Functional programming

Using lambda expressions

Since Java 8, lambda expressions, also called *anonymous functions* in many languages, are an easy and fast way to implement an interface method without having to create a new class for doing that. In this document we are going to explain what they are, how to implement them and how to use them.

1. Functional interfaces and lambda expressions

In previous documents we have explained what **functional interfaces** are: they are interfaces with just one method to be implemented. In the Java API documentation, these interfaces are usually tagged with the annotation `@FunctionalInterface` to let the developer know that this interface is functional.

There are many useful examples of functional interfaces in Java API, such as:

- `Comparator`, that we can use to define a comparison method to sort a collection of objects
- `Predicate`, that can be used to filter a sublist of objects that meet some criteria
- `Consumer`, that can be used to process each object of a collection in a given way
- ...

Lambda expressions are an easy way to implement these interfaces, without having to define a new class, or even an anonymous class.

2. Defining lambda expressions

Lambda expressions can be seen as *single use* functions. We implement a functional interface at the point where we need to use it, and then this code is left aside. Let's see how to define these expressions with some examples.

2.1. A first example: java.io.FileFilter

`java.io.FileFilter` is a functional interface available in Java core API. It defines a method called `accept` so that we can define an acceptance criteria for files. In other words, when we implement this method, we must specify which files will be accepted from a list of files, according to one or many factors: file size, extension...

```
public interface FileFilter
{
    boolean accept(File file);
}
```

We are going to implement this interface to accept only Java files (i.e. files with `.java` extension). We will see how to do this in different Java versions, so that you can compare the evolution of these programming patterns.

Implementation before Java 7: normal class

Before Java 7, we had to create a new class for every new implementation we wanted to define. For example:

```
public class JavaFileFilter implements FileFilter
{
    @Override
    public boolean accept(File file)
    {
        return file.getName().endsWith(".java");
    }
}
```

And then use it like this:

```
File dir = new File("/home/arturo");
File[] javaFiles = dir.listFiles(new JavaFileFilter());
```

Implementation in Java 7: anonymous class

Besides using normal classes, since Java 7 we could create anonymous classes implementing interfaces or extending abstract classes, whose methods are implemented when we create the object (we could make different implementations every time).

```
File dir = new File("/home/arturo");
File[] javaFiles = dir.listFiles(new FileFilter()
{
    @Override
    public boolean accept(File file)
    {
        return file.getName().endsWith(".java");
    }
});
```

Implementation since Java 8: lambda expression

Besides normal and anonymous classes, since Java 8, when we are implementing a functional interface, we can do it with less code using a lambda expression:

```
File dir = new File("/home/arturo");
File[] javaFiles=dir.listFiles((File file) -> file.getName().endsWith(".java"));
```

We don't have to specify that it's a `FileFilter` interface what we are implementing because the compiler knows that the `listFiles()` method needs a `FileFilter` object as an argument. We don't need to use the `return` word either, because the compiler will assume it. We can even omit the parameter type because the compiler can look at it in the interface definition. So the lambda expression can be even more simple, like this:

```
File[] javaFiles = dir.listFiles(file -> file.getName().endsWith(".java"));
```

2.2. Another example: java.util.Comparator

`Comparator` interface from `java.util` package is another functional interface. It has only one method called `compare` that takes two objects as parameters, and compares them returning an integer that tells us which object comes first. Let's see how to implement this comparator to compare two `String` objects according to their length.

Implementation before Java 7

If we are using Java 6 or earlier versions, we need to define a normal class that implements the interface, and then create an object of this class and use it whenever we need to compare strings. For instance:

```
public class MyStringComparator implements Comparator<String>
{
    @Override
    public int compare(String s1, String s2)
    {
        return Integer.compare(s1.length(), s2.length());
    }
}

// MAIN
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");
MyStringComparator msc = new MyStringComparator();
...
Collections.sort(list, msc);
```

Implementation in Java 7

If we are using Java 7, we can also use an anonymous class to implement the interface, this way:

```
// MAIN
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");
Comparator<String> comp = new Comparator<String>()
{
    @Override
    public int compare(String s1, String s2)
    {
        return Integer.compare(s1.length(), s2.length());
    }
};
Collections.sort(list, comp);
```

Implementation in Java 8

Finally, if we are using Java 8 or later, we can also use a lambda expression. In this case, the method to be implemented has two parameters, so we define both in the parentheses of the lambda expression:

```
Comparator<String> lComp = (s1,s2) -> Integer.compare(s1.length(), s2.length());
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");
Collections.sort(list, lComp);
```

We can even shorten this code placing the lambda expression as the second parameter of `Collections.sort` method:

```
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");
Collections.sort(list, (s1,s2) -> Integer.compare(s1.length(), s2.length()));
```

2.3. One more example: java.lang.Runnable

You will use this interface when talking about threads and multithreaded programming. It has only one method to implement, with no parameters nor return type. It is used in `Thread` objects to define the method these threads will execute.

