Golang

An Introduction for Software Engineers

structs, pointers, functions, interfaces, go routines, channels, ...

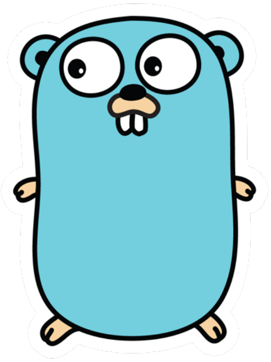
Purpose of this Workshop

- Introduction to the go programming language (golang) for developers familiar with other languages
- Mainly we will discuss tiny fully self-contained coding examples exemplifying language peculiarities, common coding patterns and pitfalls
- Focus on realistic examples and relevant features
- Distilled & compact: get you started quickly and efficiently

A Brief History of Go

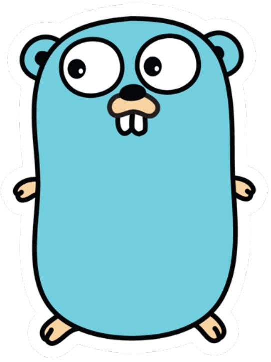
A Brief History of Go

- Invented by Robert Griesemer, Rob Pike, and Ken Thompson at Google.
- The initial design of Go began in September 2007, it was officially announced in November 2009 and version 1.0 was released in 2012. As of 2023, latest version is 1.21.4.
- The language was created to address issues related to scalability and efficiency in Google's large software systems.
- Designed as a replacement for C++ and Java and to address efficient compilation, efficient execution and ease of programming.
- Multi-threading as core tenet (goroutines and channels).
- Simple and stable syntax.



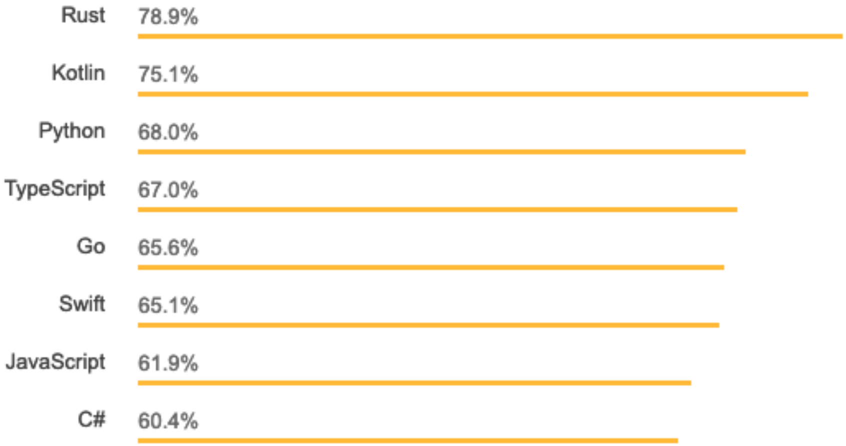
What Go Is Good For – My Opinion

- Good for both small and large projects
- Modernized ANSI C
- Easy to transition, especially if you know a little about C (structs, pointers)
- Excellent choice for server side programming with superior support for concurrency built in

