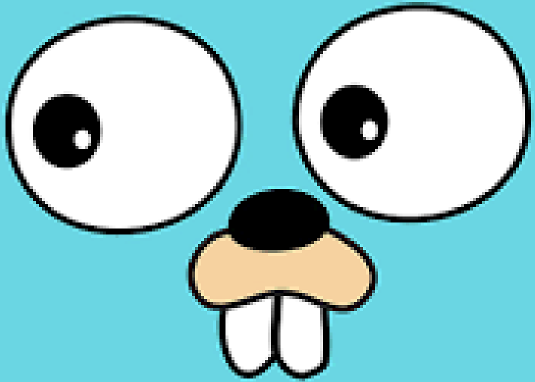


Phase 1: Introduction to Golang (Sessions 1-8)

Objective: Build a strong foundation in Golang basics.

Session 3: Operators & Control Structures

Discussion Points

- Go Operators
- Conditional statements (if, else, switch)
- Loops (for, range)
- Basic error handling

What is Go Operators?

In Computer Programming, an operator is a symbol that performs operations on a value or a variable.

For example, + is an operator that is used to add two numbers.

Go programming provides wide range of operators that are categorized into following major categories:

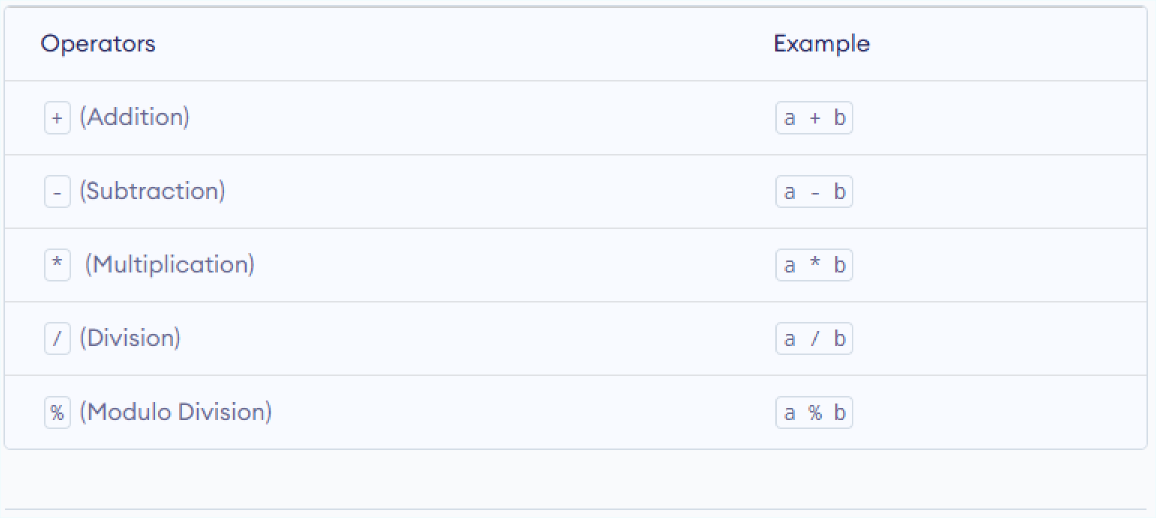
- Arithmetic operators
- Assignment operator
- Relational operators
- Logical operators
- *Binary Shift Operator*

Go Operators

What is Arithmetic operators?

We use arithmetic operators to perform arithmetic operations like addition, subtraction, multiplication, and division.

Here's a list of various arithmetic operators available in Go.



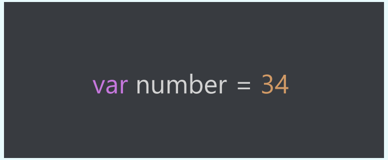
Operators	Example
+ (Addition)	a + b
- (Subtraction)	a - b
* (Multiplication)	a * b
/ (Division)	a / b
% (Modulo Division)	a % b

Go Operators

What is Assignment operator?

