The if-else Construct

This lesson discusses the if-else construct in detail.

WE'LL COVER THE FOLLOWING

Introduction

Until now, we have seen that a Go program starts executing in *main()* and sequentially executes the statements in that function. However, we often want to execute certain statements only if a condition is met, which means we want to make decisions in our code. For this, Go provides the following conditional or branching structures:

- The `if-else` construct
- The `switch-case` construct
- The `select` construct

Repeating one or more statements (a task) can be done with the iterative or looping structure:

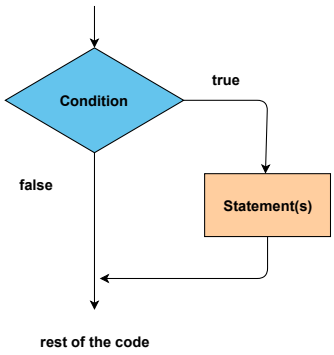
- `for` (range) construct

Some other keywords like `break` and `continue` can also alter the behavior of the loop. There is also a `return` keyword to leave a body of statements and a `goto` keyword to jump the execution to a *label* in the code. Go entirely *omits* the parentheses (and) around conditions in if, switch and for-loops, creating less visual clutter than in Java, C++ or C#.

The `if-else` Construct

The `if` tests a conditional statement. That statement can be logical or boolean. If the statement evaluates to `true`, the body of statements between **{ }** after the `if` is executed, and if it is `false`, these statements are ignored and the statement following the `if` after **}** is executed.

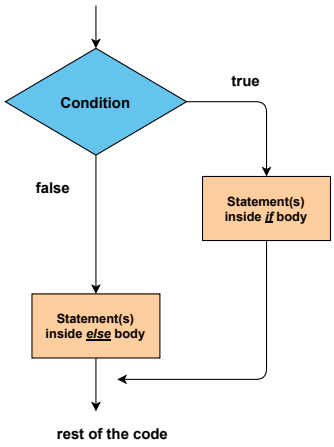
```
if condition {
    // do something
}
```



if structrure

In a 2nd variant, an `else`, with a body of statements surrounded by **{ }**, is appended, which is executed when the condition is `false`. It means we have two *exclusive* branches, and only one of them is executed:

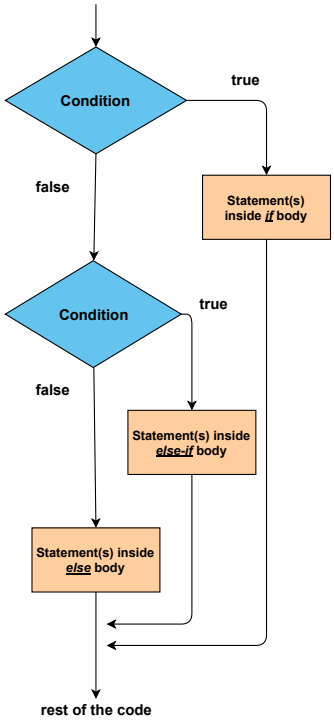
```
if condition {
// do something
} else {
// do something else
}
```



if and else structrure

In a 3rd variant, another `if` condition can be placed after the `else`, so we have 3 *exclusive* branches:

```
if condition1 {
    // do something
} else if condition2 {
    // do something else
} else {
    // catch-all or default
}
```



The number of `else if` branches is in principle, not limited, but for readability reasons, the use of `else if` should not be exaggerated. When using this form, place the condition which is most likely true first.

the condition which is most likely true first.

one else-if along with if and else structrure

The **{ }** braces are mandatory even when there is only one statement in the body (some people do not like this, but it is consistent and follows mainstream software engineering principles). The { after the `if` and `else` must be on the *same line*. The `else if` and `else` keywords must be on the *same line* as the closing } of the previous part of the structure. Both of these rules are mandatory for the compiler. This is in-valid Go code:

```
if x {
}
else { // INVALID
}
```

Note that every branch is indented with 4 (or 8) spaces or 1 tab and that the closing } is vertically aligned with the `if`; this is enforced by applying gofmt (a default formatting tool for Go). The condition can also be composite, using the logical operators `&&`, `||`, and `!` with the use of the parentheses () to enforce precedence or improve readability.

Moreover, the parentheses () around the conditions are not needed. However, for complex conditions, they may be used to make the code clearer. A possible application is the testing of different values of a variable and executing different statements in each case.

