
Práctica 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 latter¹. 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 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

¹The 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).

is, they should be caught in the methods of the command classes and then wrapped in one of the two high-level exceptions.

2.2. Aspectos generales de la implementación

Una de las principales modificaciones que realizaremos al incluir el manejo de excepciones en el juego consistirá en ampliar la comunicación entre los comandos y el controlador. Continuará existiendo la previa, a través del booleano de retorno del método `execute()` que indica si se ha actualizado el tablero y, por tanto, este debe ser mostrado. Pero también se contemplará la posibilidad de que se haya producido un error de forma que, controlando el flujo de las excepciones que puedan producirse durante el parseo o la ejecución de un comando, se podrá informar del error al usuario. Además, puesto que ahora se van a tratar las situaciones de error tanto en el procesamiento como en la ejecución de los comandos, los mensajes de error mostrados al usuario podrán ser mucho más descriptivos que en la práctica anterior.

Básicamente, los cambios que se deben realizar son los siguientes:

1. La cabecera del método abstracto `parse()` de la clase `Command` pasa a poder lanzar excepciones de tipo `CommandParseException`:

```
public abstract Command parse(String[] commandWords) throws CommandParseException;
```

2. La cabecera del método abstracto `execute()` de la clase `Command` pasa a poder lanzar excepciones de tipo `CommandExecuteException`:

```
public abstract boolean execute(Game game) throws CommandExecuteException;
```

3. La cabecera del método estático `getCommmand()` de `Command` pasa a poder lanzar excepciones de tipo `CommandParseException`:

```
public static Command getCommmand(String[] commandWords) throws CommandParseException;
```

de forma que el método lanza una excepción del tipo

```
throw new CommandParseException(String.format("[ERROR]: %s", UNKNOWN_COMMAND_MSG));
```

en caso de comando desconocido, en lugar de devolver `null` y esperar a que `Controller` trate el caso mediante un simple `if-then-else`.

4. El controlador debe poder capturar, dentro del método `run()`, las excepciones lanzadas por los dos métodos anteriores, que son subclase de una nueva clase de excepciones `GameException`

```
public void run() {  
    ...  
    while (!game.isFinished()) {  
        ...  
        try { ... }  
        catch (GameException ex) { ... }  
    }  
}
```

5. Se deben definir nuevas clases (`GameException`, `CommandParseException`, `CommandExecuteException` y algunas más) y lanzar excepciones de estos tipos y tratarlas adecuadamente de forma que el controlador pueda comunicar al usuario los problemas

que ocurran. Por ejemplo, el método `parse()` de la clase `Command` que se proporcionó en la práctica anterior pasa a ser de la siguiente forma:

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

Haciendo esto así, todos los mensajes se imprimen desde el bucle del método `run()` del controlador cuyo cuerpo se parecerá al código siguiente:

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

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

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

2.3. Nuevos tipos de excepciones

En el desarrollo de esta práctica debes definir (en algún caso solo incluir y manejar), lanzar y capturar excepciones de, al menos, los siguientes tipos (en la sección siguiente aparecerá alguno más):

- **GameException**: es la superclase de las excepciones que se deben definir y de la que heredan las subclases `CommandParseException` y `CommandExecuteException`.
- **CommandParseException**: nuevo tipo de excepción lanzada por algún error detectado en el parseo de un comando.
 - **NumberFormatException**: excepción del sistema lanzada cuando un elemento proporcionado por el jugador debería ser un dato numérico y no lo es.

- **CommandExecuteException**: nuevo tipo de excepción lanzada por algún error detectado en la ejecución de un comando. Son subclases de esta las siguientes:
 - **InvalidPositionException**: nuevo tipo de excepción lanzada cuando una posición de la carretera proporcionada por el usuario está ocupada o no pertenece a la carretera.
 - **NotEnoughCoinsException**: nuevo tipo de excepción lanzada cuando no es posible realizar alguna acción pedida por el usuario al no tener el jugador suficiente saldo para llevarla a cabo.
 - **InputOutputRecordException**: nuevo tipo de excepción lanzada cuando hay problemas en la lectura o escritura del récord.

Una buena práctica en el tratamiento de excepciones consiste en recoger una excepción de bajo nivel para a continuación lanzar una de alto nivel que envuelve la anterior y que contiene otro mensaje que, aunque necesariamente menos específico que el de la de bajo nivel, es también de utilidad. Por ejemplo, en el comando **Grenade** podemos hacer lo siguiente:

```
catch (NotEnoughCoinsException e) {
    System.out.println(e.getMessage());
    throw new CommandExecuteException(String.format(" [ERROR] :  %s", FAILED_MSG)
        , e);
}
```

Es decir, se captura la excepción de más bajo nivel **NotEnoughCoinsException**, se muestra el mensaje de error y se relanza como excepción de más alto nivel **CommandExecuteException**.

Veamos un ejemplo:

```
[DEBUG] Executing: s
Distance: 100
Coins: 1
Cycle: 0
Total obstacles: 86
Total coins: 25
Supercoin is present
Ellapsed Time: 0.00 s
```

				⌘	¢
>					\$
			⌘		
				¢	

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


En la versión serializada de los objetos mostramos su símbolo y su posición. En algunos objetos avanzados necesitaremos más información. Veamos otro ejemplo:

				⌘	¢
		ð[3]			
>					\$
			⌘		
				¢	

```

Command >
z
[DEBUG] Executing: z
----- ROAD FIGHTER SERIALIZED -----
Level: HARD
Cycles: 0
Coins: 2
EllapsedTime: 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

```

Más concretamente:

- En la granada mostraremos el tiempo que falta para explotar.
- En el peatón, la dirección hacia la cual se está moviendo.
- En el muro, cuánta resistencia le queda.

