in the name of Allah

Let's GO

Seyed Mahdi Mahdavi Mortazavi

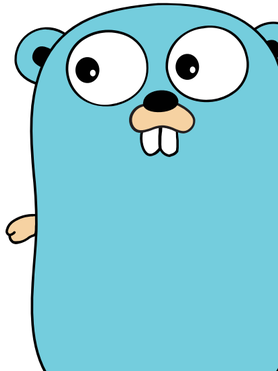
Mohammad Mahdi Zamani

Ali Shojaei

1) Introduction & History

Background and Motivation

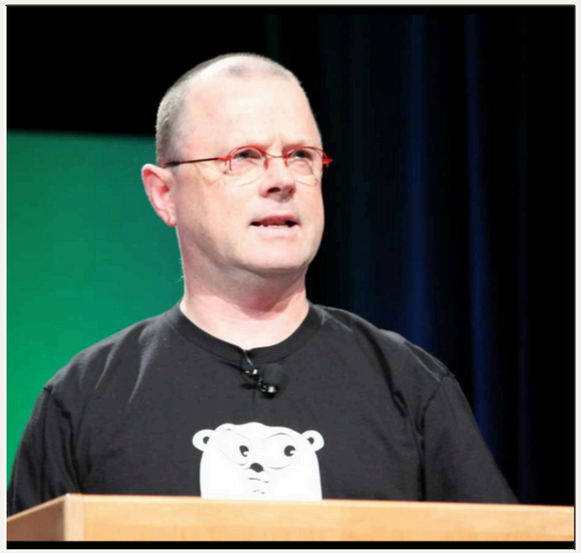
- *In the **mid-2000s**, engineers at Google—**Robert Griesemer**, **Rob Pike**, and **Ken Thompson**—were frustrated with the limitations of existing programming languages when working on large-scale, complex systems. Languages like C++ offered high performance but were overly complex and slow to compile. On the other hand, dynamic languages like Python were easy to write but lacked performance and type safety for large infrastructure.*



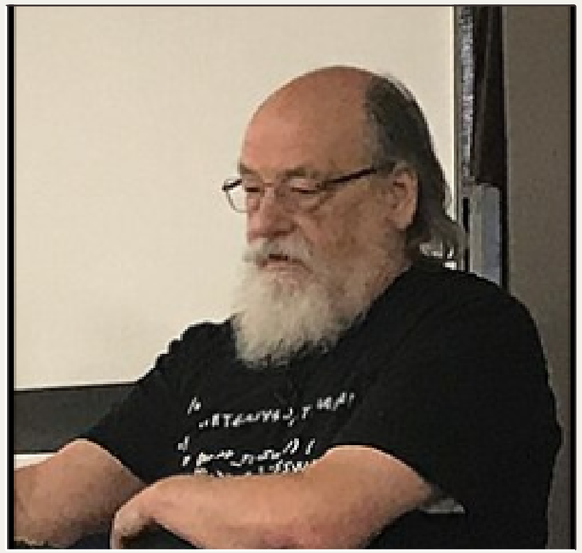
1) Introduction & History

Background and Motivation - Creators

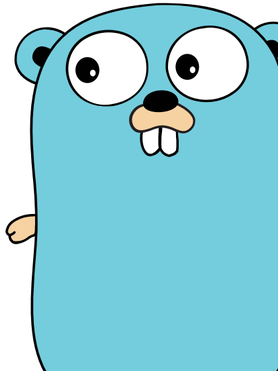
Robert Griesemer



Rob Pike



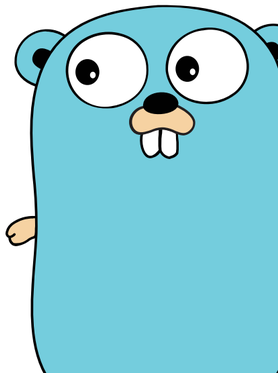
Ken Thompson



1) Introduction & History

Background and Motivation

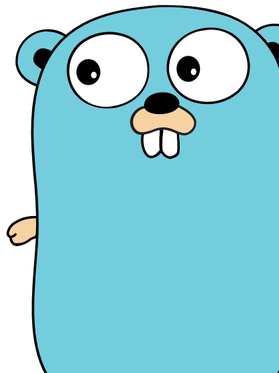
- The team wanted a language that combined (inspired by):
  - The ***efficiency and speed of C/C++***
  - ***Pascal***, and ***Modula*** (for ***clarity***)
  - The ***simplicity and ease of use*** found in ***Python*** or ***JavaScript***
  - ***Better support for concurrency***, which is essential in modern ***multi-core processors***



