the parse() method of the Command class:

```
protected Command parse(String[] words) throws CommandParseException {
    if (matchCommandName(words[0]))
        if (words.length != 1)
            throw new CommandParseException(String.format("Command %s: %s"
                , name, INCORRECT_NUMBER_OF_ARGS_MSG));
        else
            return this;
    return null;
}
```

   Note that checking the number of words, and throwing an exception if not correct, before checking the text of the first word would disrupt the parsing and sabotage the working of the command pattern since the parse methods that follow in the AVAILABLE_COMMANDS array would not be checked.

2. Any parse method that can throw a CommandParseException and any execute method that can throw a CommandExecuteException will need to declare this fact.

3. The getCommmand() of the Command class now throws a CommandParseException if none of the executions of the parse method in the different Command subclasses return null[2]:

   throw new CommandParseException(UNKNOWN_COMMAND_MSG);

   instead of also returning null and expecting the Controller to check for this.

4. The exceptions CommandParseExceptions and CommandExecuteExceptions should be caught in the run method) method of the controller. The easiest way to do this is by

[2]Since Java 8, it would probably be better for any command parse methods to return an object of the class Optional instead of returning either a Command subclass object or null but, like ArrayList, this class uses Java Generics so we won't discuss it further here.

explicitly declaring that it catches exceptions of the type defined by the superclass of these two classes. The code of this method will then have the following form[3]:

```
while (!game.isFinished()) {
   if (refreshDisplay)
      printGame();
   refreshDisplay = false;

   System.out.println(PROMPT);
   String s = scanner.nextLine();

   String[] parameters = s.toLowerCase().trim().split("\\s+");
   System.out.println("[DEBUG] Executing: " + s);
   try {
      Command command = Command.getCommand(parameters);
      refreshDisplay = Command.execute(game);
   }
   catch (GameException ex) {
      System.out.format("[ERROR]: %s %n %n", ex.getMessage());
   }
}
```

5. Define and use the following (low-level) exception classes:

   - InvalidPositionException: thrown when a position provided by the user is occupied, is off the road or is off the allowed part of the road.
   - NotEnoughCoinsException: thrown when it is not possible to perform an action requested by the user due to the player not having enough coins.
   - InputOutputRecordException: thrown when a problem occurs in the reading or writing of a record.

   You may also define and use other low-level exceptions if you wish.

2.3. Exception-handling good practice

As the Joshua Bloch quote[4] at the beginning of the exceptions chapter of the slides makes clear, good practice is particularly important when using exceptions, that is to say, the programming language itself and its compiler do not provide you with much help regarding what is a good use and what is a bad use of exceptions. For this reason, here we provide you with three further guidelines:

- **Multiplicity of the relation between errors and exception classes** It is not always obvious how specific programmer-defined exception classes should be. Clearly, defining only one class to handle all application errors would not be a good use of exceptions but nor would defining an exception class for every single specific error. Good practice is to follow the policy of the built-in exception classes which

[3]If you want to print both the message from the high-level exception and the message from the low-level exception it may contain, you should use the method getCause of the Exception class, after first checking whether it returns null (the case where the high-level exception does not contain a low-level exception).

[4]"When used to best advantage, exceptions can improve a program's readability, reliability, and maintainability. When used improperly, they can have the opposite effect."

is somewhere in-between these two extremes. For example, the built-in exception ArithmeticException is a leaf class of the built-in exception inheritance hierarchy (i.e. it has no subclasses) and division-by-zero is one of the several possible errors which provokes the system to throw this exception, the different errors of this exception being distinguished by the error message. Thus, the error message should not be a constant of an exception class so as to permit different instances of the same exception class to contain different messages[5].

- **LBYL vs EAFP**. An error-indication passes along a chain of method calls from the method where the error occurs to the method that handles it. The question arises: in which of the methods should the exception be thrown and in which should it be caught? For example, suppose that a method $m1$ calls a method $m2$ that calls a method $m3$ and an error may occur during the execution of method $m3$ which we want to handle in method $m1$. When the error occurs, two possible policies are as follows:

  - $m3$ returns an indication of error (without throwing an exception), say, the value false, to $m2$, $m2$ throws an exception on receiving the return value false from $m3$, $m1$ catches the exception.
  - $m3$ throws an exception when the error occurs, $m2$ doesn't catch the exception, $m1$ catches the exception.

  The implementation that is most coherent with the philosophy of exceptions is the second. Thus, methods should not return a boolean value when an error occurs but instead should throw an exception; in this way, exceptions are thrown as far down the call stack as possible, i.e. at the point in the code where the error occurs. This policy is often referred to as EAFP ("it is Easier to Ask for Forgiveness than Permission"), the policy of using boolean-valued functions to check if something is possible before doing it being known as LBYL ("Look Before You Leap"). You should be aware of a small complication that can arise from full use of the EAFP policy. For example, assuming that it is the player's responsibility to throw the NotEnoughCoinsException, if the game successfully removes from the player the coins needed to add a grenade but the following action of trying to add a grenade to the road fails due to the position being invalid, the game will need to give the coins back to the player[6].

