Problems returning values in pure functions and the use of the return keyword

During the first exercise session, some students were stumped that their function always returned the value of the last expression or Unit (since they only used if-statements or omitted the last else). This does not come as a surprise, as some of you are used to solving problems like that in imperative programming.

To illustrate the problem, let's start with a simple function that returns a String describing the integer we provide as an argument.

```
def foo(n: Int): String =
    if(n == 0) "Equals to zero"
    else if (n < 0) "Lower than zero"
    else "Higher than zero"
```

So far, so good. If we evaluate the following expression, we get:

```
scala> List(-5,0,99).map(foo)
val res0: List[String] = List(Lower than zero, Equals to zero, Higher than zero)
```

What some students did, however, is:

- Only using `if`-statements;
- Using two `if`-statements and one default case to be returned;
- Using one `if` and `else if`-statement and a default case to be returned.

In most cases (when one did not use the return keyword), this led to unexpected results.

Why is that? Well, let's look at an example.

```
def foo2(n: Int): String =
    if(n == 0) "Equals to zero"
    else if (n < 0) "Lower than zero"
    "Higher than zero"
```

```
scala> List(-5,0,99).map(foo2)
val res1: List[String] = List(Higher than zero, Higher than zero, Higher than zero)
```

Strange! Well, We can easily explain the problem. Scala will always return the value of the last expression in a code block. In foo, there is only one expression in the code block; the conditional (yellow in the listing below). Each branch in that conditional has only one expression (here, a constant).

```
def foo(n: Int): String =
    if(n == 0) "Equals to zero"
    else if (n < 0) "Lower than zero"
    else "Higher than zero"
```

In foo2, however, the code block that constitutes the function's body contains two expressions: a conditional and a constant. These two expressions are highlighted in yellow and blue in the listing below. Let's say we apply foo2 to the number -5. Scala will enter the second branch of the conditional and evaluate the expression of that constant, which returns itself. But, there are other expressions in the code block that Scala will evaluate. The result of the conditional is "ignored."

```
def foo2(n: Int): String =
    if(n == 0) "Equals to zero"
    else if (n < 0) "Lower than zero"
    "Higher than zero"
```

When you write functions in Scala, Scala will sometimes "notice" something is wrong and emit warnings (not errors) to the developer. These warnings may state something along the lines of: "A pure expression does nothing in statement position." This warning is a pretty good hint that something is (or will be going wrong); "you are programming imperatively (statement) with a pure expression."

How can we solve this? Well,

- You either ensure that you organize your code blocks property, e.g., by using one conditional, one case, … as we've done in foo.
- Or you explicitly use the `return` keyword.
-

```
def foo3(n: Int): String =
    if(n == 0) return "Equals to zero"
    else if (n < 0) return "Lower than zero"
    return "Higher than zero"

def foo4(n: Int): String =
    if(n == 0) return "Equals to zero"
    else if (n < 0) return "Lower than zero"
    "Higher than zero"
```

```
scala> List(-5,0,99).map(foo3)
val res2: List[String] = List(Lower than zero, Equals to zero, Higher than zero)

scala> List(-5,0,99).map(foo4)
val res3: List[String] = List(Lower than zero, Equals to zero, Higher than zero)
```

The functions foo3 and foo4 are equivalent as we can omit the last `return` occurrence (in this example).

Is the problem solved in this case? Yes. But according to functional programming principles, the outcomes are not that elegant. In this case, we can tolerate the use of return. We have not yet covered this in class, but using `return` can be problematic.

Important sidenote not yet covered in class

While the use of `return` in the example above is OK, the use of `return`, in general, can be problematic. Its use can break referential transparency in some specific cases. Referential transparency is about substituting expressions with the value they yield. But `return` does not evaluate anything. Instead, this keyword “tells” Scala to evaluate the expression next to it and exit the function while returning the result. Similarly to expression handling, you manipulate the control flow of your program.

We have seen in class that you can apply functional programming in any programming language that supports methods—however, some languages “force” us to use the `return` keyword. *Scala has been designed for functional programming*, and we can write functions that do not need to rely on `return` *statements*. **So, if you have return statements in the pure part of your code, consider eliminating those.**