1) Introduction & History

Background and Motivation

- Continue: This led to the ***creation of Go in 2007***, with its first ***public release in 2009***. The ***goal*** was to build a language that is:
  - ***Fast to compile***
  - ***Simple to learn and read***
  - Powerful ***enough for*** building ***large***, ***scalable systems—especially for cloud*** infrastructure, ***networking***, and ***backend services***
    - Go has since become ***widely adopted in cloud-native technologies*** (e.g., ***Docker***, ***Kubernetes***) and by companies needing ***reliable***, ***high-performance*** backend services.
  - ***Built-in*** support for ***concurrent programming*** (via ***goroutines***)

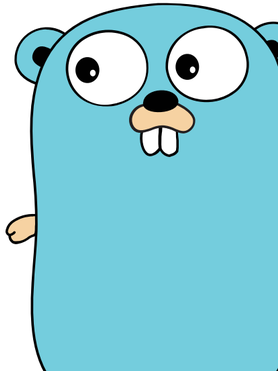


1) Introduction & History

Background and Motivation

- Go is a ***compiled language***, producing fast, standalone binary files with ***no external dependencies***.
- Common development environments include:
  - ***Visual Studio Code*** (with Go extension)
  - ***GoLand*** (a powerful IDE by JetBrains)
  - Command-line tools like **go run**, **go build**, and **go test** are widely used for development and testing.

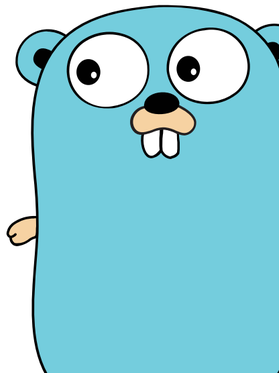
GoLand logo



2) Primitive data types

Data types

- bool
- string
- ***Integers***: int, int8, int16, int32, int64
- ***Unsigned integers***: uint, uint8, uint16, uint32, uint64, uintptr (unicode integer pointer)
    - Signed integers and unsigned integers have the same size.
- ***Unicode letters***: byte (aliases for uint8), rune (int32)
- ***Float numbers***: float32, float64
- ***Complex numbers***: complex64, complex128



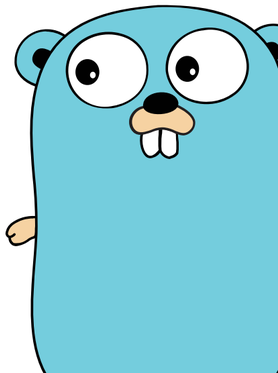
2) Primitive Data Types

Data types

Output:

```
true
Hello, world!
42
3.14
(1+2i)
```

```go
package main

import "fmt"

func main() {
  var b bool = true
  s := "Hello, world!"
  var i int = 42
  f := 3.14
  var c complex128 = 1 + 2i

  fmt.Println(b)
  fmt.Println(s)
  fmt.Println(i)
  fmt.Println(f)
  fmt.Println(c)
}
```



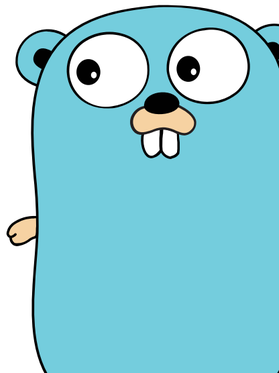
3) Type Binding

Static type binding

- Go uses ***static*** type binding.
- The type of a variable is determined at ***compile time***, not at runtime. Once a variable is declared with a type, that ***type cannot change during execution***.

```go
func main() {
  var x int = 10
  x = "hello" // compile-time error: cannot use string as int
}
```

