Advanced Functional Thinking

Dr Surbhi Saraswat
School of Computer Science

Referential transparency

- In Maths, referential transparency is the property of expressions that it can be replaced by other expressions having the same value without changing the result in any way.
- Consider the following example:
  - X = 2 + (3 * 4),
  - we can replace the subexpression (3 * 4) with any other expression having the same value without changing the final result.
- In programming, referential transparency applies to programs. As programs are composed of subprograms, which are programs themselves, it applies to those subprograms, too.
- Subprograms or the functions can be referentially transparent, if a call to this function may be replaced by its return value.

Referential transparency

- In this example, the **mult** method is referentially transparent because any call to it may be replaced with the corresponding return value. This may be observed by replacing **mult(3,4)** with **12**.
- In the same way, **add(2,12)** may be replaced with the corresponding return value, **14.**
- None of these replacements will change the result of the program, whatever it does.

```
int add(int a, int b) {
    return a + b
}

int mult(int a, int b) {
    return a * b;
}

int x = add(2, mult(3, 4));
```

Benefits of referential transparency:

- It makes reasoning about programs easier.
- It also makes each subprogram independent, which greatly simplifies unit testing and refactoring.
- As an additional benefit, referentially transparent programs are easier to read and understand, which is one reason for why functional programs need less comments.

Referential transparency

Scala Function - Test referential transparency

The first line of the below code defines a free variable. The next four lines define a function named *testRT*. The last two lines make a call to the *testRT* with a constant input.

```
var g = 10
def testRT(i: Int): Int = {
 g = i + g;
 return g
}

val v1 = testRT(5)
//Output:- v1: Int = 15
val v2 = testRT(5)
//Output:- v2: Int = 20
```

Can I replace all references of *testRT(5)* with *15* or *20*?

Your answer would be an obvious *No*.

So, *testRT* does not qualify for referential transparency.

It is not a pure function:

- The function testRT violates the first principle. Its output depends upon an external variable g.
- The testRT qualifies for the second rule because it does not modify the input parameter.
- It is not eligible to qualify the third law. The testRT has a side effect because it changes the state of an external variable.

First-class and higher-order functions

First-Class Function: A programming language is said to have **First-class functions** if functions in that language are treated like other variables. So the functions can be assigned to any other variable or passed as an argument or can be returned by another function.

Higher-Order Function: A function that receives another function as an argument or that returns a new function or both is called a Higher-order function.

- Higher-order functions are only possible because of the First-class function.
- Functions are considered higher-order functions when they **take functions as arguments,** (like most Array utilities, .map, .filter, .reduce, .every) and/or **return a function as a result**
- **Abstraction is the main benefit of Higher Order functions.**

First-class and higher-order functions

First-Class Function:

A programming language is said to have **First-class functions** if functions in that language are treated like other variables. So, the functions can be assigned to any other variable or passed as an argument or can be returned by another function.

Assigned to regular variables

```
def doubler(i: Int) = i * 2

var d = doubler _

d(5)

//Output:- res0: Int = 10
```

First-class and higher-order functions

Higher-Order Function:

A function that receives another function as an argument or that returns a new function or both is called a Higher-order function.

Passed as arguments to functions

```
def doubler(i: Int) = i * 2
def tripler(i: Int) = i * 3
def applyF(f: Int => Int, x: Int) = f(x)
applyF(doubler, 5)
//Output:- res0: Int = 10
applyF(tripler, 5)
//Output:- res1: Int = 15
```

- First & second line of code defines two Scala functions.
- The third line defines **a higher order function**.
- The function *applyF* takes two arguments.
  - First argument type is a function and the second argument is an integer.
  - The next line passes *doubler* function to *applyF*.
  - Similarly, the last line also passes a *tripler* function to applyF.

First-class and higher-order functions

Returned as results of functions

```
def getOps(c: Int) = {
    def doubler(x: Int) = x * 2
    def tripler(x: Int) = x * 3
    if (c > 0)
        doubler _
    else
        tripler _
 }
val d = getOps(1)
d(5)     //Output:- res6: Int = 10
val d = getOps(-1)
d(5)    //Output:- res6: Int = 15
```

- The code defines a function *getOps*.
- The first line inside the body of the *getOps* defines a local function *doubler*.
- The second line defines another local function *tripler*.
- Finally, the if expression returns an appropriate local function depending on the value of *c*.
- Later, we call the *getOps* function and assign the returned value to *d*.
- The variable *d* holds a function that we call as the last line. We have seen both types of Higher Order functions.