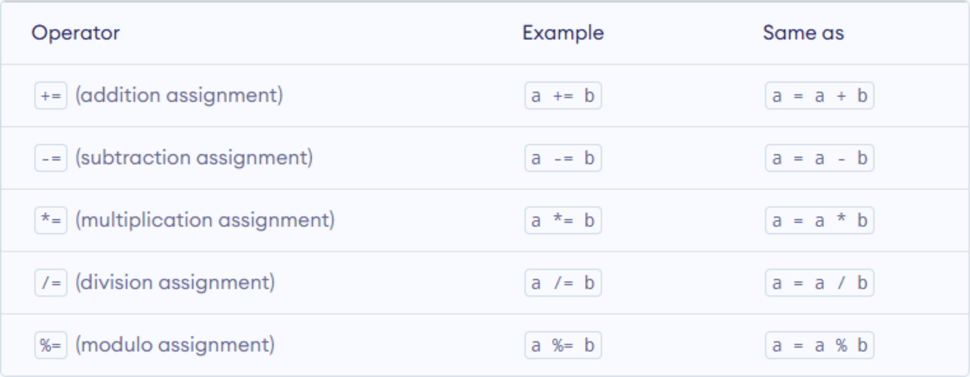
We use the assignment operator to assign values to a variable

Assignment Operator



```
var number = 34
```

Compound Assignment Operator

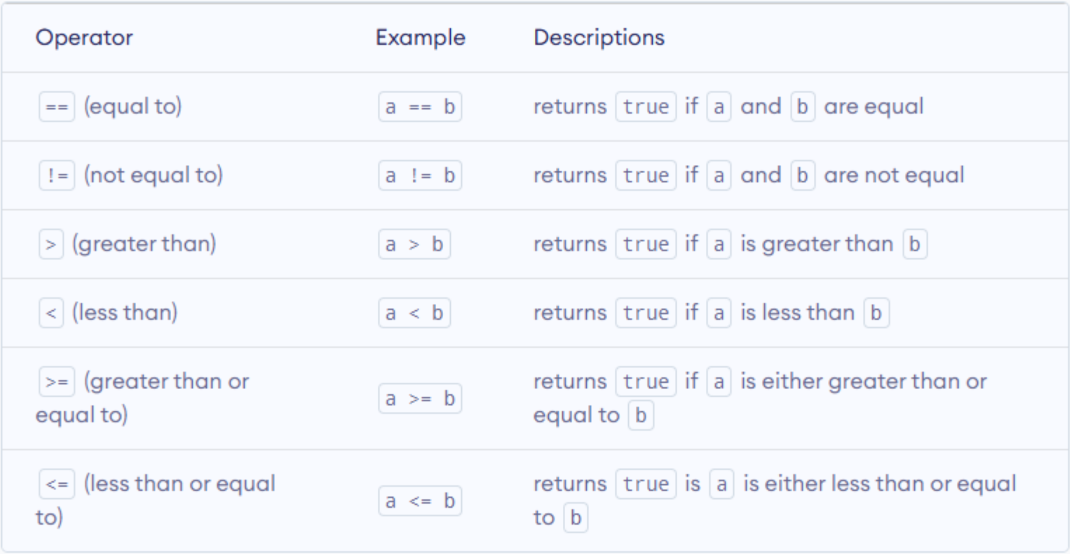


Operator	Example	Same as
`+=` (addition assignment)	`a += b`	`a = a + b`
`-=` (subtraction assignment)	`a -= b`	`a = a - b`
`*=` (multiplication assignment)	`a *= b`	`a = a * b`
`/=` (division assignment)	`a /= b`	`a = a / b`
`%=` (modulo assignment)	`a %= b`	`a = a % b`

Go Operators

What is Relational operator?