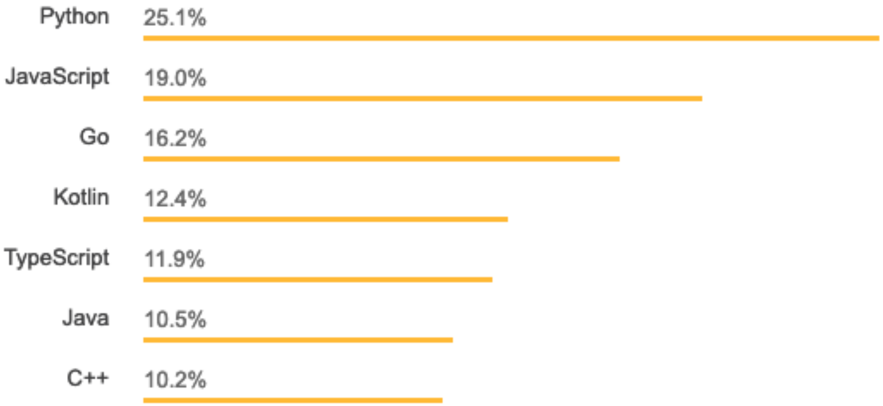


Stack Overflow Stats

Most Loved Languages



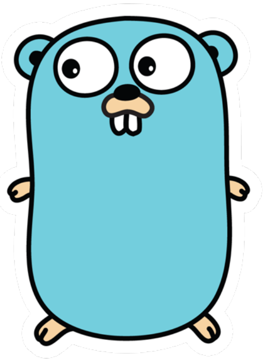
Most Wanted Languages



Golang Basics

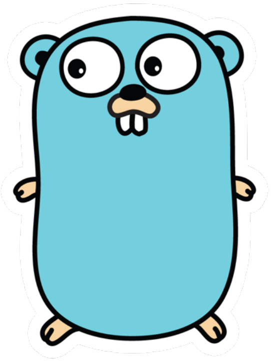
Core Language Features – The Haves

- compiled
- strongly typed
- garbage collected
- statically linked runtime
- simplified C-style pointers
- conventions
  - tests, exported functions, init() function etc.
- strong support for concurrency built in
  - ***channels*** and ***goroutines*** (light-weight threads) as first class concept, suited fo multi-core architectures and event driven asynchronous services



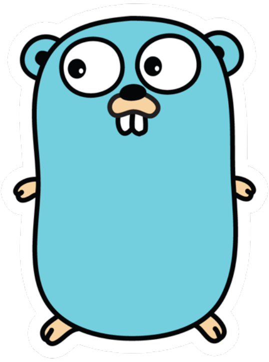
Core Language Features – The Have Nots

- no objects, no inheritance, no method overriding
  - instead: structs and composition, pointer receivers
- no function overloading ☹
- no exception handling (error type instead)

