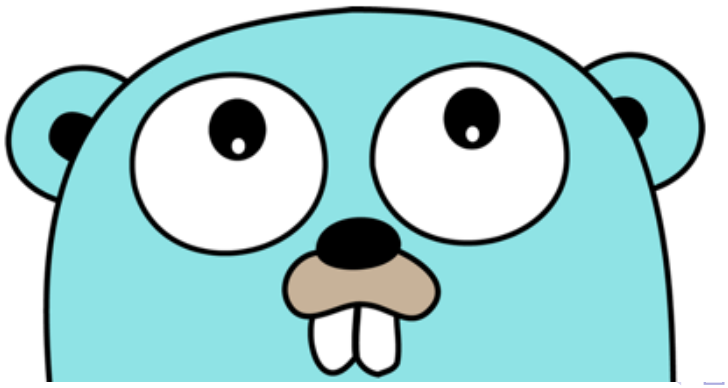
Let's Go!

Mansi Sharma

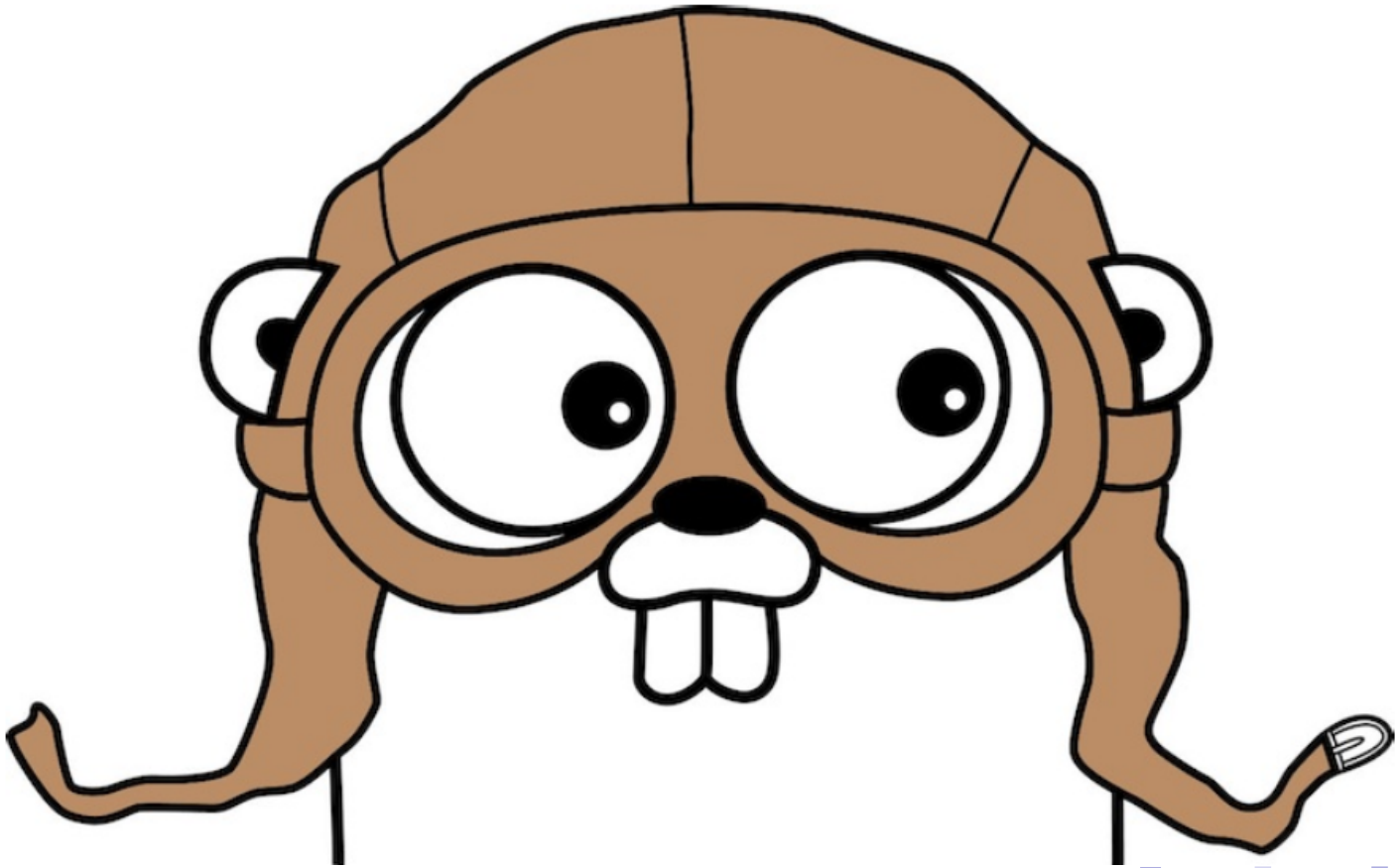
January 20, 2021



What is Go?

