Functions and error handling

Exception management

Along the development of a Java program we can find two types of errors: compilation errors and runtime errors. The first ones are detected by the compiler as we type the code, whereas runtime errors are difficult to predict (in general): network errors, dividing by zero, file not found... Most of these runtime errors can be handled by **exceptions**.

1. What is an exception?

An exception is an event that happens during the execution of a program and makes it exit from its normal instruction flow. This way, we can deal with the error in a smart way, by separating the "normal" code from the error itself. Whenever an exception occurs, we say that it has been **thrown**, and we can choose among propagating it (throwing it again) or **catching** it and process the error. We will see these two options in a few minutes.

1.1. Exception types

Runtime errors can be of two main types:

- **Errors**: in this case, we talk about *fatal* errors that happen during the execution of a program, such as hardware errors, memory errors... These errors can't be managed from within a Java application.
- **Exceptions**: they are non-critical errors that can be managed (files not found, parsing errors...). Inside this type of errors, we can talk about:
  - **Runtime exceptions**: they don't need to be catched, and they are difficult to predict, in general. For instance, assigning null to a variable, or going beyond the boundaries of an array.
  - **Checked exceptions**: these exceptions need to be catched, or declared to be thrown. In other words, if we use a function that can throw these type of exceptions, the compiler will complain if we don't catch the exception or throw it again. For instance, whenever we call the `Thread.sleep` instruction, we need to catch or throw an `InterruptedException`.

However, every type of exception is a subtype of the `Exception` main type. This generic type stores the error message produced by the exception. There are some other subtypes that store more specific information. For instance, `ParseException` is a subtype of `Exception` that is thrown whenever data can't be properly parsed. It stores the error message along with the position where the error was found.

1.2. Types of exception management

Whenever an exception is caused in a program, we can decide how to treat it. Basically, we have two options in our code:

- **Catch** the exception. This means that exception is "destroyed" and we can show a controlled, customized error message instead.
- **Throw** the exception. In this case, we don't want to care about the exception, and we delegate in another piece of code to treat it.

In next sections of this document we will learn how to manage these two options.

2. Catching exceptions

Whenever a piece of code can throw an exception, we can catch it by using a `try..catch` block. We put inside the `try` clause the code of our program that may produce an exception, and we use the `catch` clause to respond to the specified error. We can just output an error message, or return a given value, among other possible options.

This example tries to convert a string into an integer value. If the conversion can't be done because the input is not valid, then a `NumberFormatException` will be thrown, and we can produce an appropriate error message in the `catch` clause.

```java
int number;
string text = ... // Whatever value

try{
    number = Integer.parseInt(text);
} catch (NumberFormatException e) {
    System.err.println("Error parsing text: " + e.getMessage());
}
```

The `getMessage` method gets the error message produced by the exception. See that we are using `System.err` instead of `System.in` because we are printing an error, and then we should use the default error output instead of the default "normal" output.

We can also use `printStackTrace` method to print a complete stack trace of the error, so that we can see the call stack that have produced the error (this is, methods that have been called until the error was produced).

```
int number;
string text = ... // Whatever value

try{
    number = Integer.parseInt(text);
} catch (NumberFormatException e) {
    e.printStackTrace();
}
```

We can add as many `catch` clauses as we need, and each one can represent a specific exception type:

```
try{
    // Code that may fail
} catch (NumberFormatException e1) {
    // Error message for number format
} catch (ArithmeticException e2) {
    // Error message for dividing by zero
...
} catch (Exception eN) {
    // Error message for any other error
}
```

However, we must put these `catch` clauses in order, so that the most generic ones are placed at the end, because the program will enter at the first `catch` clause that matches the exception produced. In other words, if we put the `catch(Exception)` clause at the beginning, the rest of clauses will have no effect, since any of them are subtypes of `Exception` and thus, they will be catched by the first clause.

There are some instructions that force us to deal with a specific type of exception. For instance, if we call `Thread.sleep` instruction, the compiler will ask us to deal with an `InterruptedException`. We can do it this way:

```
try{
    Thread.sleep(5000);
} catch (InterruptedException e) {
    System.err.println("Interruption during sleep: " + e.getMessage());
}
```

However, we can also use a generic `Exception` element in the `catch` clause to deal with any type of exception. We only need to specify concrete types of exceptions if we want to manage different `catch` clauses, and then, produce different error messages depending on the exception produced.