- **Wrapping low-level exceptions in high-level exceptions**. It is often good practice to catch a low-level exception and then thrown a high-level exception that wraps it and *which contains a less specific message than the low-level one*[7]. Assuming you have already implemented the EAFP policy explained in the previous item, you should now ensure that any low-level exception thrown during the execution of a command in the game, the player, the gameElementGenerator, the gameElementContainer, etc. is caught in the execute method of the corresponding Command subclass which then throws a CommandExecuteException that wraps it.

[5]Even if, at the moment, there is only one possible error and message for a given programmer-defined exception, this should be open to extension to multiple errors/messages.

[6]Note that it is the game that knows whether or not a position is valid, so the game must throw the InvalidPositionException, not the container.

[7]You will probably have seen examples of bad practice in this regard, e.g. electronic commerce web applications that produce messages for the user of the type "SQL error...". The user does not care about this level of implementation detail and should not be receiving this type of message.

2.4. Sample Execution

In the following execution, both the message from the low-level exception and that from the high-level exception are printed (in that order). This is done for illustrative purposes; in a real application, only the high-level message would be printed for the user (indeed, this is the main reason for wanting to wrap the low-level message in a high-level message). If you wish to do the same, note that **both** print instructions should be in the Controller run method, the first following a use of the getCause method to extract the low-level exception from inside the high-level exception.

```
[DEBUG] Executing: s
Distance: 100
Coins: 1
Cycle: 0
Total obstacles: 86
Total coins: 25
Supercoin is present
Elapsed Time: 0.00 s
  ══════════════════════════════════════════════════════
                                          ░         ¢
   ───────  ───────  ───────  ───────  ───────  ───────

   ───────  ───────  ───────  ───────  ───────  ───────
      >                                              $
   ───────  ───────  ───────  ───────  ───────  ───────
                                 ░
   ───────  ───────  ───────  ───────  ───────  ───────
                                          ¢
  ══════════════════════════════════════════════════════

Command >
g 5 0
[DEBUG] Executing: g 5 0
Invalid position.
[ERROR]: Failed to add grenade

Command >
g 1 0
[DEBUG] Executing: g 1 0
Not enough coins
[ERROR]: Failed to add grenade
Command >
```

3. Serializing and saving

In computing, the term *serialization* refers to converting the state of an executing program, or of part of an executing program, into a stream of bytes, usually with the objective of saving it to a file or transmitting it on a network. The term *deserialization* refers to the inverse process of reconstructing the state of an executing program, or part of an executing program, from a stream of bytes. Serialization/deserialization in which the generated stream is a text stream is sometimes referred to as *stringification/destringification*. Clearly, the format used for serialization/stringification should be designed in such a way as to facilitate deserialization/destringification.

Here, our interest is to produce a text stream that represents the current state of the game, rather than the complete current state of the executing program, with a view to writing this state to, and reading this state from, a text file. Note that the textual representation produced by the game printer is not a suitable text-serialization format since text-deserialization of this format would be a rather complicated enterprise. We therefore define a *text-serialized* format, in which the state of the game is represented as a sequence of game elements, each of which is represented as a sequence of values. This format is useful for debugging purposes and also for saving the state of the game to a text file.

The serialize command

We first create a new command serialize, abbreviated with the letter z, which sends a text-serialized version of the game to the standard output as shown in the following example:

```
═════════════════════════════════════════════════════════════════════════
                                                              ¢        ░
 _______  _______  _______  _______  _______  _______  _______  _______
    >                                            ░
 _______  _______  _______  _______  _______  _______  _______  _______
                                         ░
═════════════════════════════════════════════════════════════════════════
```

```
Command >
z
[DEBUG] Executing: z
---- ROAD FIGHTER ----
Level: TEST
Cycles: 0
Coins: 5
ElapsedTime: 0
Game Objects:
> (0, 1)
░ (4, 2)
░ (5, 1)
¢ (6, 0)
░ (7, 0)
░ (9, 2)
```

From this example, we see that the text-serialized format comprises: the information about the general state of the game followed by the text-serialized version of each object present on the road, where the objects are ordered from top to bottom and from left to right. The text-serialized version of each object consists of its symbol followed by its position, except in the case of some of the advanced objects, for which additional information must be provided:

- The text-serialization of a grenade object must include the number of cycles left until it will explode.
- the text-serialization of a pedestrian object must include the direction of its next movement (one of the words “up” or “down”).
- the text-serialization of a wall object must include the number of lives it has left.