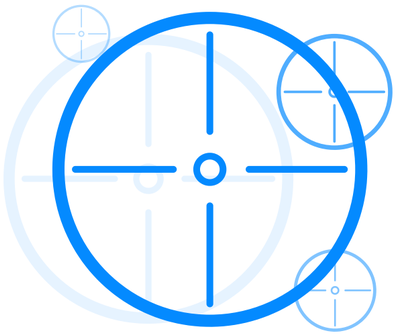
```
./main.go:5:6: declared and not used: x
./main.go:6:7: cannot use "hello" (untyped string constant) as int value in assignment
```



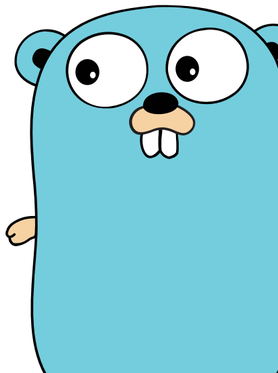
4) Scoping

Static scoping

- Go uses ***static scoping***, and the type of variables is determined using local scope, ancestor blocks, and global blocks.



```go
package main

import "fmt"

var x = 100

func printX() {
    fmt.Println(x) // Output: 100
}

func main() {
    x := 50
    fmt.Println(x) // Output: 50
    printX()
}
```

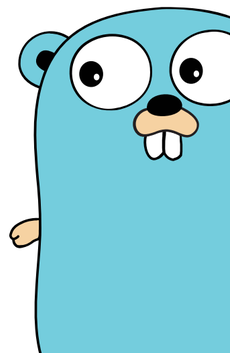


5) Enumeration

No enumeration

- Go ***doesn't have a built-in enum*** keyword like some other languages (e.g., C, Java). But you can create enumeration-like behavior using ***const*** and the ***iota*** identifier.
  - ***iota*** is a Go keyword that ***auto-increments starting from 0***.
  - It's **commonly used with const** to define a set of related constant values.

```go
package main

import "fmt"

const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)

func main() {
    fmt.Println("Tuesday is:", Tuesday)
    // Output: Tuesday is: 2
}
```

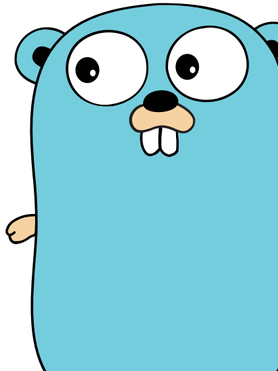


6) Associative Array (Dictionary)

Dictionary

- In Go, an associative array (often called a dictionary in other languages) is implemented using the ***built-in map type***.
- A ***map*** lets you associate keys of one type with values of another (or the same) type.

```go
func main() {
    var population map[string]int
    fmt.Println(population["Tehran"]) // output: 0

    population["Tehran"] = 9_039_000
    // panic: assignment to entry in nil map
```



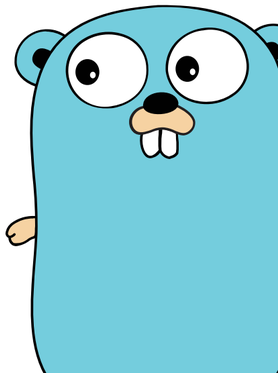
6) Associative Array (Dictionary)

Dictionary

```go
func main() {
  var population map[string]int
  fmt.Println(population)

  population = make(map[string]int)

  population["Tehran"] = 9_039_000
  population["Shiraz"] = 1_955_500
  fmt.Println("Shiraz population is", population["Shiraz"])

  delete(population, "Shiraz")
  fmt.Println(population)
}
```

