The GoLang Beginners Handbook

By Abhishek Tiwari

>>>What is GoLang?

Go programming language, something referred as GoLang because of its domain name GoLang.org, is an easy to learn and use, statically typed, compiled programming language.

It was designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but has external features like memory safety, garbage collection, structural typing, and CSP-style concurrency.

It was first released on November 10 2009.

Some features of Go are:

1. Easy to learn and use
2. Fast (better for competitive programming)
3. Object Oriented
4. Concurrent
5. Pointers like C or C++

Go is a very small programming language and can be learned very quickly.

>>>Installation and Setup:

To run Go on our machine, we need to install the Go compiler from its official website.(click here to download)

We also need a code editor to write our code. We will be using Visual Studio Code.

The installation process of both of them are very simple so I will not cover that.

If you don't want to install any of these, you can use the official online go compiler available at play.golang.org.

>>>A few steps before learning...

Before starting, make a new folder and open it in VS Code. Now, make a file with the .go extension in the folder.

Now install an extension in VS Code named "Code Runner" by clicking on the extensions menu located in the left side of the screen and searching for the same.

At last, open your go file again and paste the following code in it. Then press the play button located in the top right corner of the screen to run the code in the terminal.

```
package main

import (
     "fmt"
)

func main() {
     fmt.Println("Hello World!")
}
```

If you don't see the play button, click on the Terminal tab located on the top of the screen and then press New terminal. This will open up the terminal.

In the terminal, type go run, followed by the file name with extension. So, If my file name is main.go, I will type: *go run main.go*

You will get the **output Hello World!** in the terminal. From now, everything we type will be typed inside the curly braces of the main function as it is the place from where the execution of our program starts.

>>>Understanding our first program:

In the previous section, we wrote a simple program which outputted "Hello World".

Let's understand how the program worked.

The first line was *package main*. We need to understand that all our code is grouped as a package in go. It is not necessary to name our package main.

The second line was *import fmt*. The keyword import is used to include other's packages in our code. In the above code, we imported the fmt package which includes code which can be used to print to the terminal, take input and much more. To import more than one package, we can put all the package names inside round brackets ().

The third line was *func main(){.....}*. This is the starting point of the program. Anything written inside the main function is executed by the compiler.

The last line was *fmt.Println("Hello World!")*. This line calls a function called *Println* which is defined in the package *fmt*. It takes an argument which in our case was "Hello World", and outputs the argument to the terminal. We will learn more about arguments and functions later.

For now, just make sure that functions are ways of grouping a specific code together to reuse it multiple times in the program without typing that code again and again. Function can be called by typing the package name, followed by a dot, followed by the function name.(Make sure you type the first alphabet of all function names in capital). Arguments are the values with which the function works.

>>>Variables:

Variables are one of the core concepts of programming. Variables are simply some placeholders to store values temporarily. The value stored in a variable can be changed anytime during the execution of code.

Variables in GoLang can be declared by three methods:

1. Declare on one line and assign on the other.This method is generally used to declare variables when we don't want to assign the variable it's value at the same time.

   To use this method, first type the *var* keyword, followed by the variable name and it's type. Now as we are done with the declaration of the variable, let's assign it a value. To assign the value, type the variable name, followed by an equal sign, and at last type the value to be assigned(make sure the value is of the same type as the mentioned while declaring the variable)

```
var age int // Declare the variable age as an integer type
age = 13 // Assign the integer value 13 to variable age
```

2. The 2nd type of declaring variable is a single line declare and assign statement. To use it, type the *var* keyword, followed by the variable name, followed by the type of value to be stored(optional). Then put the equal sign followed by the value to be assigned.

```
var age int = 13 // declare an age variable and assign the
integer value 13
```

3. The 3rd type of declaring variable is a single line shortcut type. By this type you only need to type the variable name, := operator and the value. The go compiler will automatically choose the preferable data type as per the value assigned.

```
age := 13 // declare a variable age and assign the value 13 to it.
The go compiler will automatically identify this as an integer value.
```

