The for Construct

This lesson discusses the for construct in detail.

WE'LL COVER THE FOLLOWING

Introduction

So far, we have studied two types of constructs; the `if-else` and the `switch` construct. Another very important and widely used control structure in programming is the `for` construct.

In Go, the `for` statement exists to repeat a body of statements a number of times. *One* pass through the body is called an *iteration*.

> **Remark:** There is no `for` match for the `do-while` statement found in most other languages. It was probably excluded because the use case for it was not that important.

Types of `for` loops

There are *two* methods to control iteration:

- *Counter-controlled* iteration
- *Condition-controlled* iteration

Let's study them one by one.

Counter-controlled iteration

Counter-controlled iteration

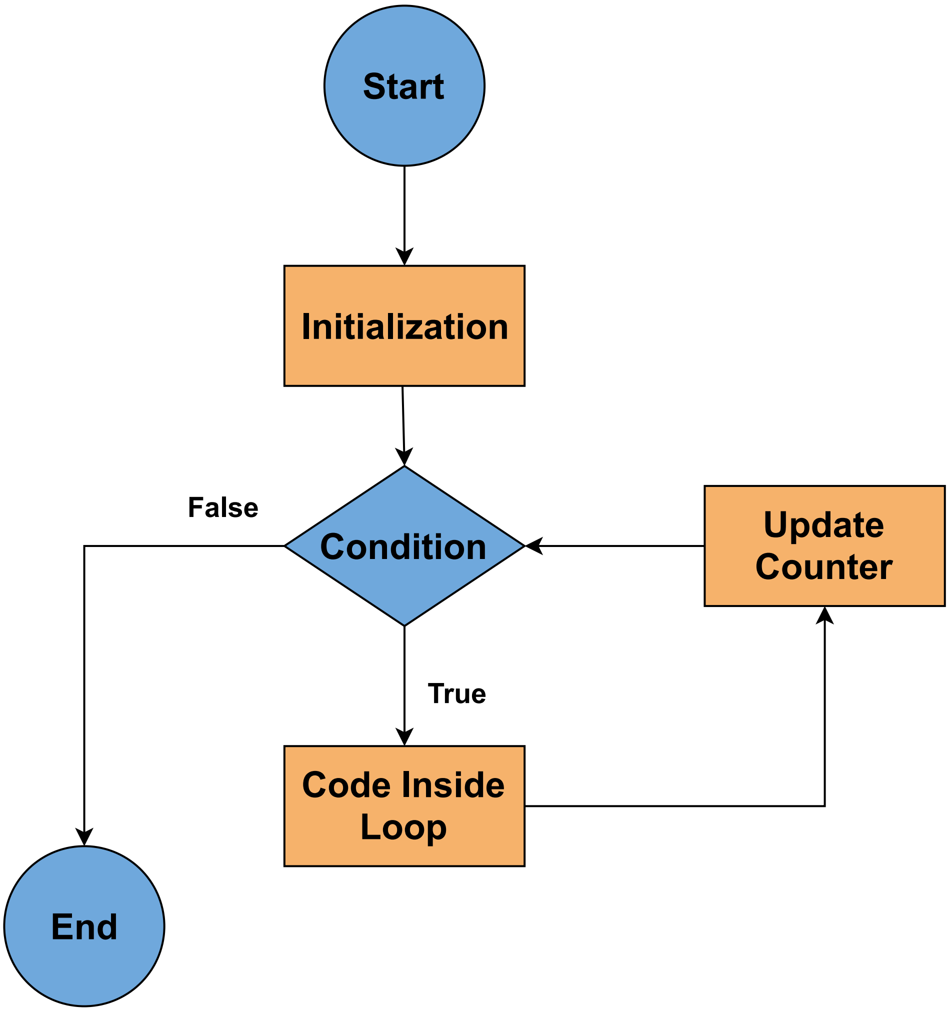
The simplest form is the counter-controlled iteration. The general format is:

```
for initialization; condition; modification { }
```

For example:

```
for i := 0; i < 10; i++ {}
```

The body **{ }** of the for-loop is repeated a known number of times; this is counted by a variable `i`. The loop starts with an *initialization* for `i` as: `i := 0` (this is performed only once). This is shorter than a declaration beforehand, and it is followed by a *conditional check* on `i`: `i < 10`, which is performed before every iteration. When the condition is *true*, the iteration is done. Otherwise, the for-loop stops when the condition becomes *false*. Then comes a *modification* of `i`: `i++`, which is performed after every iteration, at which point the condition is checked again to see if the loop can continue. This modification could, for example, also be a **decrement**, or + or –, using a step. The following is a figure that explains the `for` construct.