Implementation before Java 7

In versions earlier than Java 7, as usual, we need to define a class that implements the interface, and then use an object of this class:

```
public class MyRunnable implements Runnable
{
    @Override
    public void run()
    {
        for(int i = 0; i < 3; i++)
        {
            System.out.println("This is thread: " +
                Thread.currentThread().getName());
        }
    }
}

// MAIN
Runnable run = new MyRunnable()
Thread t = new Thread(run);
t.start();
```

Implementation in Java 7

In Java 7, we can define an anonymous class whenever we want to implement the interface:

```
// MAIN
Runnable run = new Runnable()
{
    @Override
    public void run()
    {
        for(int i = 0; i < 3; i++)
        {
            System.out.println("This is thread: " +
                Thread.currentThread().getName());
        }
    }
};
Thread t = new Thread(run);
t.start();
```

Implementation in Java 8

If we use Java 8 or later, we can also use a lambda expression. In this case, as the method has no parameters, we leave the parentheses of the lambda expression empty (we need to type the parentheses anyway):

```
Runnable lambdaRun = () -> {
    for(int i = 0; i < 3; i++)
    {
        System.out.println("This is thread: " +
            Thread.currentThread().getName());
    }
};
Thread t = new Thread(lambdaRun);
t.start();
```

Note that, in this example, our code needs more than one sentence, so we need to use the curly brackets { ... } after the arrow of the lambda expression. Again, we can also define the lambda expression in the parameter of `Thread` constructor:

```
Thread t = new Thread(() -> {
    for(int i = 0; i < 3; i++)
    {
        System.out.println("This is thread: " +
            Thread.currentThread().getName());
    }
});
t.start();
```

2.4. Conclusions

When we want to use a lambda expression, we only have to focus on the implemented method, and check:

- The input parameters of the method
- The value returned (if any)

Then, in the place of our application where we want to use the lambda expression, we define an object of the given interface, and define the lambda expression as follows:

- First, we specify the input parameters of the function, separated by commas, and between parentheses (unless there's only one parameter)
- Then we type an arrow (`->`). That's why these expressions are also called *arrow functions*.
- Finally, we write the code of the method. If it's just a simple `return` , we don't need to use curly braces `{ … }` . Otherwise, these curly braces are compulsory.

Remember the way we have created the lambda expression for `Comparator` (two parameters, a String returned):

```
Comparator<String> lComp = (s1,s2) -> Integer.compare(s1.length(), s2.length());
```

and the way we have created the lambda expression for `Runnable` (no parameters nor return type):

```
Runnable lambdaRun = () -> {
    for(int i = 0; i < 3; i++)
    {
        System.out.println("This is thread: " +
            Thread.currentThread().getName());
    }
};
```

Some more concepts we can take into account

There is a way to make a lambda expression even shorter. When it contains a method call that takes the same parameters as the lambda expression and in the same order, we can just write a reference to that method, omitting even the parameters, using this especial syntax:

```
// Normal version
Comparator<Integer> comp = (i1, i2) -> Integer.compare(i1, i2);
// Shorter version
Comparator<Integer> comp2 = Integer::compare;
```

We mentioned at the beginning of this section that lambda expressions are also called anonymous functions. If we use an anonymous class to implement an interface, then the compiler creates a whole object to deal with its methods. However, when working with lambda expressions, the compiler doesn't create full objects, but instead it creates a special and lighter type of object, so it's usually a better choice.

> **Exercise 1:**
>
> Create a project called **BookComparators**, with a `Book` class to store some information about books (title and price). Then, in the main program, create a list of books and use lambda expressions to sort and print the list by title (in alphabetical, ascending order) and then by price (in descending order).

> **Exercise 2:**
>
> Create a project called **ListFilter** that includes the following:
>
> - A class called `Student` that has these properties (getters/setters when necessary): name, age and a list of subjects (as strings).
> - A `Main` class with a main method and also a static method called:

```
List<Student> filterStudents(List<Student> srcList, Predicate<Student> predicate)
```

> The `filterStudents` method receives a student list and returns another list with only the items which meet the condition defined in the `Predicate`. A `Predicate` is a functional interface that needs to implement a method (`boolean test(T t)`). Implement it using lambda expressions.
>
> In the main method you'll have to create a list of at least 8 students, and then, using `filterStudents` method, generate 3 other lists that only hold students who:
>
> - Are older than 20.
> - Have the "Programming" subject.
> - His name contains "Peter".