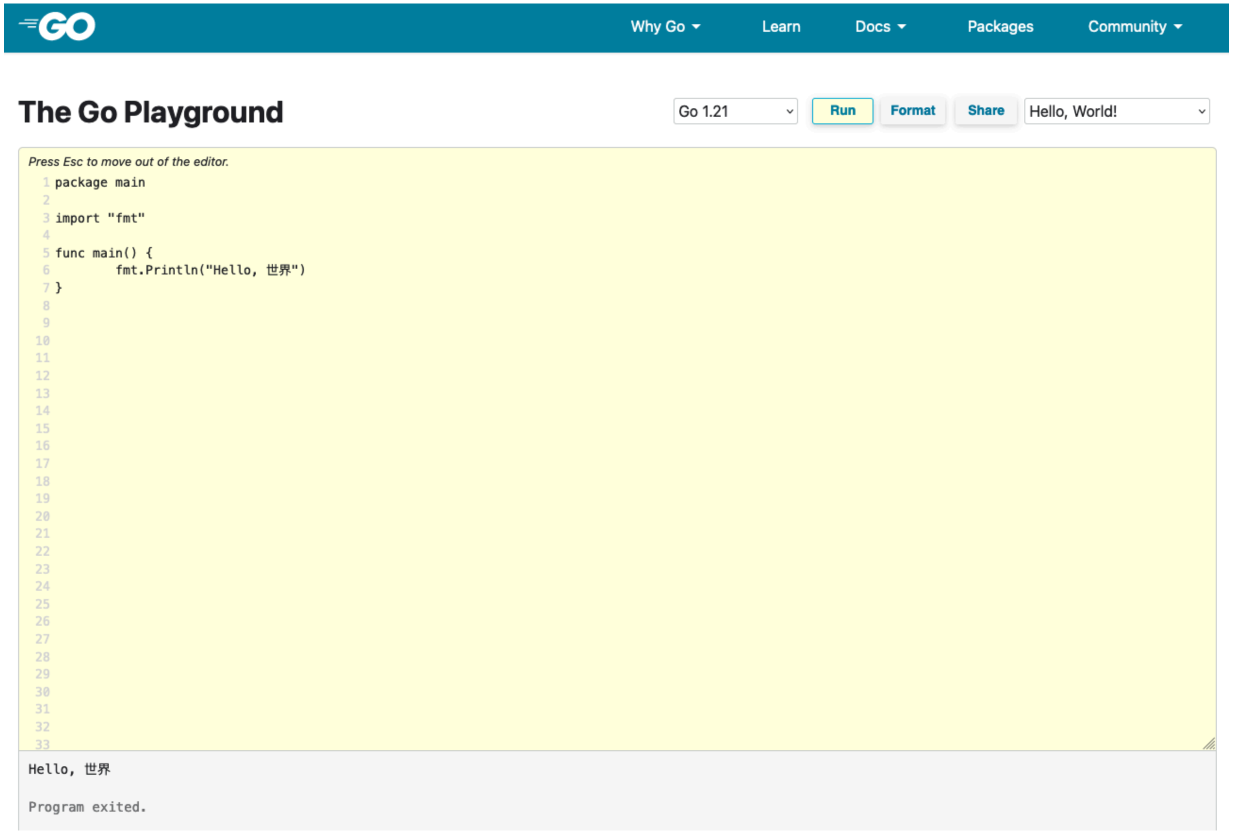


Hello World or 你好世界

```go
//
// Golang Workshop 2024
//

package main

import "fmt"

func main() {
    fmt.Println("你好世界")
}
```



The Basics

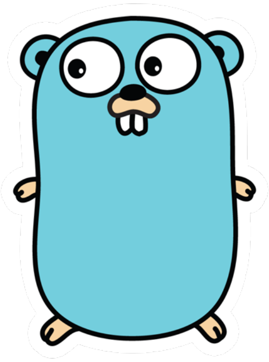


The Go Playground

Press Esc to move out of the editor.

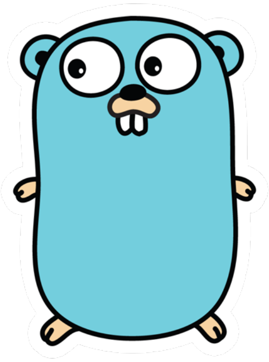
```go
package main

import "fmt"

func main() {
	fmt.Println("Hello, 世界")
}
```

```
Hello, 世界

Program exited.
```



Hello World Online

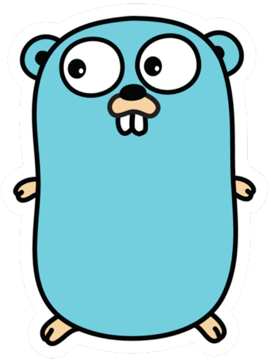
```
package main

import (
    "fmt"
    "net/http"
)

func index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "<html><body><p>你好世界</p></body></html>")
}

func main() {
    http.HandleFunc("/", index)
    http.ListenAndServe(":8080", nil)
}
```



Packages and Visibility

convention: everything uppercase is exported, everything lowercase is private

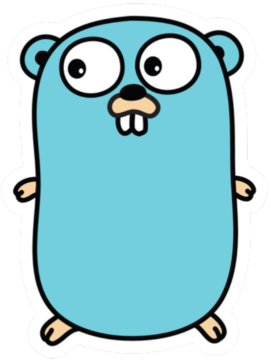
```go
//
// Golang Workshop 2024
//

package foo

var (
    ExportedVar int = 42
    privateVar  int = 76
)

func ExportedFunction() string {
    return "hello world!!!"
}

func privateFunction() string {
    return "very private!!!"
}
```