There are some **rules** which need to be followed while **declaring variable names**:

- Variable names should only include alphabets, numbers and underscores(_).
- Variable names can start with only alphabets and underscores.
- Variable names cannot start with numbers or special characters.
- Variable names cannot include spaces.
- Variable names are case sensitive
  (*age* and *Age* are two different variable names)
- To use more than one word as a variable name, use camel casing or underscores.
  (*my name is* can be written as *myNameIs* or *my_name_is*)
- Use senseful variable names.
  (A variable containing your name may be named as *MyName* but not n, x, y, etc.)

As we read before, the values stored in a variable can be changed. For example:

```
name := "GoLang"
fmt.Println(name) // will output "GoLang"
name = "Go Programming"
fmt.Println(name) // will output "Go Programming"
```

Another amazing feature of go is that it treats unused variables in the code as errors. This means that if you declare and and assign a variable or do any one of them only, and don't use that variable to do anything in your program, you get an error. This helps to remove unwanted variables taking space in the memory from the program.

Level of Variables in GoLang:

- **Package Level (Global access)**
- **Function level (Local access)**

Package level variables:
These types of variables can be accessed anywhere in the program. To declare these types of variables, you need to declare the variable in the main package and not in any function inside the code.

```
package main

import (
     "fmt"
)
age := 13 // This is a global variable
func main() {
     age := 13 // This is a not a global variable
}
```

Function level variables:
These variables are declared inside functions and can be used within that function only.

```
package main

import (
     "fmt"
)
age := 13 // This is a not a local variable
func main() {
     age := 13  // This is a local variable
}
```

Using CAPITAL LETTERS for naming variables:

When working on a large project, the code files are separate. If we want to use the variables which are declared in another file, we can't use it normally. To do so, we need to make sure that the variable names are in capital letters.

So, we need to use Variable *AGE* instead of *age* to use it in other files as well.

>>>Constants:

We have learned that the value stored in a variable can be changed easily. Sometimes, we don't want this.

In that case, we use constants. Constants are placeholders whose value can't be changed. To declare a constant, we use the keyword *const* in place of *var*.

```
const pi = 3.145
pi = 3.2 // This will throw an error as we can't change the values of constants
```

>>>Comments:

Comments are statements within the code which are ignored by the go compiler. It helps to take note of anything within the program. An example of comments is the code above. The line "This will throw..." is a comment. To define a comment in the code, put two forward slashes before the statement. For example:

```
// This is a comment
```

What we just saw was a single line comment. But, what if we need to comment more than 1 line, we use multi-line comment. To declare multi-line comments, we put our comment between /* and */.

Example:

```
/*This is
a multi-line comment*/
```

>>>Data types in GoLang:

Data types are again one of the most important topics in any programming language.

Data types simply mean the different types of values we can work with.

The following mainly used data types are present in GoLang:

1. Numbers
    a. Integers
    b. Unsigned Integers
    c. Floating point numbers
    d. Others
2. Boolean
    a. true = 1
    b. false = 0
3. Strings (Normal text)
4. Derived types (advanced topic hence not covered in this book)
    a. Pointer types
    b. Array types
    c. Structure types
    d. Union types
    e. Function types
    f. Slice types
    g. Interface types
    h. Map types
    i. Channel Types