Output:

```
map[]
Shiraz population is 1955500
map[Tehran:9039000]
```

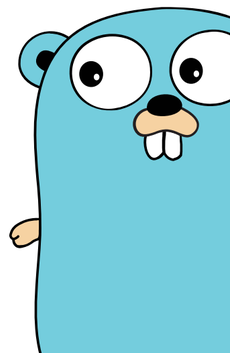


7) Union

No union

- Go has ***no built-in union type***; you typically use the empty interface or a tagged struct to hold multiple possible types, and it is ***not discriminated*** by the language (you must distinguish cases yourself).

```go
package main

import "fmt"

func main() {
    var virtual_union interface{} = 88
    // could also be a string, etc.

    switch union_val := virtual_union.(type)
    case int:
        fmt.Println("int:", union_val)
    case string:
        fmt.Println("string:", union_val)
    default:
        fmt.Println("unknown type")
    }
}
```

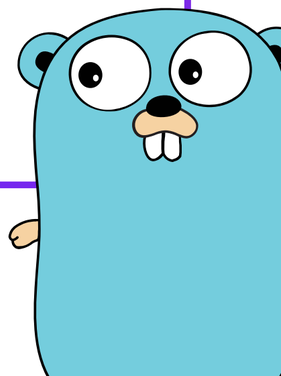


8) Dangling Pointers

You will not encounter🙂✅

- ***Escape Analysis***:
  - Go's compiler analyzes whether a variable's address will "escape" the current function or goroutine. If it does—meaning the ***address is returned or stored somewhere outside the function***—Go automatically ***allocates it on the heap*** instead of the stack.

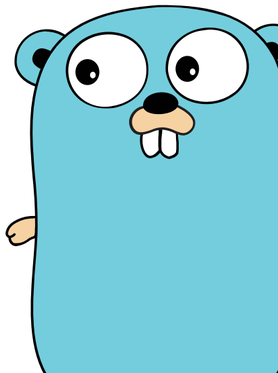
```go
package main

import "fmt"

func createPointer() *int {
    x := 10   // Normally stack-allocated
    return &x // Escapes → moved to heap
}

func main() {
  fmt.Println(createPointer())
  // output: 0xc00009a040
}
```



8) Dangling Pointers

You will not encounter🙂✅

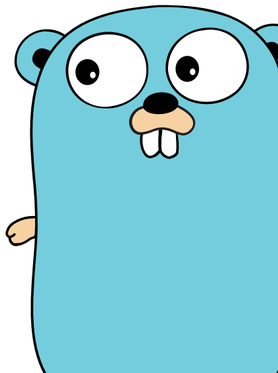
- ***Garbage Collector (GC)***:
  - Go uses a *concurrent garbage collector*, and it's based on a ***Mark-and-Sweep*** approach:
    - The GC tracks which variables are still "reachable" from active parts of your code.
    - As long as a pointer to an object exists and is reachable, that ***object is not collected***, even if it's no longer explicitly used.
    - This guarantees that you won't get a dangling pointer to memory that's already been freed.
    - Requires ***no manual memory management*** (free() or delete).



9) Garbage Collection

Automatic garbage collection

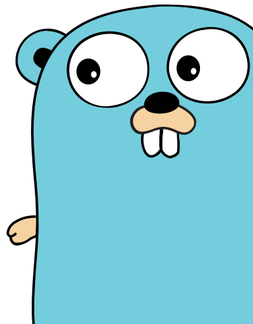
- Go uses a ***concurrent***, ***automatic*** garbage collector to free memory that is no longer reachable by your program.
  - You ***don’t need to free memory*** manually.
  - When there are ***no references to a value*** (i.e. it's unreachable), the ***GC will clean it up*** in the background.
  - ***Mark and Sweep***:
    - ***Mark***: Find all reachable objects starting from root variables.
    - ***Sweep***: Reclaim memory from unreachable (unmarked) objects.
    - ***Runs automatically***: You don’t need to trigger it.
      - It can be triggered manually (not recommended):
        - ***runtime.GC()***



9) Garbage Collection

Automatic garbage collection

```
[Root Variables]
        |
        v
  (MARK) → live object → more pointers → mark those
        |
   Unreachable objects = not marked
        |
        ↓
  (SWEEP) → free unmarked memory

