Blocks

In this lesson, you will be introduced to blocks and learn how to set the scope of a program.

We'll cover the following

- Nested Functions
- Visibility

In our program for computing the square root of a number, we had to define multiple functions. You might have noticed that most of the functions were very specific to our program and might not be that useful anywhere else. We can better organize our program using a **block**.

Before we get started, let’s take a look at the code we have written so far.

```
def abs(x: Double) =
  if (x < 0) -x else x

def isGoodEnough(guess: Double, x: Double) =
  abs(guess * guess - x) / x < 0.0001

def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)

def sqrt(x: Double) = sqrtIter(1.0, x)
```

Nested Functions

Did you know that you’ve been using blocks this whole time? Blocks are a sequence of expressions wrapped in curly brackets `{}` which are themselves expressions. The last element of a block is an expression that defines its value.

Blocks allow us to create **nested functions**, a functionality not to be found in many programming languages. Nested functions are functions defined within another function.

Let's put the implementation of `sqrt` in a block and see how nested functions would be written.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
def abs(x: Double) =
  if (x < 0) -x else x

def sqrt(x: Double) ={
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def isGoodEnough(guess: Double, x: Double) =
    abs(guess * guess - x) / x < 0.0001

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  sqrtIter(1.0, x)
}

println(sqrt(4))
```

The great thing about blocks is that you don't have to worry about function interdependency. You might have noticed that we defined `sqrtIter` before `isGoodEnough` and `improve` even though they are both being used in its implementation. Blocks combine all the expressions in it into one unit, hence, when the compiler runs into a function which is dependent on a function which hasn't been defined yet, it will go through the rest of the expressions in the block until it finds the function.

> Just remember, the final value or result of the block must be the last expression.

Visibility

When it comes to what is and isn't visible in a block there are two rules.

Rule #1: When you define something inside a block, it is only visible from within the block.

Rule #2: The definitions inside a block, shadow definitions of the same names

Rule #2: The definitions inside a block, shadow definitions of the same names outside the block.

For instance, if I define a variable outside of a block, I can define a variable with the same name inside the block as well without it affecting the outside variable. The outer variable will be shadowed by the inner. Let's look at an example below.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
val amIVisible = 0
def square(x: Int) =
  x * x

val result = {
  val amIVisible = square(3)
  println(s"Variable Inside Block: $amIVisible")
}

println(s"Variable Outside Block: $amIVisible")
```

While we are printing a variable with the same name, the variables are actually different, hence, we get two different values in the output. However, as `square` is not being defined inside the block it is not shadowing the `square` outside the block.

Let's look at another example similar to the one we looked at before. In this example, we are printing `result` rather than the variable.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
val amIVisible = 1

def plusOne(x: Int) =
  x + 1

val result = {
  val amIVisible = plusOne(3)
  amIVisible * amIVisible
}

println(result)
```

When you press run, the output you should get is as expected...**16**.

Now, let's slightly modify this example.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
val amIVisible = 1

def plusOne(x: Int) =
  x + 1

val result = {
  val amIVisible = plusOne(3)
  amIVisible * amIVisible
} + amIVisible

println(result)
```

When we run the code above, we are getting **17** as our output. This is because `amIVisible` on **line 9** is referring to the one outside the block. Hence, when we add `amIVisible` to the `result` we get **16** + **1** = **17**.

In the next lesson, you will apply the concepts you learned in this lesson and write your own nested function.