As a first example, let's test whether an integer is even or odd:

```
package main
import "fmt"

func main() {
    n := 42
    // Use of control strcuture if and else to check whether number is even or not
    if n % 2 == 0 {
        fmt.Printf("The value is even\n")
    } else {
        fmt.Printf("The value is odd\n")
    }
}
```

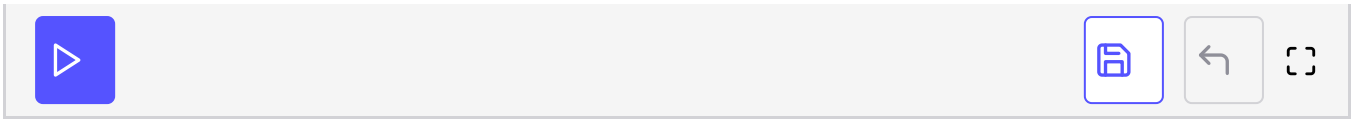
Even Integers

In the above code, we declare a variable `n` at **line 5**. Now, there are only two cases whether a number is even or odd. So we'll use the 2nd variant of the `if-else` structure. We make a condition at **line** 7 that `if n%2==0`, which will be only *true* if `n` is even. If it is true then, we'll print **The value is even**. Otherwise, if the condition is *false* the control will transfer to the `else` structure without even executing **line 8**. Then **line 10** will be executed and **The value is odd** will be printed.

The conditional statement after the `if` is a boolean expression, but it can also be a boolean value, as in the following example:

```
package main
import "fmt"

func main() {
    bool1 := true
    // condition as a boolean value itself
    if bool1 {
        fmt.Printf("The value is true\n")
    } else {
        fmt.Printf("The value is false\n")
    }
}
```



Condition as a Boolean Value

In the above code, we declare a boolean variable `bool1` and initialize it with **true** at **line 5**. At **line** 7, we make the condition `if bool`. Note that it is not necessary to test: if `bool1 == true`, because `bool1` is already a *boolean value*. For this program, control will always transfer to the `if` structure as `bool1` is true, and **The value is true** will be printed on the screen.

For special cases, note that you could write something like:

```
if true {
    fmt.Printf("I'm always executing this branch");
}
```

Here, the code block in the {} is always executed, so this is not very useful.

Inversely you could write:

```
if false {
    fmt.Printf("I will never execute this code!")
}
```

Now, the code block in the {} will never be executed. It is almost always better to test for true or positive conditions, but it is possible to test for the reverse with **!** (not):

```
if !bool1 // or if !(condition)
```

In the last case, the ( ) around the condition are often necessary, for example:

```
if !(var1 == var2)
```

which can be rewritten as the shorter:

```
if var1 != var2
```

The *idiom* in Go-code omits the `else` clause when the `if` ends in a `break`, `continue`, `goto` or `return` statement. For example, when an even value is to be returned from the code, you would write:

```
if n % 2 == 0 {
    return n
}
fmt.Printf("Continuing with an odd value\n")
```

When returning different values `x` and `y` whether or not a condition is true, use the following pattern:

```
if condition {
    return x
}
    return y
```

Here is an example:

```
if n % 2 == 0 {
    return n // return even value
```

```
}
return (n + 1) // return odd value
```

The structure of `if` can start with an initialization statement (in which a value is given to a variable). This takes the form (the ; after the initialization is mandatory):

```
if initialization; condition {
    // do something
}
```

For example, instead of:

```
val := 10
if val > max {
    // do something
}
```

you can write:

```
if val := 10; val > max {
// do something
}
```

But remember, the `val` variable will only be accessible within statements of `if` block. If you try to access `val` anywhere in the program, it will give the compiler error: `undefined: val`. This initialization during the `if` statement can be put to use even more, as the following code snippet shows:

```
if value := process(data); value > max {
    ...
}
```

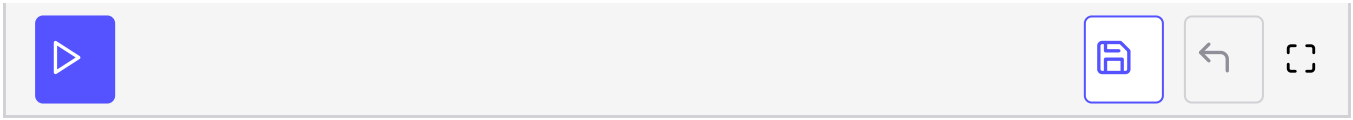
The result of a function `process()` can be retrieved in the `if`, and action is taken according to the `value`.