```

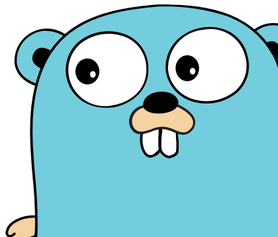


9) Garbage Collection

Automatic garbage collection

```go
type Person struct {
    Name string
}

func createPerson() *Person {
    p := &Person{Name: "Alice"}
    return p
}

func main() {
    p := createPerson()
    fmt.Println(p.Name) // output: Alice

    p = nil // no more references to "Alice"
    // GC may now reclaim the memory used by the Person
}
```

Garbage Collector



10) Function & Procedure

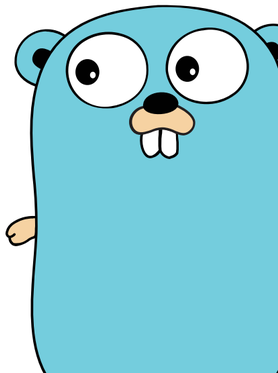
Function and Procedure Definition

- In Go, you define a function using the ***func*** keyword. There is ***no separate "procedure"*** type—a function that ***returns nothing*** simply behaves ***like a procedure***.

```go
// Function without a return value
func sayHi(name string) {
    fmt.Println("Let's", name)
}

// Function with a return value
func square(x int) int {
    return x * x
}

// Function with two return values
func divide(a, b int) (int, int) {
    return a / b, a % b
}

func main() {
    sayHi("GO!")
    sqr5 := square(5)
    div, remainder := divide(5, 2)
    fmt.Println(sqr5, " --- ", div, remainder)
}
```

Output:

```
Let's GO!
25  ---  2 1
```

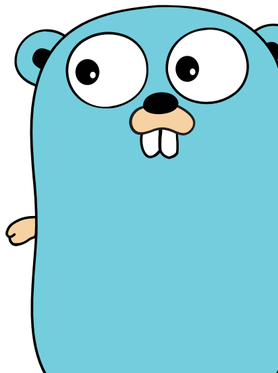


11) Keyword/Positional Parameters

Only Positional Parameters!

- Go does ***not support keyword*** arguments.
- All function parameters are passed by position.

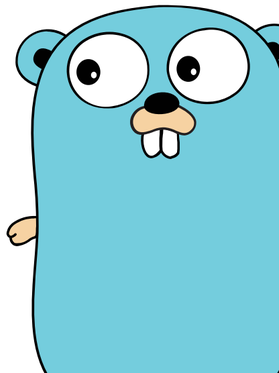
```go
package main
import "fmt"

func greet(name string, age int) {
    fmt.Printf("Hello %s, you are %d years old\n", name, age)
}

func main() {
    greet("Alice", 30) // Positional: first is name, second is age
}

```



12) Default Parameter Values

Go does not support default parameter values!

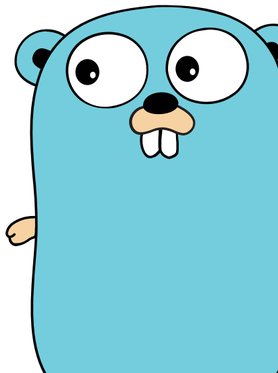
```go
package main
import "fmt"

func greet(name string, age int) {
    fmt.Printf("Hello %s, age %d\n", name, age)
}

func main() {
    greet("Ali", 25)
}

```



13) Variadic Parameters

ellipsis(...) allows passing a variable number of arguments

```go
package main
import "fmt"

func sum(numbers ...int) int {
    total := 0
    for _, n := range numbers {
        total += n
    }
    return total
}

func main() {
    fmt.Println(sum(1, 2, 3))          // 6
    fmt.Println(sum(4, 5, 6, 7, 8))    // 30
}
```

