Unit 1f: Functional Thinking

1 Introduction

Functional programming requires you to think about programming tasks in quite a different way from imperative programming. It is not necessarily a “better” way of thinking, but provides a more efficient solution to some common problems.

A functional language (or one that supports functional programming) provides you with the tools to design and implement programs based on functional thinking.

Here are two important general principles to keep in mind:

- focus on results, not steps (be declarative, not imperative);
- offload mundane details to programming languages; focus on the unique aspects of your programming problems.

(http://nealford.com/functionalthinking.html)

2 Motivating examples

Before we dive into the details, here are two simple examples of **imperative** and **functional** approaches to solving the same problem. We are interested in the approach and *functional thinking*, not specifically in the language. The functional approach is shown with both Java and Scala. Java has some support for functional techniques, particularly in Java 8. You will learn more soon about how the functional examples work; later you will learn about Scala’s extensive support for further functional programming techniques.

2.1 Factorial

Calculate the *factorial* of a number, for example, the factorial of 4 is:

4! = 1 * 2 * 3 * 4 = 24

2.1.1 *Imperative (Java)*

```
public int factorial(int n)
{
    int result= 1;
    for (int i=2; i<=n; i++)
    {
       result = result * i;
    }
    return result;
}
```

Note the step-by-step instructions, variables *i* and *result* change to keep track of count and value after each iteration.

2.1.2 *Functional (Java)*

```
public int factorial(int n)
{
    if(n==0 || n==1)
        return 1;
    return factorial(n-1) * n;
}
```

2.1.3 *Functional (Scala)*

```
def factorial(n: Int): Int = n match {
  case 0 => 1
  case _ => factorial(n-1) * n
}
```

With the functional approach, you just evaluate expressions ("focus on results…"), making use of *recursion*, where an expression includes a call to the function that it is inside, widely used in Functional Programming.

2.2 Check for upper case

We want to check whether a string contains any upper class characters e.g. return true for "FishAndChips", false for "fishandchips".

2.2.1 *Imperative (Java)*

```
boolean nameHasUpperCase = false;

for (int i = 0; i < name.length(); ++i)
{
    if (Character.isUpperCase(name.charAt(i)))
    {
        nameHasUpperCase = true;
        break;
        }
    }
```

The algorithm explicitly combines iteration and selection – *if* statement condition is the rule we want to apply.

2.2.2 Functional (Java 8)

```
    boolean nameHasUpperCase =
          name.
          chars().
          anyMatch(c -> Character.isUpperCase(c));
```

`c -> Character.isUpperCase(c)`is a Lambda expression. String in Java has *chars* method that returns a **stream** of characters, stream has *anyMatch* method that is equivalent to *exists* in Scala.

Java SE 8 streams allow you to work with collections in a functional style. Both the existing Java notion of collections and the new notion of streams provide interfaces to a sequence of elements.

So what's the difference? In a nutshell, collections are about *data* and streams are about *computations*. Streams provide aggregate functions such as *anyMatch* and *filter*, and allow these to be chained in a pipeline. Streams can make use of multi-core architectures without the need for explicit multithreaded code – important benefit of functional style.

www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

2.2.3 Functional (Scala)

```
    val nameHasUpperCase = name.exists(x => x.isUpper)
```

`x=>x.isUpper` is a Lambda expression; note => instead of ->. The *exists* method of *string* applies the lambda to each character in string and returns true if the lambda evaluates to **Boolean** for any character.

We can use a Scala shorthand notation that works for the case of one bound variable:

```
    val nameHasUpperCase = name.exists(_.isUpper)
```

Note the composition of functions – *exists* and the function defined by lambda (which itself calls *isUpper* method of *character*).

We use the method that is provided by the programming language to take care of iteration and applying the condition/rule that we specify - "offloading mundane details…"

Note, these examples show features of different *programming styles*, not of the languages chosen: you can write imperative style code in Scala just as you can in Java; you can use recursion in Java just as you can in Scala. A new streams API was introduced to Java in Java 8 which is illustrated in the second example (the *anyMatch* method); Scala had similar features built in from the start, illustrated by the use of the *exists* method of a Scala string.

3 General Principles

The general principles are:

- Focus on results, not steps (be declarative, not imperative)
- Offload mundane details to programming languages, focus on the unique aspects of your programming problems

It is often possible to write the same computation in imperative or functional styles.

Functions map inputs to outputs; you always get the same output for the same input. Functions can be composed.

The Functional style favours *no side effects* (immutability) and *no shared state*.

Recursion is commonly used to perform iterative computations in functional programming.