Here is another complete example using variants of the `if` construct:

```
package main
import "fmt"

func main() {
```

```
    var first int = 10
    var cond int
    if first <= 0 {
        fmt.Printf("first is less than or equal to 0\n")
    } else if first > 0 && first < 5 {
        fmt.Printf("first is between 0 and 5\n")
    } else {
        fmt.Printf("first is 5 or greater\n")
    }
    if cond = 5; cond > 10 {
        fmt.Printf("cond is greater than 10\n")
    } else {
        fmt.Printf("cond is not greater than 10\n")
    }
}
```



If-Else with Variants

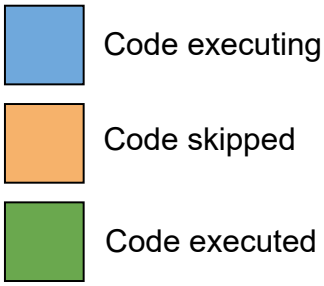
In the above code, we declare two integer variables `first` and `cond`. At **line**7, we make a condition `if first <=0`. If it is *true* for the value of `first`, **line 8** will be executed, and then control will go to **line 14**.

If **false**, control will transfer from **line** 7 to **line 9** directly. There, we make another condition using `else if`, i.e., `else if first > 0 && first < 5`. If this condition is *true*, then control will transfer to **line 10** and then directly to **line 14**. Otherwise, control will be transferred directly to **line 11** from **line 9**.

At **line 11** is the `else` structure. Within the `else` structure, **line 12** will be executed. After executing statements within `else`, control will come to **line 14**. If the condition at **line 14** is true, then **line 15** will be executed. Otherwise, **line 17** within `else` structure.

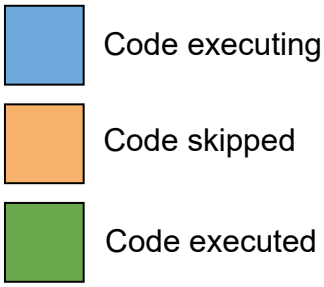
Look at the condition on **line 14**: `if cond =5; cond > 10`. We are initializing `cond` at the time of condition.

Below is the illustration that shows the transfer of control throughout the above program.



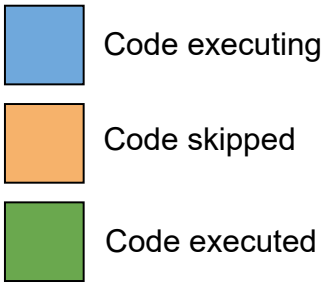

```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution



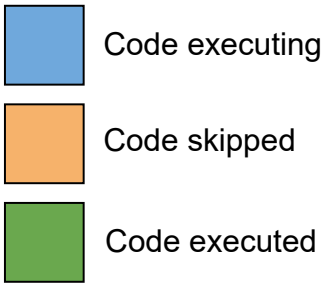

```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution



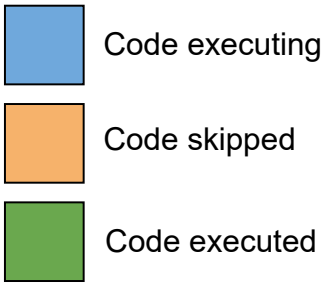

```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution