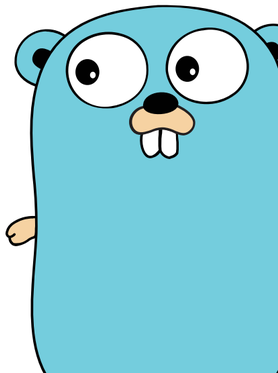


14) Call by Value

Go is call-by-value

- To ***simulate call-by-reference***, use ***pointers***:

```
4  func byValue(x int) {
5      x = x + 1
6  }
7
8  func byReference(x *int) {
9      *x = *x + 1
10 }
11
12 func main() {
13     a := 10
14     byValue(a)
15     fmt.Println("After byValue:", a) // 10 (unchanged)
16
17     byReference(&a)
18     fmt.Println("After byReference:", a) // 11 (changed)
19 }
```

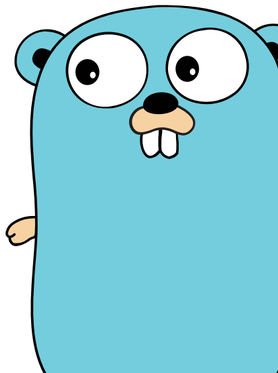


15) Function as Parameter

Go supports functions as parameter

- supports ***first-class*** functions

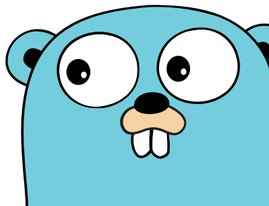

```go
package main
import "fmt"

func apply(f func(int) int, x int) int {
    return f(x)
}

func main() {
    double := func(n int) int {
        return n * 2
    }
    result := apply(double, 5)
    fmt.Println(result) // 10
}
```



16) Function Overloading

Go does not support function overloading

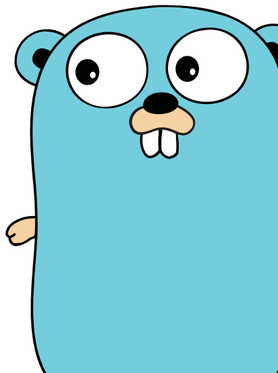
```go
// Invalid in Go – duplicate function name
func show(x int) {}
func show(x string) {} // ERROR: redeclared
```



17) Closure

Go supports closures

```go
package main
import "fmt"

func counter() func() int {
    count := 0
    return func() int {
        count++
        return count
    }
}

func main() {
    next := counter()
    fmt.Println(next()) // 1
    fmt.Println(next()) // 2
}
```



18) Coroutines

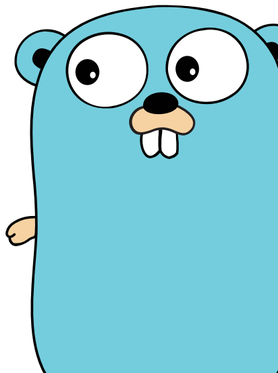
Goroutines in Go

- Go supports ***lightweight coroutines*** using '***go***' keyword:

```go
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  say("hello")
}
```

Output:

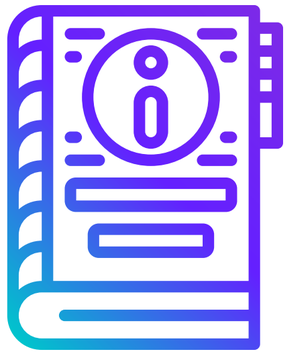
```
hello
world
world
hello
world
hello
hello
world
hello
```

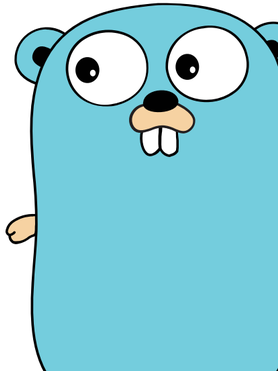


References

Link to resources

- https://go.dev
- https://www.computerhope.com/people/robert_griesemer.htm
- https://en.wikipedia.org/wiki/Rob_Pike
- https://en.wikipedia.org/wiki/Ken_Thompson
- https://medium.com/@ltcong1411/how-does-garbage-collection-work-in-go-how-to-minimize-memory-leaks-in-a-high-load-application-770520467d1c





Thank You!