Loop with while

In this lesson, you will be introduced to the while loop and learn how it is different from the other control structures.

We'll cover the following

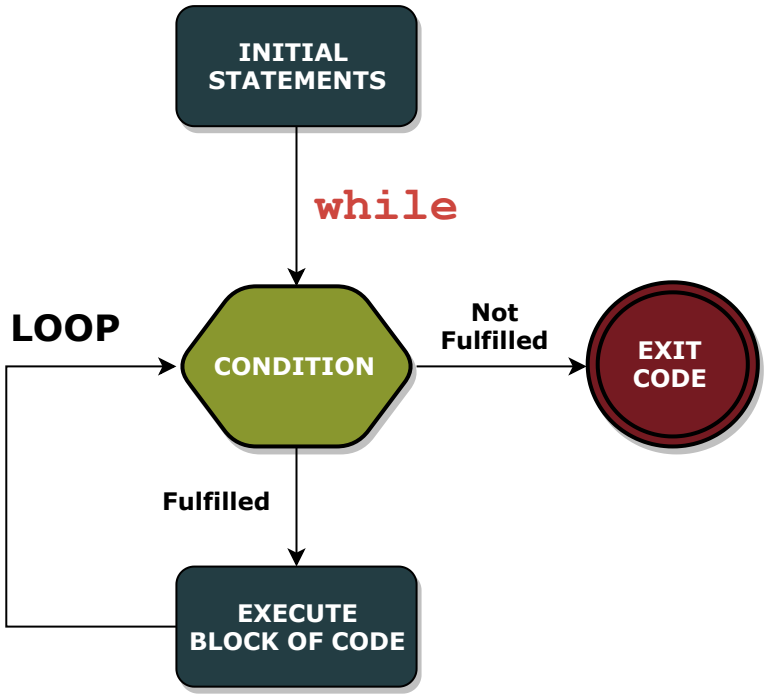
Introduction

When using the word **“while”** in a sentence, we relate it to the construct of time. In English, **“while”** can be used as a conjunction, indicating the occurrence of one event *while* another one is in place. For instance, you can say “I am reading a book while my sister is getting ready”. This sentence is indicating that the event, **reading a book**, will continue to take place as long as the event **sister is getting ready** is in place.

Keeping the above concept in mind, a `while` loop in Scala behaves the same way. One event is a condition and the other event is a body of code to be executed. The body of code will be executed repeatedly again and again as long as the condition is being fulfilled (holds true).

Control Flow

Let’s look at the control flow of a `while` loop below.



The above flow is showing that if the condition has been fulfilled, the compiler will execute the block of code and check the condition again. This loop will occur until the condition is not fulfilled in which case the compiler will exit the block of code.

Syntax

Before we see the `while` loop in action, let's go over the syntax and see how to write a block of code with `while` using Scala.

```
while (condition) {
  block of code}
```

The syntax is pretty straight forward with a `while` followed by the condition to be checked in `()` which is further followed by curly brackets `{}` in which appears the block of code to be executed if the condition holds `true`.

> The condition must be an expression of type `Boolean`. If the condition is `true`, the code will execute. If the condition is `false`, the compiler will exit the code.

`while` in Action

In the example below, we want to print the numbers from **1** to **10** using a `while` loop.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
var count = 1
while (count <= 10) {
  println(count)
  count += 1
}
```

The variable `count` is acting as our loop control variable. It is initialized with the value **1** and while its value remains less than or equal to **10** the block of code in `{}` will keep executing. The block of code is simply printing the current value of `count` on **line 3** and incrementing it by one on **line 4**.

`do-while`

Scala also has a `do-while` loop which works exactly like the `while` loop with one added difference that it executes the block of code before checking the condition. `do-while` ensures that the code in the curly brackets `{}` will execute at least once.

Syntax

The syntax is as follows.

```
do {
    block of code
   } while (condition)
```

Let's look at a very simple example to understand the subtle difference between `while` and `do-while`.

The code below declares an immutable variable `alwaysOne` which is assigned a value of **1**. We want to print the value of `alwaysOne` when it is not **1** (what a

paradox!)

As the condition of `a != 1` can never be `true`, the value of `a` should never be printed. But that's not the case. Let's run the same code using `while` and `do-while` and see what happens.

While the `while` loop doesn't execute the block of code, the `do-while` loop does as it checks the condition after block execution.

Loops not Expressions

You might have noticed that we have been using the term **while loop** instead of **while expression**. This is contradictory to the term **if expression** we have been using throughout the several previous lessons. So, why is it that we haven't been calling `while` and `do-while` *expressions*.

This is because *expressions* have return values and `while` and `do-while` don't result in any significant return value. Both loops have a return type of `unit` which, as mentioned in an earlier chapter, is just a *void* value; empty.

While pure functional programming languages have expressions and not loops, Scala still gives you the option of using `while` and `do-while` for imperative programming. However, like `var`, it is preferable for one to challenge themselves to not use `while` loops.

In the next lesson, we have a challenge for you to solve for yourself.