Packages and Visibility

convention: everything uppercase is exported, everything lowercase is private

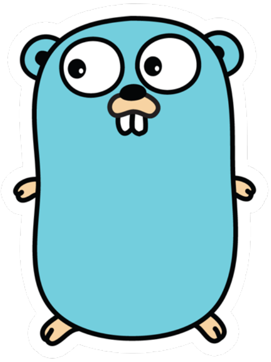
```
//
// Golang Workshop 2024
//

package main

import (
    "fmt"
    “github.com/boriwo/golang/packages/foo"
)

func main() {
    fmt.Println(foo.ExportedFunction())
    fmt.Println(foo.ExportedVar)
}
```



Packages and Visibility

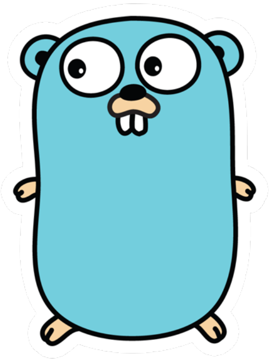
convention: everything uppercase is exported, everything lowercase is private

```
//
// Golang Workshop 2024
//

package main

import (
    "fmt"
    bar “github.com/boriwo/golang/packages/foo"
)

func main() {
    fmt.Println(bar.ExportedFunction())
    fmt.Println(bar.ExportedVar)
}
```



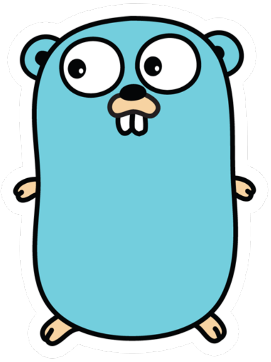
Simplicity: Implicit Variable Declarations

```
// variable declaration and initialization in one line

var a int = 42

// or even shorter

b := 84
```



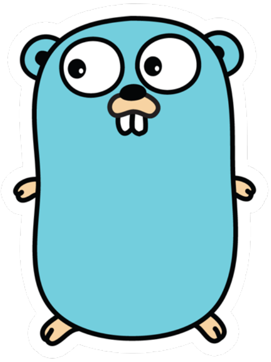
Simplicity: No Parentheses, No Semicolons

```
// no parentheses in control structures

if 1 == 1 {
   fmt.Println("one is still one")
}


// implicit variable definitions and C-inspired syntax

for i := 1; i <= 3; i++ {
   fmt.Printf("i have been here %d times\n", i)
}
```



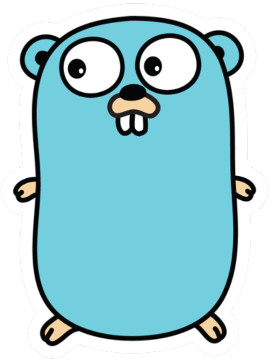
Infinite Loop

```
i := 1
for {
    if i > 3 {
        break
    }
    fmt.Printf("i have been here %d times\n", i)
    i++
}
```



Switch Statement without break

```
v := "foo"

switch v {
case "foo":
    fmt.Println("v is foo")
case "bar":
    fmt.Println("v is bar")
default:
    fmt.Println("v is none of the above")
}
```



Maps

```
// allocate map with make()

m1 := make(map[string]int)

// ...or initialize map in one line

m2 := map[string]int{
   "one":   1,
   "two":   2,
   "three": 3,
}

// read and write elements

m1["one"] = 1
val := m2["two"]
fmt.Printf("map element two is %d\n", val)
```

Maps

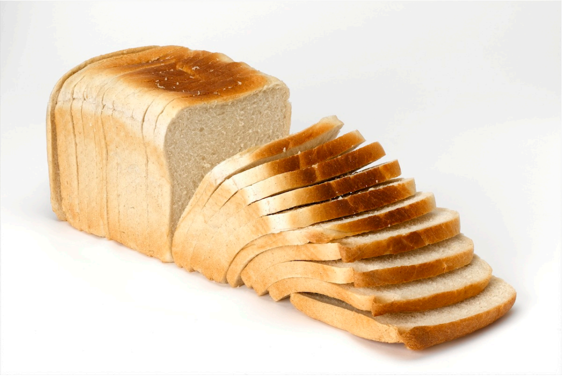
```
// remove element with delete()

delete(m1, "one")

// check length with len

if len(m2) > 2 {
   fmt.Println("m2 has more than 2 elements")
}
```