We use the relational operators to compare two values or variables

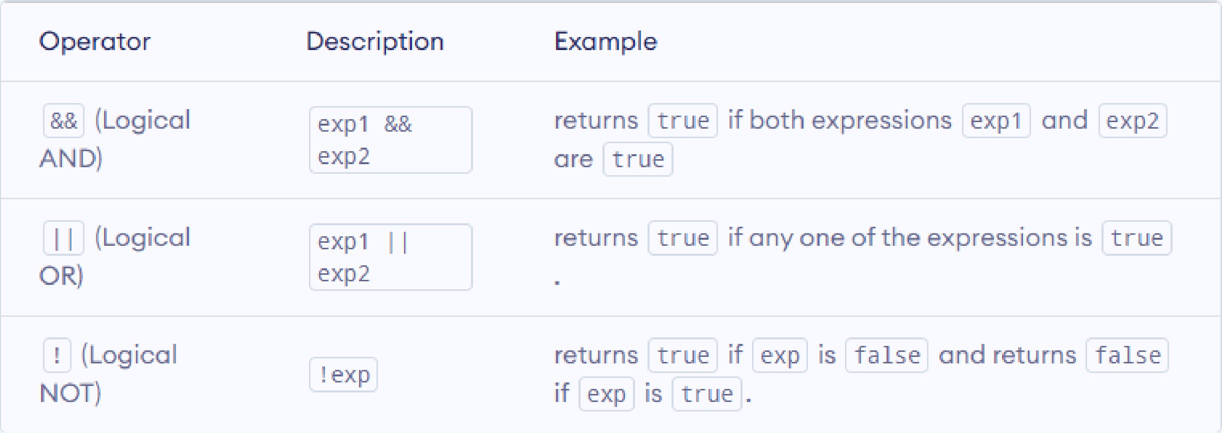


Operator	Example	Descriptions
`==` (equal to)	`a == b`	returns `true` if `a` and `b` are equal
`!=` (not equal to)	`a != b`	returns `true` if `a` and `b` are not equal
`>` (greater than)	`a > b`	returns `true` if `a` is greater than `b`
`<` (less than)	`a < b`	returns `true` if `a` is less than `b`
`>=` (greater than or equal to)	`a >= b`	returns `true` if `a` is either greater than or equal to `b`
`<=` (less than or equal to)	`a <= b`	returns `true` is `a` is either less than or equal to `b`

Go Operators

What is Logical operator?

We use the logical operators to perform logical operations. A logical operator returns either **true** or **false** depending upon the conditions.



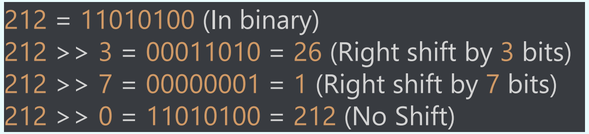
Operator	Description	Example
`&&` (Logical AND)	`exp1 && exp2`	returns `true` if both expressions `exp1` and `exp2` are `true`
`||` (Logical OR)	`exp1 || exp2`	returns `true` if any one of the expressions is `true` .
`!` (Logical NOT)	`!exp`	returns `true` if `exp` is `false` and returns `false` if `exp` is `true`.

Go Operators

What is Binary Shift Operator?

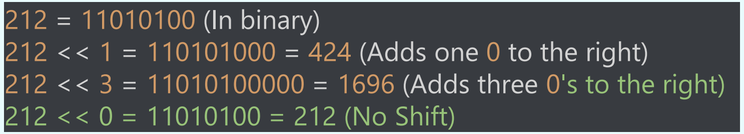
The bitwise left shift operator, which shifts the bits of corresponding integer to the left….the rightmost bit being ‘0’ after the shift

