Unit 1 extras

Some more examples to explain some difficult concepts of this unit

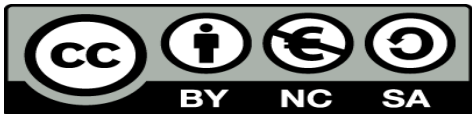

Table of Contents

Unit 1 extras

1. Introduction

In this first unit you have learnt some difficult concepts, specially if you have not read about them before. Some of them are more or less easy, if you have been programming with any object oriented language: abstract classes, interfaces... But some others are more difficult, specially for those of you who are used to "traditional" programming structures. Then, concepts like anonymous classes, lambda expressions, functional interfaces, streams... might be difficult to understand. These features have been added in the latest versions of Java (7 and 8), and are not present in many programming languages (yet).

We are going to see how to use some of these concepts with simple examples:

- From subsections 2 to 7 we are going to take a String value, convert it to upper case and add it a prefix. We could do this by using the String methods directly, but we are going to solve the problem in many different ways by using normal classes, abstract classes, interfaces, anonymous classes, lambda expressions and functions.
- In subsections 8 and 9 we are going to deal with a list of *Book* objects. We will learn how to implement an interface with an anonymous class, or with a lambda expression, to compare and sort books by title. Then, we will see how to manage trees, and how to process a stream (a new feature of Java 8), to filter books that match a given set of requirements.

2. Using a "normal" class

The first (and maybe the easiest way for you) to solve the problem is to create a "normal" class that takes a *String* in its constructor, the prefix to be added, and has a method that converts the string to uppercase and adds the prefix. This class could be like this one:

```
class ConvertString
{
    String text, prefix;

    public ConvertString(String text, String prefix)
    {
        this.text = text;
        this.prefix = prefix;
    }

    public String getConversion()
    {
        return prefix + text.toUpperCase();
    }
}
```

And we would use it in a main program this way:

```
public class MainClass
{
    public static void main(String[] args)
    {
        ConvertString conv = new ConvertString("Hello everyone",
            "(c) SPP 2015 - ");
        System.out.println(conv.getConversion());
    }
}
```

If we run the application, we will see this result in the console:

```
(c) SPP 2015 - HELLO EVERYONE
```

In the following subsections, we will solve the same exercise with some other strategies, such as using an abstract class, an interface, a lambda expression and functional interfaces.

3. Using an abstract class

If we want to solve previous problem with an abstract class, we would define a class with the same two attributes (the text to be converted and the prefix), the constructor and an abstract method that would perform the conversion.

```
abstract class ConvertStringAbstract
{
    String text, prefix;

    public ConvertStringAbstract(String text, String prefix)
    {
        this.text = text;
        this.prefix = prefix;
    }

    public abstract String getConversion();
}
```

Then, we would need to define a class that extends this abstract class and implements the abstract method, in the same way that we did with normal class in subsection 2.

```
class ConvertString extends ConvertStringAbstract
{
    public ConvertString(String text, String prefix)
    {
        super(text, prefix);
    }

    @Override
    public String getConversion()
    {
        return prefix + text.toUpperCase();
    }
}
```

Finally, we would instantiate an object of this "normal" class in the main application, and call the conversion method.

```
public class MainClass
{
    public static void main(String[] args)
    {
        ConvertStringAbstract c = new ConvertString("Hello everyone",
            "(c) SPP 2015 - ");
        System.out.println(c.getConversion());
    }
}
```

Obviously, using an abstract class to solve this exercise is not a good idea, because we are adding an additional, useless class (our abstract class) to the initial code. Abstract classes are useful when:

- We want to provide several ways of solving a problem (for instance, several ways of performing this string transformation).
- We have several subtypes of objects that implement a method that has no solution for the parent class. For instance, if we have a method called *talk* in a class *Animal*, we wouldn't know what to put on it until we know the concrete type of animal we are

instantiating. If it is a *Dog*, then we would say "*wof*", if it is a cat, we would say "*meoow*", and so on. So the *talk* method would be abstract in the *Animal* class, and we would implement it depending on the concrete type of animal, in the subclasses.

4. Using an interface

If we used an interface to solve this problem, then we would only need to define the *getConversion* method, but we would need to pass the text and prefix as parameters of this method.

```
interface ConvertStringInterface
{
    public String getConversion(String text, String prefix);
}
```

Then, we would implement this interface in a class:

```
class ConvertString implements ConvertStringInterface
{
    @Override
    public String getConversion(String text, String prefix)
    {
        return prefix + text.toUpperCase();
    }
}
```

And finally, we would instantiate this class in our main application:

```
public class MainClass
{
    public static void main(String[] args)
    {
        ConvertStringInterface conv = new ConvertString();
        System.out.println(conv.getConversion("Hello everyone",
           "(c) SPP 2015 - "));
    }
}
```

In this case, we are saving some code, since we don't need a constructor, and we don't use any attribute to store the information. We just pass the parameters to the method and get a result.