Go is a compiled, concurrent, garbage-collector, statically typed language developed at

see a bit of code!



Hello, World!

- Programs start running in package main.
- It is good style to use the factored import statement.
- A name is exported if it begins with a capital letter.

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  // import "fmt"
9  // import "math"
10
11 func main() {
12     fmt.Printf("Now you have %v problems.\n", math.Sqrt(7))
13 }
14
```

Variables

- The var instruction declares a list of variables
- The type is informed at the end
- The var instruction could includes initializers, 1 per variable. In this case, the type could be ommited

```
$ go run variables.go
0 false false false
```

variables.go

```go
package main

import "fmt"

var c, python, java bool

func main() {
    var i int
    fmt.Println(i, c, python, java)
}
```

```
$ go run variables-with-initiali
1 2 true false no!
```

variables-with-initializers.go

```go
package main

import "fmt"

var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"
    fmt.Println(i, j, c, python, java)
}
```

Short Variable Declarations

- Inside a function, the short attribution instruction := can be used instead of a var declaration

```
$ go run short-variable-declarations.go
1 2 3 true false no!
```

short-variable-declarations.go

```
1  package main
2
3  import "fmt"
4
5  i, j := 1, 2
6  func main() {
7      k := 3
8      c, python, java := true, false, "no!"
9
10     fmt.Println(i, j, k, c, python, java)
11 }
12
```

Constants

- Constants are declared like variables but with keyword const
- Can not use the syntx :=

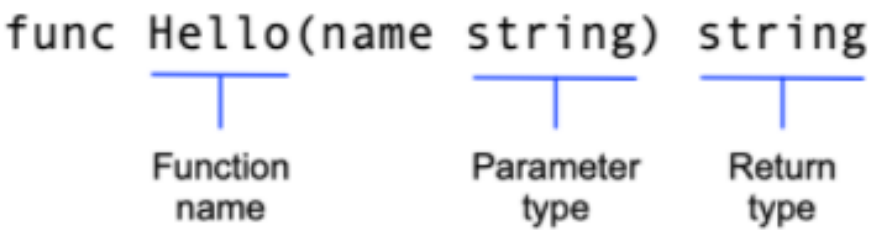
```
$ go run constants.go
Hello world! Happy 3.14 Day! Go rules?
```

constants.go

```go
package main

import "fmt"
const Pi = 3.14

func main() {
    const World = "world! "
    fmt.Print("Hello ", World)
    fmt.Print("Happy ", Pi, " Day! ")

    const Truth = true
    fmt.Print("Go rules? ", Truth)
}
```

Functions

- Type comes after the parameter name, like variables
- Shorten (x int, y int) to (x, y int)




```
$ go run functions.go
55
```

functions.go

```
1  package main
2
3  import "fmt"
4
5  func add(x int, y int) int {
6      return x + y
7  }
8
9  func main() {
10     fmt.Println(add(42, 13))
11 }
12
```

Multiple Return Values

- A function can have multiple return values

```
$ go run multiple-results.go
world hello
```

multiple-results.go

```go
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}

```

Looping For

- Go has only one looping construct, the for loop
- No parentheses required, braces are always required
- The init and post statements are optional

```
$ go run for.go
45
```

for.go

```go
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

```
$ go run for-continu
1024
```

for-continued.go

```go
package main

import "fmt"

func main() {
    sum := 1
    for ; sum < 1000; {
        sum += sum
    }
    fmt.Println(sum)
}
```

For is Go's "while" and forever

- Semicolon can be removed and you will have while
- for can run forever

```
$ go run for-is-go-while.go
1024
```

for-is-gos-while.go

```go
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum)
}
```

```
$ go run forever.go
process took too long
```

forever.go

```go
package main

func main() {
    for {
    }
}
```

if Condition

- No parentheses required, braces are always required

if.go

```go
package main

import (
    "fmt"
    "math"
)

func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}

func main() {
    fmt.Println(sqrt(2), sqrt(-4))
}

```

if-with-a-short-statement.go

```go
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}