Para hacer el comando `serializer` la idea es hacer una nueva *vista* que llamaremos `GameSerializer` y que será muy parecida a `GamePrinter`. De hecho aunque no es obligatorio podemos programar una clase abstracta `View` que encapsule la funcionalidad común entre las dos vistas.

Aunque no lo vamos a implementar, con esta información podríamos reconstruir el estado del juego programando un comando `LoadGame`.

SaveCommand

Para implementar este comando se debe definir la clase `SerializeCommand`. A partir de ella no es difícil definir la clase `SaveCommand` cuya ejecución es similar a la serialización,

pero volcando el resultado en un archivo, al que se le añade un encabezamiento.

En Java existen muchos mecanismos para manejar ficheros. En esta práctica usaremos flujos de caracteres en lugar de flujos de bytes. En particular, recomendamos el uso de `BufferedWriter` y `FileWriter` para escribir en un fichero. Además, se recomienda el uso de bloques `try-with-resources` para el código donde se abre el fichero, capturando `IOException` en el método `execute()` de la clase `SaveCommand`.

Para poder guardar el estado de una partida, deberán tenerse en cuenta las siguientes consideraciones:

- El método `parse()` de la clase `SaveCommand` debe lanzar excepciones de tipo `CommandParseException` si no se proporciona argumento o se proporciona más de uno.
- El método `execute()` de la clase `SaveCommand` debe hacer lo siguiente:
 - La clase tiene un atributo `String filename` para guardar el nombre del archivo que se crea al ejecutar el comando. Para simplificar, vamos a suponer que el nombre proporcionado por el usuario es correcto en varios sentidos. Primero, porque no se va a analizar si los caracteres que componen el nombre conforman un nombre válido de archivo, algo dependiente del sistema operativo. Segundo, porque no se va a comprobar si el programa tiene permisos de escritura en el fichero. Y tercero, porque aunque quisiéramos guardar el juego en un archivo que ya existe, el resultado de la ejecución del comando hará que ese archivo se sobrescriba.
 - Una vez conocido el nombre, el atributo `filename` de la clase se completará con una extensión `.txt`. Por ejemplo, si escribimos el comando `s datos`, la ejecución del juego creará un archivo `datos.txt`.
 - Se debe crear un `FileWriter` junto con un `BufferedWriter` que lo envuelva.
 - Se debe utilizar el `GameSerializer` para generar un `String` con los datos del juego y después volcarlos en el archivo.
- En caso de haber podido guardar el estado del juego en un archivo – con éxito – se debe imprimir el mensaje:

“Game successfully saved in file <nombre_proporcionado_por_el_usuario>.txt”.

Lo ideal es que el modelo no sea consciente de que se ha hecho un `Save`, es decir, para este comando no debemos añadir nada en la clase `Game`.

DumpCommand

Para comprobar que se ha guardado correctamente la información en el fichero vamos a implementar un comando `Dump`, que básicamente vuelca el contenido de un fichero, línea a línea, en la consola.

2.5. Guardar y cargar Record

La última extensión que vamos a hacer es la funcionalidad de `Guardar` y `Cargar` el record del juego. El record se cargará al principio de la partida o cuando hay `reset`. Veamos el formato del fichero `record.txt`:

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


- Guardaremos un récord por cada nivel.
- Los records no deben almacenarse en ningún orden concreto.
- El récord se almacena en milisegundos.
- Si el fichero de records no existe o está corrupto se debe lanzar una excepción `InputOutputRecordException` y terminar el juego.
- Si el récord de un nivel no existe se debe crear un valor por defecto (el máximo valor de `Long`, por ejemplo).
- Lo ideal es crear una nueva clase `Record` para encapsular esta funcionalidad.

Para facilitarnos el testeo de esta funcionalidad vamos a crear un comando `ShowRecordCommand`, usando la letra *o*, que muestra el récord del nivel actual por pantalla (el récord se muestra formateado en segundos con dos decimales):

				⌘	¢
>					\$
			⌘		
				¢	

```
Command >
o
[DEBUG] Executing: o
HARD record is 22.34 s
```

2.6. Resumen

La lista de comandos disponibles en el `help` queda ahora de la siguiente manera:

```
Available commands:
[h]elp: show this help
[i]nfo: prints gameobject info
[n]one | []: update
[q]: go up
[a]: go down
[e]xit: exit game
[r]eset [<level> <seed>]: reset game
[t]est: enables test mode
[s]hoot: shoot bullet
[g]renade <x> <y>: add a grenade in position x, y
[w]ave: do wave
seriali[z]e: Serializes the board.
sa[v]e <filename>: Save the state of the game to a file.
[d]ump <filename>: Shows the content of a saved file
rec[o]rd: show level record
Clear [0]: Clears the road
Cheat [1..5]: Removes all elements of last visible column, and adds an Advanced Object
```

2.7. Casos de prueba

Los tests deben coincidir con la práctica anterior, salvo algún detalle de los mensajes de error. Se han añadido dos tests:

- 19_easy_s37.txt: Comprobar el comando save, serialize y dump.
- 20_easy_s37.txt: Prueba excepciones.