Right Shift



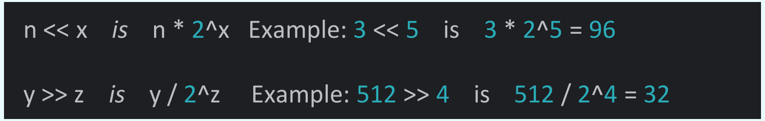
```
212 = 11010100 (In binary)
212 >> 3 = 00011010 = 26 (Right shift by 3 bits)
212 >> 7 = 00000001 = 1 (Right shift by 7 bits)
212 >> 0 = 11010100 = 212 (No Shift)
```

Left Shift



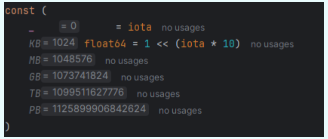
```
212 = 11010100 (In binary)
212 << 1 = 110101000 = 424 (Adds one 0 to the right)
212 << 3 = 11010100000 = 1696 (Adds three 0's to the right)
212 << 0 = 11010100 = 212 (No Shift)
```

Sugar



```
n << x   is   n * 2^x   Example: 3 << 5   is   3 * 2^5 = 96

y >> z   is   y / 2^z    Example: 512 >> 4   is   512 / 2^4 = 32
```



```
const (
    _    = 0        = iota  no usages
    KB = 1024  float64 = 1 << (iota * 10)  no usages
    MB = 1048576   no usages
    GB = 1073741824   no usages
    TB = 1099511627776   no usages
    PB = 1125899906842624   no usages
)
```

Conditional statements (if, else, switch)

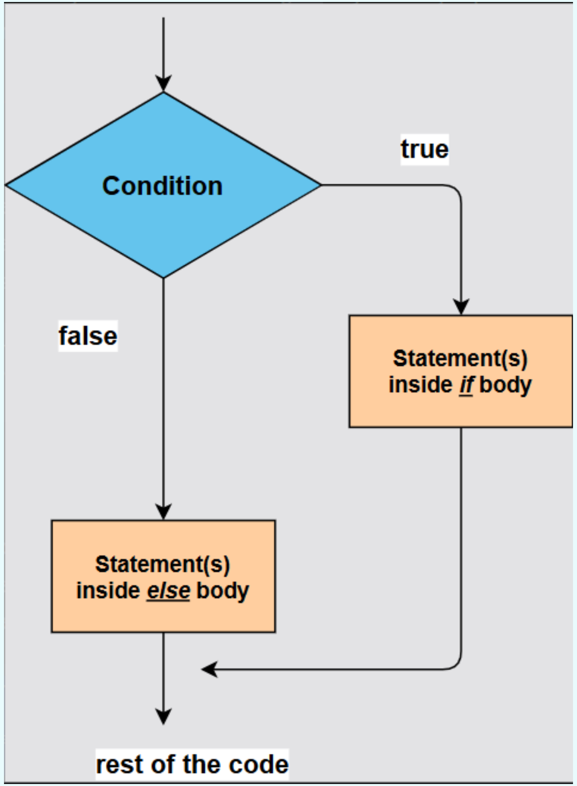
What is if, else?

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
}
```

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
}
```

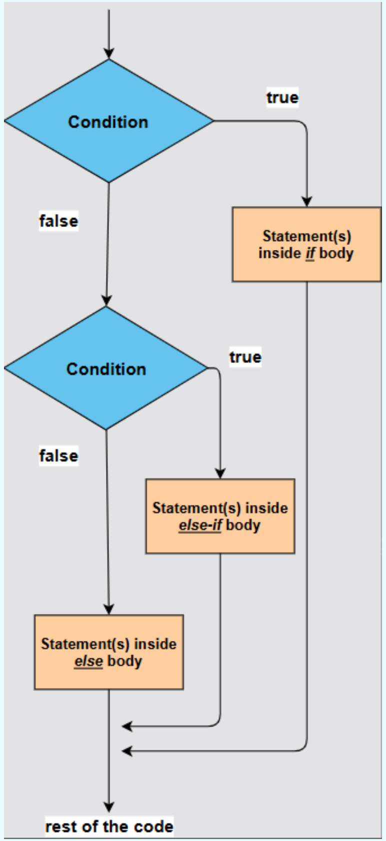

Condition
true
false
Statement(s) inside *if* body
Statement(s) inside *else* body
rest of the code

Conditional statements (if, else, switch)

What is if, else?