Numbers:-

a) Integers:(Just like normal integers in math, -ve, +ve and 0)

int8	=>	8-bit (Signed)	=>	-128 to 127
int16	=>	16-bit (Signed)	=>	-32768 to 32767
int32	=>	32-bit (Signed)	=>	-2147483648 to 2147483647
int64	=>	64-bit (Signed)	=>	-9223372036854775808 to 92233720368547758O

b) Unsigned Integers:(Just like whole numbers in math, 0 to +ve)

uint8	=>	8-bit (unsigned)	=>	0 to 255
uint16	=>	16-bit (unsigned)	=>	0 to 65535
uint32	=>	32-bit (unsigned)	=>	0 to 4294967295
uint64	=>	64-bit (unsigned)	=>	0 to 18446744073709551615

c) Floating point numbers:(Just like decimal numbers in math)

float32	=>	32-bit (IEEE-754)
float64	=>	64-bit (IEEE-754)
complex32	=>	32-bit
complex64	=>	64-bit

d) Other numerical data types:

byte	=>	same as uint8
rune	=>	same as int32
uint	=>	32 or 64 bits (Depends upon the machine)
int	=>	32 or 64 bits (Depends upon the machine)
uintptr	=>	an unsigned integer to store the uninterpreted bits of a pointer value

The bits in all the number data types tell the amount of space it will occupy in the system memory. Try using the least possible bit so that it does not occupy extra space in the memory, slowing up the program.

Boolean:-

It is probably the simplest data type in Go, as it only consists of *true* and *false*. To declare a boolean variable, we use the keyword *bool* during defining the data type of variable.

```
var switchOn bool = true
var switchOff bool = false
```

In boolean, true = 1 & false = 0. As they represent numbers, we can perform arithmetic operations on them as well. We'll look into it later.

Strings:-

Strings are the normal text that we see. In any programming language, strings are identified by being enclosed in Double quotes(“”).

```
var myName string = "Abhishek"
```

Stings can also be empty, by just not entering any value within the quotes.

```
var emptyString string = "" // The variable emptyString
holds an empty string
```

A string is a slice of bytes in Go. Strings in Go are Unicode compliant and are UTF-8 Encoded.

Formatted Strings-

Sometimes, we need to print a value in between a string. For that we use formatted strings. To print a string consisting of formatted information, we use *fmt.Printf* which represents print formatted information.

Some major string formatters are listed below:

1. **General:**

%v => Used to print the value =>

```
age := 13
fmt.Printf("I am %v years old", age)
```

%T => Used to print the type of value =>

```
iAmOld := false
fmt.Printf("Am I old? %T", iAmOld)
```

%% => Used to print a % symbol =>

```
fmt.Printf("I scored 95 %% in the text")
```

2. **Boolean:**

%t => Used to print the value of boolean =>

```
switchOn := false
fmt.Printf("The switch is on = %t", switchOn)
```

3. **Integer:**

%b => Get the binary form or base 2 of any integer =>

```
myAge := 13
fmt.Printf("My age in binary is %b", myAge)
```

%o => Get the octal form or base 8 of any integer =>

```
myAge := 13
fmt.Printf("My age in octal is %o", myAge)
```

%d => Get the decimal or base 10 of any integer =>

```
myAgeInOctal := 15
fmt.Printf("My age in decimal is %d", myAgeInOctal)
```

%x => Get the hexadecimal or base 16 of any integer =>

```
myAge := 13
fmt.Printf("My age in hexadecimal is %x", myAge)
```

As hexadecimal consists of alphabets also, you can use capital x (%X) to get the outputted alphabets in capital.

```
myAge := 13
fmt.Printf("My age in hexadecimal is %X", myAge)
```

4. **Floating point numbers:**

%e => Get the scientific notation =>

```
var valueOfPi float64 = 3.14159265358979323846264338327950288
fmt.Printf("The exact value of pi is %e", valueOfPi)
```

%g => Used for large exponents =>

```
var valueOfPi float64 = 3.14159265358979323846264338327950288
fmt.Printf("The exact value of pi is %g", valueOfPi)
```

%f => Used to shorten a large floating point number =>

```
var valueOfPi float64 = 3.14159265358979323846264338327950288
fmt.Printf("The exact value of pi is %f", valueOfPi)
```

5. **Strings:**

%s => Output the string as it is =>

```
myName := "Abhishek"
fmt.Printf("I am %s", myName)
```