For Construct

Run the following program and see how a *for* loop is implemented.

```
package main
import "fmt"

func main() {
    // for loop with 5 iterations
    for i := 0; i < 5; i++ {
        fmt.Printf("This is the %d iteration\n", i)
    }
}
```

Counting Iterations in a For Loop

As you can see, we initialize `i:=0` for a for loop at **line 6**. We set the condition as `i<5`, which means the loop will run for **5** times (as there are 5 numbers from 0 to 4 inclusively). As a modification, we set `i++`. After every iteration, `i` will be incremented by 1. Inside for loop's body (at **line** 7), we are just printing the iteration number, which is the value of `i` for a certain iteration. According to the output, you can see that there are *5* iterations in total, ranging from **0** to **4** iterations.

These are 3 separate statements which form the header of the loop. They are separated by ;, but there are no **( )** surrounding the header:

```
for (i = 0; i < 10; i++) { } //is invalid Go-code!
```

Again the opening **{** has to be on the same line as the for keyword. The counter-variable ceases to exist after the **}** of the for; always use short names for it like `i`, `j`, `z`, or `x`. Never change the counter-variable in for loop itself; this is considered bad practice.

More than 1 counter can also be used as in:

```
for i, j := 0, N; i < j; i, j = i+1, j-1 { }
```

This method is often the preferred way in Go as we can use **parallel assignment**. Here, we have two counters `i` and `j`. Counter `i` is incrementing by 1, and counter `j` is decrementing by 1.

For loops can be nested, like this:

For-loops can be nested, like this:

```
for i:=0; i<5; i++ {
    for j:=0; j<10; j++ {
        println(j)
    }
}
```

Here, for every iteration of `i` , `j` will run *10* times. This means, that the outer loop: `for i:=0; i<5; i++` will run **5** times and inner loop : `for j:=0; j<10; j++` will run **50** times in total.

What happens if we use for-loop for a general Unicode-string? Let’s run a program and find out.

```
package main
import "fmt"

func main() {
    str := "Go is a beautiful language!"
    fmt.Printf("The length of str is: %d\n", len(str))

    // for loop to find character at each position
    for ix :=0; ix < len(str); ix++ {
        fmt.Printf("Character on position %d is: %c \n", ix, str[ix])
    }
    str2 := "Chinese: 日本語"
    fmt.Printf("The length of str2 is: %d\n", len(str2))

    // for loop to find character at each position
    for ix :=0; ix < len(str2); ix++ {
        fmt.Printf("Character on position %d is: %c \n", ix, str2[ix])
    }
}
```

Unicode String with For

As you can see at a **line 5**, we declare string `str` and initialize it with **“Go is a beautiful language!”**. Then, at the next line, we find the length of `str` and print it via function `len(str)`. We make a for loop at **line 9**: `for ix :=0; ix < len(str); ix++`. Here, `ix` is a counter whose initial value is **0** and will increment after every iteration by **1**. The loop will run `len(str)` times. Inside its body, at **line 10**, we are printing every instance of the string line by line by indexing as `str[ix]`.

Similarly, we declare another string at **line 12** as `str2` and initialize it with **“Chinese: 日本語”**. Then, at the next line, we find the length of `str` and print it via function `len(str2)`. We made a for loop at **line 16**: `for ix :=0; ix < len(str2); ix++`. Here, `ix` is a counter whose initial value is **0**, and it will increment after every iteration by **1**. The loop will run `len(str2)` times. Inside its body at **line 17**, we are printing every instance of the string line by line by indexing as `str2[ix]`. The output will show characters of each string line by line.

If we print out the length of strings `str` and `str2`, we get **27** and **18**, respectively. We see that for normal ASCII-characters using 1 byte, an indexed character is a full character. Whereas, for non-ASCII characters (who need 2 to 4 bytes), the indexed character is no longer correct! We can solve this problem through **for range**, which we will cover later in this lesson.

Condition-controlled iteration

The 2nd form contains no header and is used for *condition-controlled iteration* (known as the while-loop in other languages) with the general format:

```
for condition { }
```

You could also argue that it is a `for` without the **init** and **modif** sections, so that the semicolons (; ;) are superfluous.

Run the following program and see how this *for* loop does the required task by only using a condition.

```
package main
import "fmt"

func main() {
    var i int = 0
    // condition controlled for loop with 5 iterations
    for i < 5 {
        fmt.Printf("This is the %d iteration\n", i)
        i = i + 1
    }
}
```

Counting Iterations in a Condition Controlled For Loop

We had written a similar program previously, but this time, we use condition-controlled iterations except for the counter-controlled iteration. At **line 5**, we declare a new variable `var i int =0` instead of doing it in *init* part of for loop. At **line** 7, you can see a condition: `for i<5`, which means that loop will run until `i` is less than 5. You may notice that the *modification* part is missing. In a condition-controlled iteration, we modify the counter in the body of the loop. At **line 8**, we are printing the iteration number of the loop, and at **line 9** is the modification of counter: `i=i+1`. As the counter `i` increases by 1 in each iteration, it means that the loop will run **5** times only. The output is the same as when dealing with counter-controlled iterations.

That's it about how control is transferred using the `for` construct. The next lesson focuses on a variation for running loops called `for range`.