3. Throwing exceptions

The second way of managing an exception is throwing it. This way, we pass it to the next function in the stack call... until we reach the *main* function (in this function we should no longer throw exceptions, we must catch them).

For instance, this function receives two numbers and returns the division n1 / n2. But if n2 is 0, we can't divide them, so in this case we can throw a new exception to indicate that data is not correct:

```
public static int divide (int n1, int n2){
    if (n2 == 0)
        throw new ArithmeticException("Can't divide by zero");
    else
        return n1 / n2;
}
```

So, if we try to use this method in our program, we must be aware that an exception can be thrown, and catch it:

```
public static void anotherFunction(){
    int number1, number2;

    // ... Ask user to fill number1 and number2

    try    {
        int result = divide(number1, number2);
    }catch (ArithmeticException e)
    {
        System.err.println("Error: " + e.getMessage());
    }
}
```

Now, let's have a look at this example:

```java
public static void a() throws InterruptedException{
    throw new InterruptedException ("Exception in a");
}

public static void b() throws InterruptedException{
    a();
}

public static void c() throws InterruptedException{
    b();
}

public static void d() throws InterruptedException{
    c();
}

public static void main(String[] args){
    try
    {
        d();
    } catch (InterruptedException e) {
        System.err.println("Exception: " + e.getMessage());
    }
}
```

This example produces an `InterruptedException` in function `a` (we can produce exceptions by throwing new exception elements of any type). Then, as `b` function calls `a` function, it is asked to either catch the exception or throw it. By adding the `throws` clause in the function definition, we explicitly say that this function can throw `InterruptedException` exceptions. This chain goes on with functions `c` and `d`. Finally, main function calls function `d`, and as this function can throw InterruptedExceptions, we need to catch the possible exception in *main*.

All this chain of exception throwing have been originated from `a` function, since it throws a *checked* exception that needs to be catched or thrown. If this function had thrown a runtime exception (such as `NullPointerException`), then none of the `throws` clauses would have been necessary, since it is a non checked exception. The example would have been like this:

```
public static void a(){
    throw new NullPointerException ("Null pointer exception in a");
}

public static void b(){
    a();
}

public static void c(){
    b();
}

public static void d(){
    c();
}

public static void main(String[] args){
    d();
}
```

However, if we try to run this last example, a `NullPointerException` exception will be produced in our console. As this is a non checked exception, we don't need to catch it but, as soon as it is produced, we should, to avoid these huge error messages in the console as we run the program:

```
Exception in thread "main" java.lang.NullPointerException:
Null pointer exception in a
    at Pruebas.a(Pruebas.java:6)
    at Pruebas.b(Pruebas.java:11)
    at Pruebas.c(Pruebas.java:16)
    at Pruebas.d(Pruebas.java:21)
    at Pruebas.main(Pruebas.java:26)
```

We can even throw (or declare to be thrown) as many exception types as we want, separated by commas in the `throws` clause. Then, we will need to catch all of them sooner or later:

```
public static void multipleExceptionsFunction()
throws IOException, InterruptedException {
    ...
    if (...){
        throw new IOException("IOException produced");
    }
    ...
    if (...){
        throw new InterruptedException("Interrupted!!");
    }
}

...

public static void anotherFunction(){
    try{
        multipleExceptionsFunction();
    } catch (IOException e1) {
        System.err.println(...);
    } catch (InterruptedException e2) {
        System.err.println(...);
    }
}
```

> **Exercise 1**:
>
> Create a program called **CalculateDensity** that asks the user to type a weight (in grams) and a volume (in liters). Then, the program must output the density, which is calculated by dividing weight / volume. The program must catch every type of possible exception: `NumberFormatException` and `ArithmeticException` whenever they can be thrown. You can only use `Scanner.nextLine` method to get the user input in this exercise.

> **Exercise 2**:
>
> Create a program called **WaitApp** with a function called *waitSeconds* that will receive a number of seconds (integer) as a parameter. Internally, this function will call `Thread.sleep` method to pause the program the given number of seconds (this function works with milliseconds, so you must convert seconds to milliseconds when calling it). As the `sleep` method can throw an `InterruptedException` element, you will need to deal with it. In this case, you are asked to throw the exception from *waitSeconds* method, and catch it in the *main* method, that will call *waitSeconds* with the number of seconds specified as a *main* parameter (inside the `String[] args` parameter). After waiting the specified number of secods, the program will prompt a "Finish" message before exiting.