by Nacho Iborra

Development Environments
Block 2
Unit 4: Functions and exception handling

2.4.1. Using functions

Functions let us arrange our code so that we can re-use a piece of code many times without having to duplicate the code. We just assign this piece of code a name (*function name*) and then we can *call* this code from other parts of our program. This paradigm is also called **modular programming**, since we define *modules* or functions to group our code in small subtasks, and call each subtask whenever we need it.

2.4.1.1. Function definition

If we want to use functions within a class, we can declare them as `public static` if we want to call them from anywhere. We will learn in next units how to declare other types of functions and how to set their visibility, but for now we are going to deal with public, static functions.

When defining a function, we need to specify the return type (or `void` if the function does not return anything), the function name and a pair of parentheses:

```
public static String myFunction()
{
    return "Hello";
}
```

We can call this function from any other function of the same class (including `main` function) by using the function name and the parentheses:

```
public static void main(String[] args)
{
    String result = myFunction();    // result = "Hello"
    System.out.println(result);
}
```

Note that functions always start in lower case in Java (in C#, they start in uppercase if they are public).

2.4.1.2. Function parameters

Some functions need some additional data to do their job. These data can be passed to the function as **parameters**, some kind of variables that are specified within the parentheses, including the type of each parameter and its name. For instance:

```
public static void myOtherFunction (int a, String b)
{
    ...
}

public static void main (String[] args)
{
    ...
    myOtherFunction(3, "Hello");
}
```

2.4.1.3. Parameters passed by value or by reference

In many languages we can choose among passing parameters by value or by reference. Regarding Java, every parameter of a primitive type is ALWAYS passed by value. In other words, if we change the value of the original parameter inside a function, this change will have no effect once the function is done. Regarding objects or complex data types (such as arrays), they are passed by reference. This means that we can change the properties of this parameter/object as long as we don't assign it to a new object or memory location.

```
public static void myFunction(int value)
{
    // This increment will have no effect after exiting the function
    value = value + 1;
}

public static void myOtherFunction(Integer value)
{
    // This assignment has no effect out of the function
    value = new Integer(10);
}

public static void myFunctionWithArray(int[] data)
{
    data[0] = 10;         // OK
    data = new int[10];  // has no effect after the function
}

public static void myFunctionWithObject(Person person)
{
    person = new Person(...); // No effect
    person.setName("Nacho");  // Name successfully changed
}
```

2.4.1.4. Parameters to *main* function

As you may have noticed, *main* function has a String array as parameter:

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

This means that we can pass as many arguments as we need to this *main* function from the command line. The first argument will be placed at index 0, the second argument at index 1 and so on.

```
public static void main(String[] args)
{
    if (args.length > 0)
        System.out.println("Received " + args.length + " args.");
}
```

> **Proposed exercises**:
>
> **2.4.1.1.** Create a program called *Palindrome* with a function called *isPalindrome*. This function will take a string as a parameter and return a boolean indicating if this string is a palindrome (this is, a string that can be read the same backward as forward, ignoring upper or lower case, and whitespaces). Test this function from the *main* function with the texts *Hannah*, *Too hot to hoot* and *Java is the best language* (this last text is NOT a palindrome).
>
> **2.4.1.2** Create a program called *CountOcurrences* with a function called *countString*. This function will take two strings *a* and *b*, and an integer *n* as parameters, and it will return a boolean indicating if the string *b* is contained at least *n* times in the main string *a*. Try it from the main function with the main string a = `This string is just a sample string`, the substring b = `string` and the number n = 2 (it should return `true`).
>
> **2.4.1.3** Let's try with recursion with these challenges from *Acepta el reto*:
>
> - Three fingers in each hand
> - Investing in Jaén

2.4.2. Exception handling

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**.

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.

2.4.2.1. Runtime error and 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.

2.4.2.2. 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.

```
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. 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).

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, as we have said before, 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.

2.4.2.3. 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). Let's have a look at this example:

```
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(...);
    }
}
```

Proposed exercises:

2.4.2.1. 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.

2.4.2.2. 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.