```

Switch

- only runs the selected case, not all the cases that follow
- break statement is not required

switch-evaluation-order.go

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Println("Today.")
    case today + 1:
        fmt.Println("Tomorrow.")
    case today + 2:
        fmt.Println("In two days.")
    default:
        fmt.Println("Too far away.")
    }
}

```

switch-with-no-condition.go

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}

```

Defer

- Semicolon can be removed and you will have while
- for can run forever

```
$ go run for-is-go-while.go
1024
```

for-is-gos-while.go

```go
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum)
}
```

```
$ go run forever.go
process took too long
```

forever.go

```go
package main

func main() {
    for {
    }
}
```

Stacking Defer

- Deferred function calls are pushed onto a stack

defer-multi.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("counting")
7
8      for i := 0; i < 10; i++ {
9          defer fmt.Println(i)
10     }
11
12     fmt.Println("done")
13 }
14
```

```
counting
done
9
8
7
6
5
4
3
2
1
0
```

Pointers

- Synatx for pointers is the same as in C or C++
- But unlike C, Go has no pointer arithmetic.

pointers.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      i, j := 42, 2701
7
8      p := &i          // point to i
9      fmt.Println(*p) // read i through the pointer
10     *p = 21          // set i through the pointer
11     fmt.Println(i)  // see the new value of i
12
13     p = &j          // point to j
14     *p = *p / 37    // divide j through the pointer
15     fmt.Println(j) // see the new value of j
16 }
17
```

Structs

- A struct is a collection of fields
- Struct fields are accessed using a dot
- Struct fields can be accessed through a struct pointer
- You can list just a subset of fields by using the Name: syntax
- The special prefix & returns a pointer to the struct value

Structs

struct-literals.go

```
1  package main
2
3  import "fmt"
4
5  type Vertex struct {
6      X, Y int
7  }
8
9  var (
10     v1 = Vertex{1, 2}  // has type Vertex
11     v2 = Vertex{X: 1}  // Y:0 is implicit
12     v3 = Vertex{}      // X:0 and Y:0
13     p  = &Vertex{1, 2} // has type *Vertex
14 )
15
16 func main() {
17     v1.X = 3
18     q := &v1
19     q.Y = 5
20     fmt.Println(v1, p, v2, v3)
21 }
22
```

```
{3 5} &{1 2} {1 0} {0 0}

Program exited.
```

Arrays

- The type [n]T is an array of n values of type T
- An array's length is part of its type, so arrays cannot be resized

array.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var a [2]string
7      a[0] = "Hello"
8      a[1] = "World"
9      fmt.Println(a[0], a[1])
10     fmt.Println(a)
11
12     primes := [6]int{2, 3, 5, 7, 11, 13}
13     fmt.Println(primes)
14 }
15
```

```
Hello World
[Hello World]
[2 3 5 7 11 13]

Program exited.
```

Slicing

- Slicing of arrays is done the same way as in Python
- Slices are like references to arrays

slices-pointers.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      names := [4]string{
7          "John",
8          "Paul",
9          "George",
10         "Ringo",
11     }
12     fmt.Println(names)
13
14     a := names[0:2]
15     b := names[1:3]
16     fmt.Println(a, b)
17
18     b[0] = "XXX"
19     fmt.Println(a, b)
20     fmt.Println(names)
21 }
```

```
[John Paul George Ringo]
[John Paul] [Paul George]
[John XXX] [XXX George]
[John XXX George Ringo]

Program exited.
```

Slice Literals Length and Capacity

- A slice literal is like an array literal without the length
- The length and capacity of a slice s can be obtained using the expressions len(s) and cap(s)
- There are also many more built-in functions like make and append that can be used with slices

Slice Literals Length and Capacity

slice-len-cap.go

```go
package main

import "fmt"

func main() {
    s := []int{2, 3, 5, 7, 11, 13}
    printSlice(s)

    // Slice the slice to give it zero length.
    s = s[:0]
    printSlice(s)

    // Extend its length.
    s = s[:4]
    printSlice(s)

    // Drop its first two values.
    s = s[2:]
    printSlice(s)
}

func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

```
len=6 cap=6 [2 3 5 7 11 13]
len=0 cap=6 []
len=4 cap=6 [2 3 5 7]
len=2 cap=4 [5 7]

Program exited.
```

Range

- The range form of the for loop iterates over a slice or map
- two values are returned for each iteration, the index and the element at that index
- If you only want the index, you can omit the second variable

Range

range-continued.go

```go
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}

```

```
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
2**8 = 256
2**9 = 512

Program exited.
```


Questions?