%q => Outputs the string enclosed within double quotes =>

```
myName := "Abhishek"
fmt.Printf("I am %q", myName)
```

>>>Taking Input in GoLang:

There are 2 ways of taking input from the user in the terminal:

1. **Using Scanln function in fmt**(easy, short but less recommended)

   As I said at the start, the fmt package is also capable of taking input from the user in the terminal by using the scanln function.

```
var myAge int
fmt.Scanln(&myAge)
```

You must have noticed the **&** sign in the round brackets of Scanln function. The & sign refers to the location in the memory where the value of the variable after the & sign is stored. So, after taking the input, the Scanln function stores the value to the place in the same memory location of the variable.

The problem with this function is that sometimes it does not work as we want. For example, if someone enters more than one word in the terminal, this function will use the first word for further processing and return an error for the other words entered.

2. Using another package - *bufio*(Lengthy, highly recommended)

   There is another package in go which is specially designed for taking input. It is called *bufio*. It is a little lengthy to use, but is recommended as it is the official and non-buggy way of taking input from the terminal. To use it-

```
package main
import (
    "fmt"
    "bufio"
    "os"
)
func main(){
scanner := bufio.NewScanner(os.Stdin) // Create a scanner
scanner.Scan() // Scan the line from the scanner
input := scanner.Text() // Declare a variable input and
store the value taken by the scanner in the variable input.

}
```

>>>Arithmetic Operators in GoLang:-

Arithmetic operators are used to perform Mathematical operations. The following arithmetic operators present in GoLang:

Operator	Description	Example a=20, b=30
+	Addition	a + b = 50
-	Subtraction	a - b = -10
*	Multiplication	a * b = 600
/	Division without remainder	a / b = 0
%	Remainder	a % b = 20
++	Add 1 (increment operator)	a ++ will give 21
--	Subtract 1 (decrement operator)	a -- will give 29

Click here to see sample code for this topic.

>>>Relational Operators:

We have got to know about Boolean values already. Relational operators are some operators that tell whether an equation is correct or not and returns a boolean value. For example, == operator is used to see if 2 values are the same or not. So, 2 == 3 will give *false*.

Operator	Description	Example
== (Equal to)	Check if 2 expressions/values are the same.	23 == 23 will give True 23 == 32 will give false
!= (not equal to)	Checks if 2 values are not the same.	23 != 23 will give false 23 != 32 will give true
>(Comparison operator)	Checks if the value on the left hand side is greater than the value on the right hand side	23 > 13 will give true 23 > 32 will give false
<(Comparison operator)	Checks if the value on the right hand side is greater than the value on the left hand side	23 < 13 will give false 23 < 32 will give true
>=	Checks if the value on the left hand side is greater or equal to the value on the right hand side	23 >= 23 will give true 23 >= 32 will give false
<=	Checks if the value on the left hand side is smaller or equal to the value on the right hand side	23 <= 23 will give true 23 <= 32 will give true

>>>Logical Operators:

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. There are 3 logical operators in GoLang:

Operator	Description	Example
&& (Logical AND)	Returns true when both the conditions in consideration are satisfied. Otherwise it returns false.	a && b returns true when both a and b are true (i.e. non-zero).
|| (Logical OR)	returns true when one (or both) of the conditions in consideration is satisfied. Otherwise it returns false.	a || b returns true if one of a or b is true (i.e. non-zero). Of course, it returns true when both a and b are true.
! (Logical NOT)	returns true if the condition in consideration is not satisfied. Otherwise it returns false.	returns true if the condition in consideration is not satisfied. Otherwise it returns false.

>>>Bitwise Operator:

In Go language, there are 6 bitwise operators which work at bit level or used to perform bit by bit operations. Following are the bitwise operators :

Operator	Description
& (Bitwise AND)	Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
| (Bitwise OR)	Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
^ (Bitwise XOR)	Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
<< (Left shift)	Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
>> (Right shift)	Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
&^ (AND NOT)	This is a bit clear operator.