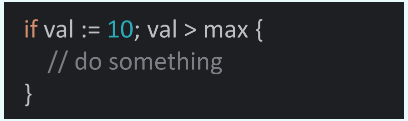
In a 3rd variant, another **if** condition can be placed after the **else**, so we have 3 exclusive branches:

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


Condition
true
false
Statement(s) inside *if* body
Condition
true
false
Statement(s) inside *else-if* body
Statement(s) inside *else* body
rest of the code

What is if, else?

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):

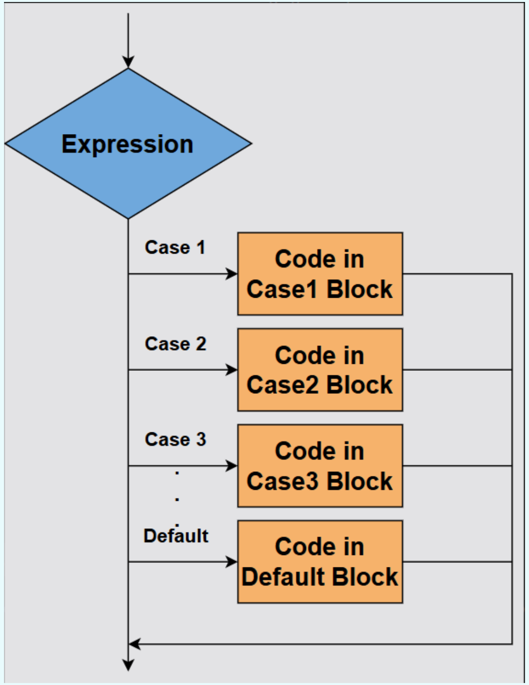


```
if val := 10; val > max {
   // do something
}
```

Conditional statements (if, else, switch)

What is switch?

The keyword **switch** is used instead of long **if** statements that compare a variable to different values. The **switch** statement is a multiway branch statement that provides an easy way to transfer flow of execution to different parts of code based on the value. The following figure explains the basic structure of the **switch-case** construct.



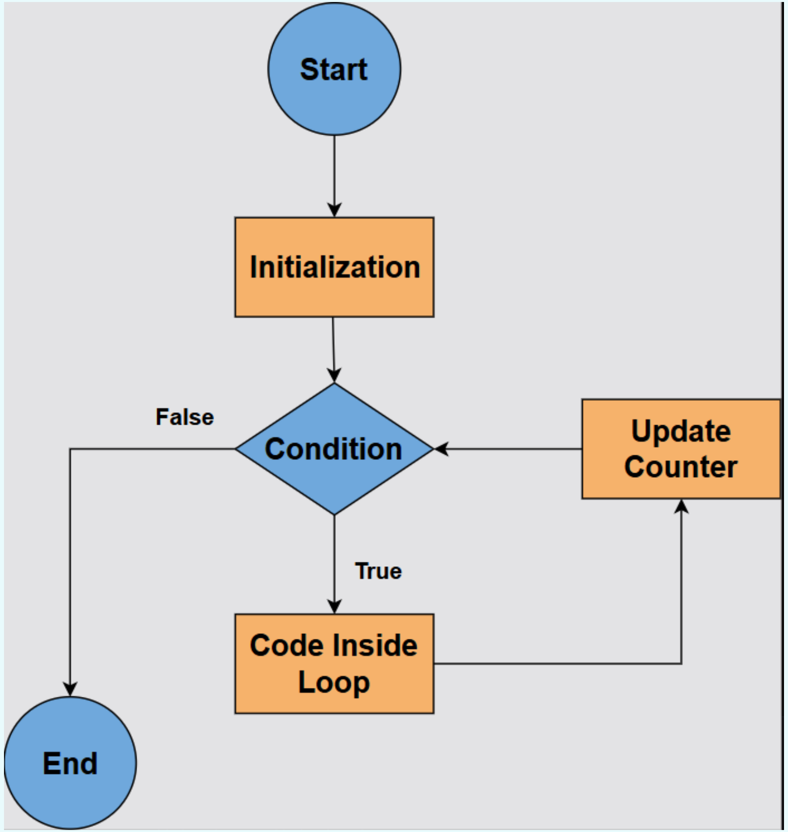
```
switch var1 {
  case val1:
    ...
   case val2:
    ...
   default:
    ...
}
```

```
switch {
  case i < 0:
    f1() // function call
  case i == 0:
    f2() // function call
  default:
    f3() // function call
}
```