This is shown in the following example:

```
 ══════════════════════════════════════════════════════
                                              ░       ¢
  ________ ________ ________ ________ ________ ________
                      ð[3]
  ________ ________ ________ ________ ________ ________
      >                                               $
  ________ ________ ________ ________ ________ ________
                                     ░
  ________ ________ ________ ________ ________ ________
                                              ¢
 ══════════════════════════════════════════════════════
```

```
Command >
z
[DEBUG] Executing: z
---- ROAD FIGHTER ----
Level: HARD
Cycles: 0
Coins: 2
ElapsedTime: 0
Game Objects:
> (0, 2)
ð[3] (2, 1) 3
░ (3, 3)
░ (4, 0)
¢ (4, 4)
¢ (5, 0)
$ (5, 2)
>>> (6, 4)
¢ (7, 4)
☺ (8, 0) down
...
░ (16, 3) 1
```

To implement the serialize command, apart from introducing a SerializeCommand class, you should create a new *view* represented by a class called GameSerializer which will be similar to the GamePrinter class. In fact, though it is not obligatory, you may wish to create an abstract class View from which these two classes derive, in order to encapsulate the functionality common to both of them.

The `save` command

The `save` command can be implemented by simply writing a header to the file followed by the text-serialized game state.

The execute() method of the SaveCommand class should take into account the following:

- To simplify, you do not need to check whether the text provided by the user can be a valid file name, this notion being operating-system dependent, nor whether the program has write permissions for the file, if it already exists. If these conditions are not met, on attempting to open the file, the JVM will throw an exception that you should catch and wrap in a CommandExecuteException.

- Add the extension `'.txt'` to the file name provided by the user. If a file with this name already exists it will be overwritten and if not does not, it will be created (you do not need to do anything to get this behaviour since it is the default behaviour in Java).
- In Java there are many different ways to connect a program to a file; we propose the following: create a FileWriter object and a BufferedWriter object that wraps it. Place the code which opens the file in a *try-with-resources* construct, catching any possible IOExceptions.
- If the state of the game has been successfully saved to file, print the following message on the screen (from the execute() method of the SaveCommand class; no need to print it from the controller):
  "Game successfully saved in file <*filename_provided_by_the_user*>.txt".

Ideally, the *Model* part of the application should have no knowledge of saving to files so implementing this command should not involve adding any code to the game. Ideally also, the `save` command should be implemented at the same time as a `load` command, to load a game state from file. However, in order to simplify the assignment, you are not required to implement the `load` command.

The `dump` command

To check that the game state has been correctly saved in a file by the `save` command we implement the `dump` command which simply reads a file line-by-line and sends the contents to the standard output.

3.1. Saving and loading "top scores"

The last extension we implement is the saving and loading of the record game-completion times in a file with the name `record.txt` having the following format:

```
HARD:22340
TEST:1760
EASY:6057
ADVANCED:1030
```

Observe that:

- A record is saved for each level.
- The different records can be saved in any order.
- The unit used is milliseconds.
- When attempting to read the records file (at the start of the game or when the `reset` command is used):
  - if the file does not exist or there is a permissions problem (catch FileNotFoundException) or the file is corrupted (catch NumberFormatException), an InputOutputRecordException with a suitable message (and wrapping the original system exception) should be thrown and a new records file should be created. In the case of a reset, this new records file may contain the current records of the application, otherwise, it will contain default values.

- if any other IO problem occurs, an InputOutputRecordException with a suitable message (and wrapping the original system exception) should be thrown and, after the error message has been printed, the game should finish.

- When attempting to write the records file (at the end of a game and only if a new record has been obtained):
  - if an IO problem occurs, an InputOutputRecordException with a suitable message (and wrapping the original system exception) should be thrown and, after the error message has been printed, the game should finish.
  - if there is no record for a given level, the default value should be used for that level.
- InputOutputRecordExceptions should be handled in the same way as any other low-level exception.
- Ideally, all this functionality should be encapsulated in a new class called Record.

Finally, the greeting on the screen when the game starts should now show the record for the current level (in seconds with two decimal places).

3.2. Summary

The output of the help command is now as follows:

```
Available commands:
[h]elp: show this help
[i]nfo: print game element info
[n]one | []: update
[q]: go up
[a]: go down
[e]xit: exit game
[r]eset [<level> <seed>]: reset game
[t]est: enable test mode
[s]hoot: shoot bullet
[g]renade <x> <y>: add a grenade in position x, y
[w]ave: do wave
seriali[z]e: text-serialize the state of the game
sa[v]e <filename>: save the state of the game to a file.
[d]ump <filename>: show the content of a saved file
Clear [0]: clear the road
Cheat <AO-name>: remove all elements of last visible column and add advanced object A
```

3.3. Test cases

The test cases for Assignment 3 are the same as those for Assignment 2 with some small differences in the error messages and the addition of two more:

- 19_easy_s37.txt: Test the save, serialize and dump commands
- 20_easy_s37.txt: Prueba excepciones.

Use of the tests is not obligatory (see the last section of the Assignment 2 problem statement for more details).