Click here to open sample code for the same topic.

>>>Assignment Operators:

Assignment operators are used to assign a value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error. Different types of assignment operators are shown below:

Operator	Description
= (Simple Assignment)	Used to assign values to variables normally.
+= (Add Assignment)	This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
-= (Subtract Assignment)	This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.
*= (Multiply Assignment)	This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
/= (Divide Assignment)	This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
%= (Modulus Assignment)	This operator first modulo the current value of the variable on left by the value on the right and then

	assigns the result to the variable on the left.
&= (Bitwise AND Assignment)	This operator first “Bitwise AND” the current value of the variable on the left by the value on the right and then assigns the result to the variable on the left.
^= (Bitwise exclusive OR)	This operator first “Bitwise Exclusive OR” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
|= (Bitwise inclusive OR)	This operator first “Bitwise Inclusive OR” the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

>>>Misc Operators:

Some special operators:

Operator	Description	Examples
&	Returns the memory address of the variable	Click here to see the sample code for the same.
*****	Provides a pointer to the variable	
<-	Receive a value from the channel	

>>>Decision making in GoLang:

Decision making in programming is similar to decision making in real life. For example, if you need a driving licence you will only apply for it if a specific condition, that you are above 18 is True.

Decision making statements in GoLang are:

1. **If statements**
2. **If-else statements**
3. **If-else-if ladder**

1) If statements:

This is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not. If a certain condition is true then a block of statement written within it is executed, otherwise it is not executed. The syntax of If statements in Go is:

```
if (condition){
/* Statements to execute if condition is true */
}
```

So, let's make a simple program to apply the above example of driving license.

```
package main
import (
     "fmt"
)
func main() {
     age := 20
     if age >= 18{
     fmt.Println("You can get a driving licence")
     }
}
```

2) If-else statements:

The above program will output *you can get a driving licence* if the value of *age* is 18 or above.

But what if the value of *age* is less than 18. Then the program will end without giving any output. In that case, we use the *else* statement. The *else* statement will be executed if the condition given in the if statement evaluates to *false*.

So let's modify our driving licence program to add an else statement.

```
age := 10
     if age >= 18{
     fmt.Println("You can get a driving licence")
     }else{
     fmt.Println("You cannot get a driving licence")
     }
}
```

Now as we have changed the value of *age* to 10, the program instead of exiting will give the output *You cannot get a driving licence.*

4) If-else-if ladder:

Till now we have seen how to put 2 conditions in a program with the help of *If* and *else* statements. But what if we need more than 2 conditions in the code.

In that case, we use the *If-else-if ladder*. This helps us to give multiple conditions in our program. The syntax of *If-else-if ladder* :

```
if(condition_1) {
    /* this block will execute when condition_1 is true */

} else if(condition_2) {

   /*this block will execute when condition_2 is true */
}
.
.
. else {
   /*this block will execute when none of the condition is true */
}
```

Let's modify our driving licence program to give the output according to the age.

```
age := 20
     if age >= 18{
          fmt.Println("You can get a driving licence")
     } else if age == 17{
          fmt.Println("You will get a driving license very
soon...")
     } else if age == 16{
          fmt.Println("You will get a driving license in 2
years")
     } else{
     fmt.Println("You are too young to get a driving
licence")
     }
}
```

Try changing the value of age and see what happens.
Exercise - Modify this code to take the value of age as input from the user.

>>>Loops in GoLang:

Unlike other programming languages with multiple types of loops, Go only has one type of loop, the *for loop*. But this doesn't mean that go has less features in the term of loops. The for loop in Go can work as a *while loop* as well.

