Break and Continue

This lesson discusses the break and continue construct in detail.

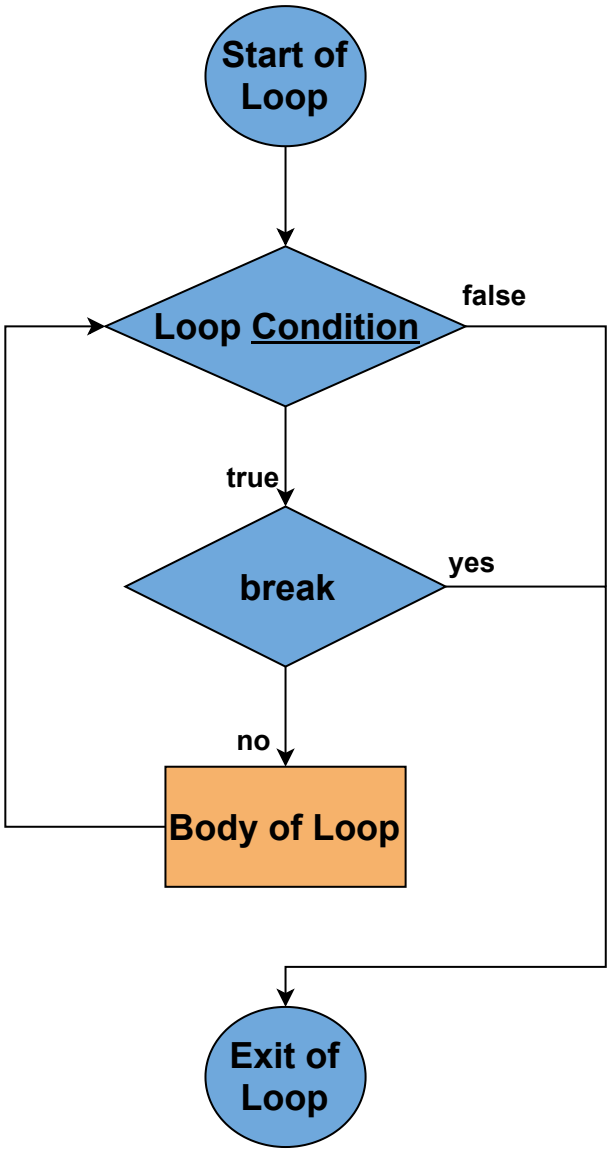
WE'LL COVER THE FOLLOWING

Introduction

Sometimes, we may want to skip the execution of a loop for a certain condition or terminate it immediately without checking the condition. To specifically change the flow of execution, we have another two statements, `break` and `continue`.

`break` statement

In every iteration, a condition has to be checked to see whether the loop should stop. If the exit-condition becomes *true*, the loop is left through the `break` statement. A `break` statement always breaks out of the innermost structure in which it occurs; it can be used in any for-loop (counter, condition, and so on), but also in a `switch`, or a `select` statement. Execution is continued after the ending **}** of that structure. The following figure explains the `break` statement.



Break Statement

Run the following program to understand how `break` works.