Loops (for, range)

What is for?

The keyword **switch** is used instead of long **if** statements that compare a variable to different values. The **switch** statement is a multiway branch statement that provides an easy way to transfer flow of execution to different parts of code based on the value. The following figure explains the basic structure of the **switch-case** construct.



Loops (for, range)

What is range?

The for range is the iterator construct in Go, and you will find it useful in a lot of contexts. It is a very elegant variation used to make a loop over every item in a collection. The general format is:

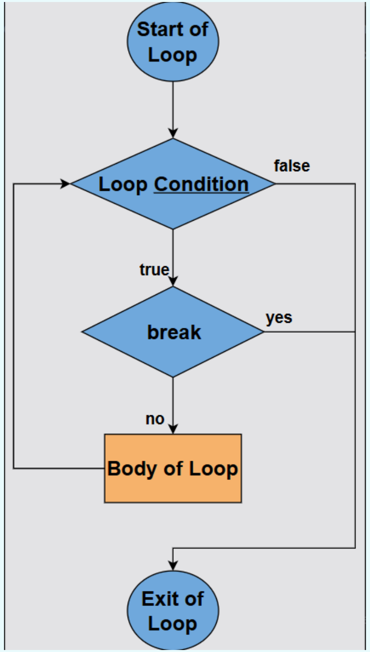
```
for ix, val := range coll { }
```

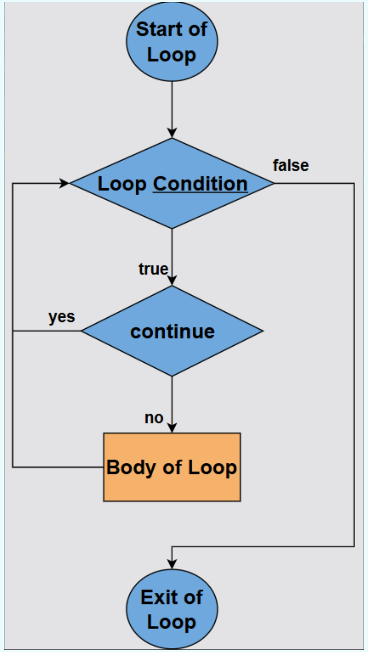
Loops (for, range)

What is break and continue?

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.

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.





Basic error handling

What is Basic error handling?

Go does not have an exception-handling mechanism, like the try/catch in Java or .NET. For instance, you cannot throw exceptions. Instead, it has a defer-panic-and-recover mechanism. The designers of Go thought that the try/catch mechanism is overused and that the throwing of exceptions in lower layers to higher layers of code uses too many resources. The mechanism they devised for Go can ‘catch’ an exception, but it is much lighter. Even then, it should only be used as a last resort.

Session Quiz

Go?

Join at
slido.com
#1831999

Thank you.

Questions?

- Go Operators
- Conditional statements (if, else, switch)
- Loops (for, range)
- Basic error handling

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- Albert Einstein