- The syntax of *for loop* in Go when used as a simple loop is as follows:

```
for initialization; condition; post{
       // statements....
}
```

Let's make a simple program to output the numbers from 0 to 6. For that we need to initialize a variable, then put the condition, followed by an expression which will be executed after the loop iterates.

```
for i:=0; i<=6; i++{
fmt.Println(i)
}
```

- Lets now see how to use a *for loop* as an *infinite loop*. For that, the syntax is:

```
for{
     // Statement...
}
```

Let's make a loop to run infinitely and print *GoLang*.

```
for{
fmt.Println("GoLang")
}
```

Let's now see how to use a *for loop* as a *while loop*. For that we have the following syntax:

```
for condition{
   // statement..
}
```

Let's now make a *for loop* to work as a *while loop*.

```
i := 0
for i <=20{
fmt.Println(i)
i++
}
```

>>>Switch Statements in GoLang:

A switch statement is a multiway branch statement. It provides an efficient way to transfer the execution to different parts of a code based on the value(also called case) of the expression. Go language supports two types of switch statements:

1. **Expression Switch**
2. **Type Switch(Advanced topic hence not covered)**

1) Expression Switch statements:-

The Expression Switch statements in GoLang are just like the normal switch statements in other programming languages. It provides an easy way to dispatch execution to different parts of code based on the value of the expression. The syntax of Expression Switch Statement in Go is:

```
switch optstatement; optexpression{
case expression1:
Statement..
case expression2:
Statement..
...
default:
Statement..
}
```

Let's write a simple switch statement to output the value of a variable in words.

```
day:=2
switch day{
case 1:
fmt.Println("Monday")
case 2:
fmt.Println("Tuesday")
case 3:
fmt.Println("Wednesday")
case 4:
fmt.Println("Thursday")
case 5:
fmt.Println("Friday")
case 6:
fmt.Println("Saturday")
case 7:
fmt.Println("Sunday")
}
```

Change the value of day between 1 to 7 and see the output.
Mini-Exercise - Take the value of *day* as input and apply it in the switch statement.

*You can put multiple cases in one line by separating with commas i.e. *case* 7, 8, 9:
**In place of numbers after the *case* keyword, you can also use expressions and strings.

>>>Functions in GoLang:

Assume that you are creating a program where you need to find the average of some numbers 15 times. In that case, you will copy paste the average formula 15 times, making the code longer. In programming, we have something called DRY, standing for Don't repeat yourself.

In this condition, *functions* come to rescue. *Functions* are blocks of code which can be reused multiple times in the program. This makes the code cleaner and shorter. Ultimately saves the excessive use of memory, acts as a time saver and more importantly, provides better readability of the code. The following syntax should be followed while declaring a function:

```
func function_name(Parameter-list)(Return_type){
    // function body.....
}
```

The declaration(defining) of the function contains:

1. ***func***: It is a keyword in Go language, which is used to tell Go that a function is being declared.
2. **function_name**: It is the name of the function.
3. **Parameter-list**: It contains the name and the type of the function parameters(the values that the function will use).
4. **Return_type**: It is optional and it contains the types of the values that function returns. If you are using return_type in your function, then it is necessary to use a return statement in your function.

After declaring a function, we need to call the function wherever we want in the code, but after declaration of function. To call a function:

```
function_name(parameter-list)
```

Let's make a function to find the area of a rectangle and return the area.

```
package main
import "fmt"

func area(length, width int)int{
area := length * width
return area
}

func main() {
fmt.Printf("Area of rectangle is : %d", area(2, 10))
}
```

The *return* keyword is used to return a value. So, when the function area is called, the statement *area*(2, 10) gets replaced by the value returned by the *area* function.

Hurray!! The course is completed

Exercises:

1. Make a simple Go calculator that can add, subtract, multiply and divide 2 user inputted numbers.
   Click here to view code for the same.
2. Take 2 numbers as input from the user and swap those numbers. Make a swap function to do so.
   Click here to view code for the same.
3. Make an area and circumference calculator.
4. Use a for loop to make a number printer, which takes an integer as input and prints the numbers from 0 to the input number.
   Click here to view code for the same.