5. Using an anonymous class

Anonymous classes are very useful when we don't want to create a separate source file for a class that implements an interface, or extends an abstract class. In this case, we can just instantiate the abstract class or interface when we want to use it, and implement the abstract methods within the code block. So, to do this, we would only need the initial abstract class or interface that we want to implement (we are going the use the same interface of subsection 4):

```
interface ConvertStringInterface
{
    public String getConversion(String text, String prefix);
}
```

Then, in our main application, we instantiate an object of this interface with an anonymous class, and we implement the *getConversion* method right there:

```
public class MainClass
{
    public static void main(String[] args)
    {
        ConvertStringInterface conv = new ConvertStringInterface()
        {
            @Override
            public String getConversion(String text, String prefix)
            {
                return prefix + text.toUpperCase();
            }
        };
        System.out.println(conv.getConversion("Hello everyone",
            "(c) SPP 2015 - "));
    }
}
```

This code is equivalent to the one shown in subsection 4, but in this case we don't need to define the *ConvertString* class that implements the interface.

When to use normal classes or anonymous classes?

- We will use normal classes to implement an interface or extend an abstract class when we need to use them in more than one place of our code.
- Otherwise (i.e., when we only want to use an instance of an abstract class or interface, in a concrete place of our code), we can also use anonymous classes.

6. Using a lambda expression

Lambda expressions are useful when we want to implement a **functional interface**, this is, an interface with just one method, such as our interface of previous examples. This interface can be (optionally) marked with the annotation *@FunctionalInterface* to prevent the programmer from adding more methods to the interface:

```
@FunctionalInterface
interface ConvertStringInterface
{
    public String getConversion(String text, String prefix);
}
```

Then, instead of creating an anonymous class, or a "normal" class that implements this interface, we can use a **lambda expression**. To use them, we only have to focus on the implemented method, and check:

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

Then, in our main application, we define an object of the given interface with a lambda expression this way:

```
public class MainClass
{
    public static void main(String[] args)
    {
        ConvertStringInterface conv = (t, p) ->
        {
            return p + t.toUpperCase();
        };

        System.out.println(conv.getConversion("Hello everyone",
            "(c) SPP 2015 - "));
    }
}
```

The expression (t, p) represents the input parameters of the *getConversion* method (text and prefix, respectively). If there were no parameters, then we would only write the parentheses (). At the right of the arrow, there is the code of the *getConversion* method (*return p + t.toUpperCase()*). In this case, as we only have one instruction, we could even leave the lambda expression in just one line, this way:

```
        ConvertStringInterface conv = (t, p) -> return p + t.toUpperCase();
```

7. More about functional interfaces

We have just seen how to use lambda expressions to implement functional interfaces. In section *7.4 Interfaces* from Unit 1 you can learn more about functional interfaces, and the main subtypes included in package *java.util.function*. With these functions, we can define interfaces that can be implemented to produce a given result, or to take some parameters as input and produce a result as output.

For instance, we can use the interface **Function** to define a process that takes one input parameter and produces a result. We can also use the interface **BiFunction**, that takes two input parameters and produces a result. These functional interfaces have a method *apply* that has one/two input parameter(s) (respectively) and a result. We could implement this interface (in our case, *BiFunction*) with a lambda expression, and solve previous problem with it:

```
public class MainClass
{
    public static void main(String[] args)
    {
        // Takes 2 strings as parameters and produces a String as result
        BiFunction <String, String, String> cnv = (t, p) -> p + t.toUpperCase();
    }
}
```

Once we define the function, we can apply it to our input stream and get the output:

```
public class MainClass
{
    public static void main(String[] args)
    {
        // Takes 2 strings as parameters and produces a String as result
        BiFunction <String, String, String> cnv = (t, p) -> p + t.toUpperCase();
        System.out.println(cnv.apply("Hello everyone", "(c) SPP 2015 - "));
    }
}
```

We can also split the function in two functions: one that takes a String as input and converts it to uppercase:

```
Function <String, String> toUpper = (t) -> t.toUpperCase();
```

and another one that takes a String as input and adds a prefix to it:

```
BiFunction <String, String, String> prefix = (t, p) -> p + t;
```

With these two functions, we can define our desired function by joining them into a new one function:

```
BiFunction <String, String, String> cnv = prefix.andThen(toUpper);
```

Then, we would add these definitions in our main application, and use the last composed function to produce the desired result:

```
public class MainClass
{
    public static void main(String[] args)
    {
        // Takes a String and converts it to uppercase
        Function <String, String> toUpper = (t) -> t.toUpperCase();
        // Takes a String and a prefix and adds the prefix to the string
```

```
        BiFunction <String, String, String> prefix = (t, p) -> p + t;
        // Joining previous functions
        BiFunction <String, String, String> cnv = prefix.andThen(toUpper);
        // Using this last function to produce the result
        System.out.println(cnv.apply("Hello everyone", "(c) SPP 2015 - "));
    }
}
```

You can always link or join functions, as long as the output of the first one is of the same type (or subtype) than the input of the second one. In our case, the output of *prefix* function is a String, and the input parameter of *toUpper* function is also a String.