```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

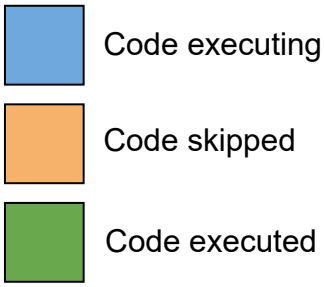
Flow of Control of Execution



False

```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution




```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution

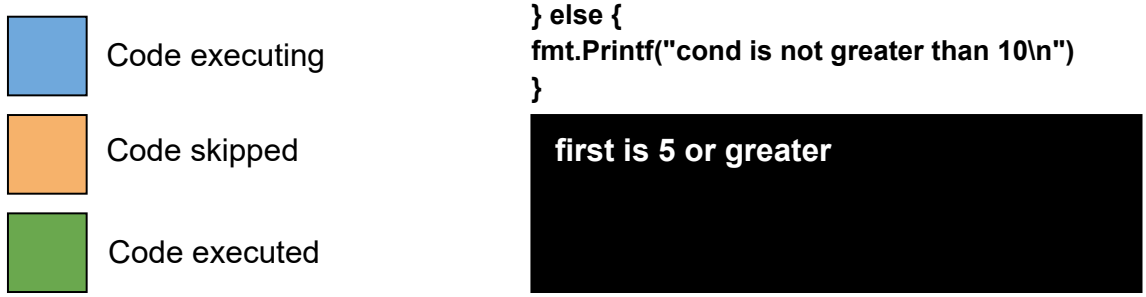
```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

False

Flow of Control of Execution


```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

Flow of Control of Execution

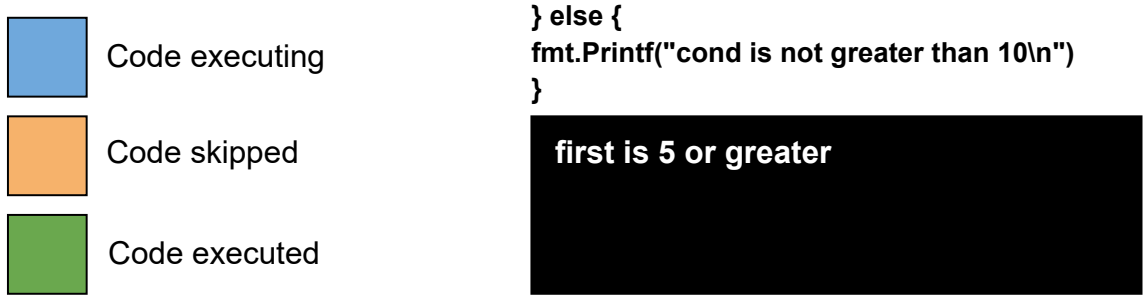



```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

True

```
first is 5 or greater
```

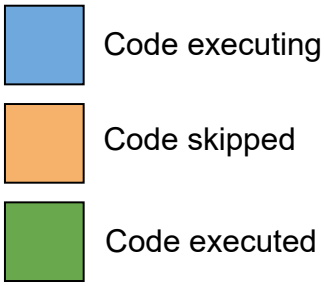
Flow of Control of Execution




```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

```
first is 5 or greater
```

Flow of Control of Execution

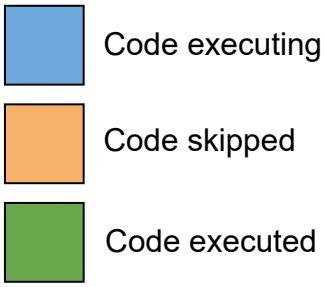



```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

False

```
first is 5 or greater
```

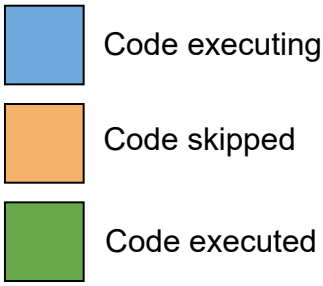
Flow of Control of Execution




```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

```
first is 5 or greater
```

Flow of Control of Execution

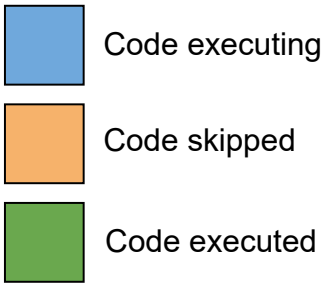



```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

True

```
first is 5 or greater
cond is not greater than 10
```

Flow of Control of Execution




```
var first int = 10
var cond int
if first <= 0 {
fmt.Printf("first is less than or equal to 0\n")
} else if first > 0 && first < 5 {
fmt.Printf("first is between 0 and 5\n")
} else {
fmt.Printf("first is 5 or greater\n")
}
if cond = 5; cond > 10 {
fmt.Printf("cond is greater than 10\n")
} else {
fmt.Printf("cond is not greater than 10\n")
}
```

```
first is 5 or greater
cond is not greater than 10
```

Flow of Control of Execution

That’s it about how control is transferred using the if-else construct. The next lesson describes how to test for errors on built-in functions.