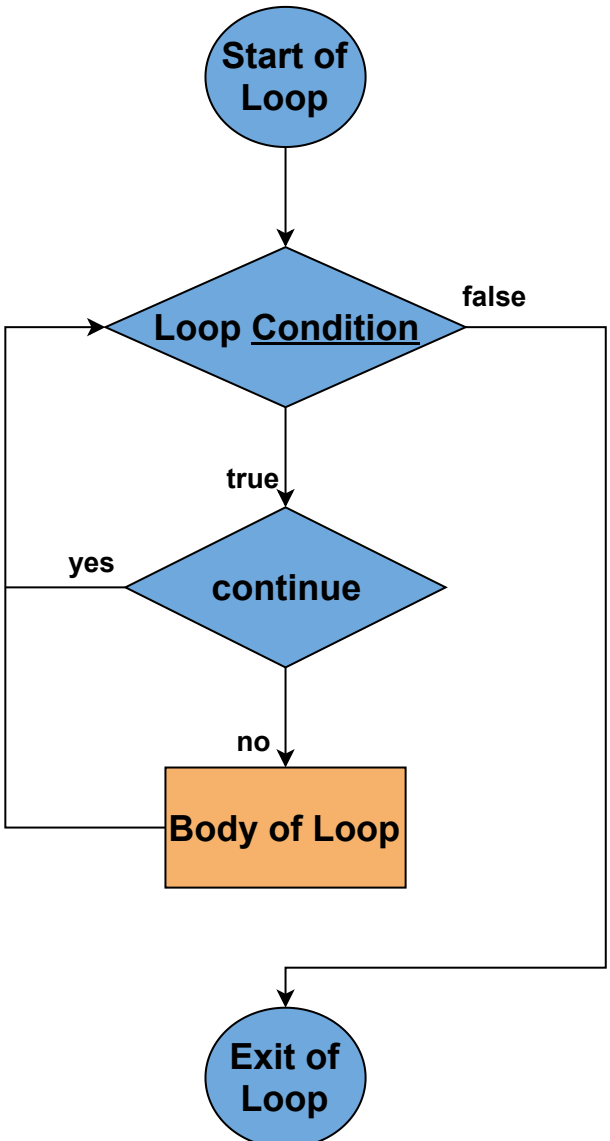
```
package main
import "fmt"

func main() {
    for i:=0; i<3; i++ {    // outer loop
        for j:=0; j<10; j++ {   // inner loop
            if j>5 {
                break                   // breaking inner loop
            }
            fmt.Printf("%d",j)
        }
        fmt.Printf(" ")
    }
}
```

Break statement in Nested Loop

The program implements the `break` statement within the *nested loop*. At **line 5**, we made an outer counter-controlled loop: `for i:=0; i<3; i++` which will iterate *three* times. Then at **line 6**, we made an inner counter-controlled loop: `for j:=0; j<10; j++` that will run **10** times for each `i`. You may have noticed that we added a `break` statement at **line 8** for a condition `j>5` at **line** 7. It means that the inner loop will always break when `j` is greater than **5**. Control will transfer to the outer loop. That's why the output is **012345 012345 012345**. The pattern **012345** is printed *thrice* because the outer loop runs three times only. Additionally, numbers ranging from **0** to **5** inclusively are printed because the inner loop breaks after **5**.

`continue` statement

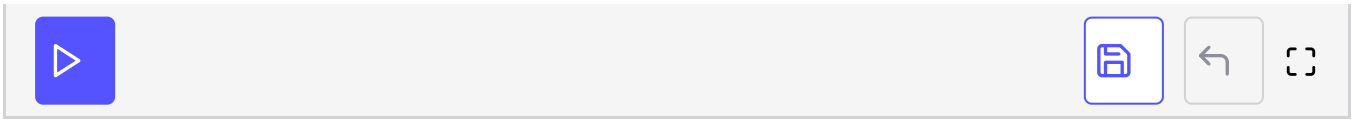
The keyword `continue` skips the remaining part of the loop but then continues with the next iteration of the loop after checking the condition. The following is a figure that explains the `continue` statement.



Continue Statement

Run the following program to understand how `continue` works.

```
package main
import "fmt"

func main() {
    for i := 0; i < 10; i++ {     // for loop
        if i == 5 {
            continue                        // continuing the loop
        }
        fmt.Printf("%d",i)
        fmt.Print(" ")
    }
}
```



Continue Statement in For Loop

This program is an implementation of the `continue` statement within a loop. At **line 5**, we made a counter-controlled loop: `for i:=0; i<10; i++` which will iterate **10** times. You may have noticed that we added a `continue` statement at **line** 7 for a condition `i==5` at **line 6**. This means that the loop will iterate again for a new `i`, leaving further statements in that loop when `i` is equal to **5**. That's why the output is **0 1 2 3 4 6 7 8 9**. Numbers up to **9** are printed because the loop runs ten times only. And the number **5** is missing because the iteration starts again leaving **line 9** and **line 10** unexecuted.

Remember that the keyword `continue` can only be used within a *for-loop*.

That's it about how control is transferred using the break and continue construct. The next lesson describes the use of *labels* in Go to control the flow of execution.