8. More about lambda expressions and anonymous classes

We have been solving a really simple example along previous sections. But we can use these features in any other problem. For instance, if we have a class called *Book* with a set of attributes, getters, setters and so on:

```
public class Book
{
	String title;
	String author;
	int pages;
	...
}
```

We can, for instance, define a *TreeSet* that orders a set of books by their title. To do this, we need to define an instance of *Comparator* interface. This interfaces gets two objects (books in our case) and returns which is greater or lower (alphabetically, by title). If it returns a positive number, then the first object is greater. If it returns a negative number, then the second object is greater. If it returns 0, then both objects are equal (alphabetically). We can do this with either an anonymous class:

```
Comparator<Book> comp = new Comparator<Book>()
{
    @Override
    public int compare(Book b1, Book b2)
    {
        return b1.getTitle().compareTo(b2.getTitle());
    }
};
```

or a lambda expression:

```
Comparator<Book> comp = (b1, b2) -> b1.getTitle().compareTo(b2.getTitle());
```

Then, we create a *TreeSet* with this comparator, and as we add books to it, they get automatically ordered:

```
TreeSet<Book> tree = new TreeSet<>(comp);
tree.add(new Book("Alice in Wonderland", "Lewis Carroll", 196));
tree.add(new Book("Hamlet", "William Shakespeare", 523));
tree.add(new Book("Ender's game", "Orson Scott Card", 213));
```

Finally, we can iterate over the *TreeSet* and show the books in order:

```
for (Book b: tree)
{
    System.out.println(b);
}
```

9. Lambda expressions & streams

Streams are a really powerful (and maybe hard to understand) concept of Java 8. They are useful when we want to filter and process a large list or amount of data.

Basic usage

For instance, we could have a list of book objects:

```
List<Book> books = new ArrayList<>();
books.add(new Book("Alice in Wonderland", "Lewis Carroll", 196));
books.add(new Book("Hamlet", "William Shakespeare", 623));
books.add(new Book("Ender's game", "Orson Scott Card", 213));
books.add(new Book("The Holy Bible", "Anonymous", 596));
...
```

Then, we get a *Stream* object from this list:

```
Stream<Book> stream = books.stream();
```

And we can apply some filters to the list, and get some partial lists as a result. For instance, if we want to get a list of all the books with more than 500 pages, and print them, we could apply this filter (intermediary operation):

```
Stream<Book> moreThan500 = stream.filter(b -> b.getPages() > 500);
```

and then use this final operation to print them:

```
moreThan500.forEach(System.out::println);
```

Combining filters

We can combine some filters by joining them with the "." operator. For instance, we can calculate the average number of pages for those books with more than 500 pages:

```
OptionalDouble avg = stream.filter(b -> b.getPages() > 500)
                           .mapToInt(b -> b.getPages()).average();
```

We use the *OptionalDouble* object because the stream might be empty (if there are no books with more than 500 pages). Then, we check if this object has a value or not, and print the corresponding result message:

```
System.out.println(avg.isPresent()?avg.getAsDouble():"No matches");
```

Reloading the stream

Whenever we filter anything from a stream, we need to get the stream again from the original collection if we want to apply new filters apart from the previous ones. In our previous example, we would need to get the stream again before calculating the average. The whole process would be like this:

```
Stream<Book> stream = books.stream();
Stream<Book> moreThan500 = stream.filter(b -> b.getPages() > 500);
moreThan500.forEach(System.out::println);
stream = books.stream();
OptionalDouble avg = stream.filter(b -> b.getPages() > 500)
                           .mapToInt(b -> b.getPages()).average();
System.out.println(avg.isPresent()?avg.getAsDouble():"No matches");
```

Mapping streams

When we map a stream, we focus on a single attribute of the objects of the stream. For instance, with this filter and mapping we print just the titles of the books with more than 500 pages:

```
books.stream().filter(b -> b.getPages() > 500)
              .map(b -> b.getTitle())
              .forEach(System.out::println);
```

Collecting streams

The collecting operations can either return a collection which is a subset of the original one. For instance, we can return a list containing only the books with more than 500 pages:

```
List<Book> listMoreThan500 =
        books.stream().filter(b -> b.getPages() > 500)
                      .collect(Collectors.toList());
```

We can also return, for instance, a list of the titles of these books, separated by commas:

```
String resultMoreThan500 =
        books.stream().filter(b -> b.getPages() > 500)
                      .map(b -> b.getTitle())
                      .collect(Collectors.joining(", "));
```