Maps

```
// iterate over all elements

for k, v := range m2 {
   fmt.Printf("map element %s is %d\n", k, v)
}

// comma ok idiom to check for presence of values

val := m2["two"]

val, ok := m2["two"]

if ok {
   fmt.Printf("map element is present, value is %d\n", val)
} else {
   fmt.Printf("map element is not present, value is %d\n", val)
}
```

Slices

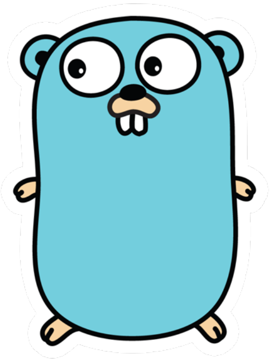
```
// slices are arrays that can dynamically change size

slice := []int{4, 5, 6, 7}

slice2 := make([]int, 4)

slice[0] = 3

// add elements with append(), must reassign reference because it may change if memory has to be reallocated

slice = append(slice, 8)

// iterate of slice

for idx, val := range slice {
   fmt.Printf("element %d has value %d\n", idx, val)
}

// use len to determine number of elements

if len(slice) > 3 {
   fmt.Println("slice has more than 3 elements")
}
```



Functions, Pointers and OOP

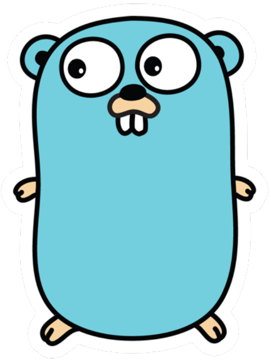
Functions with Multiple Return Values

```go
// functions can have more than one return value, useful for error
// handling and many other things

func Sqrt(f float64) (float64, error) {
   if f < 0 {
      return 0, errors.New("negative numbers not allowed")
   }
   return math.Sqrt(f), nil
}
```



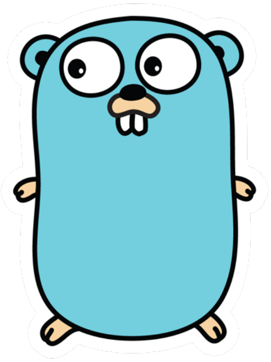
Functions with Multiple Return Values

```
// when calling the function, evaluate error…

root, err := Sqrt(-64.0)

if err != nil {
    fmt.Printf("%s\n", err)
} else {
    fmt.Printf("square root of %f is %f\n", num, root)
}

// ...or if you are brave, ignore the error

root, _ := Sqrt(num)
```



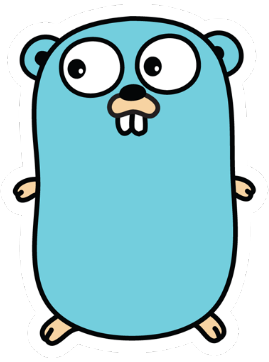
Variadic Functions

```go
func sum(num ...int) int {
    sum := 0
    for _, n := range num {
       sum += n
    }
    return sum
}

func main() {
    s := sum(1, 2, 3)
    fmt.Printf("the some of 1, 2, 3 is %d\n", s)
}
```



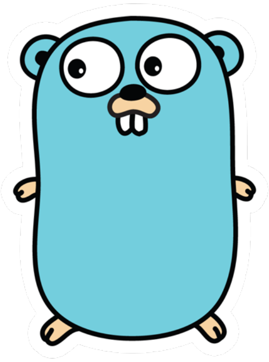
Anonymous Functions

```go
func main() {
    f := func(num ...int) int {
        sum := 0
        for _, n := range num {
            sum += n
        }
        return sum
    }
    s := f(1, 2, 3)
    fmt.Printf("the some of 1, 2, 3 is %d\n", s)
}
```



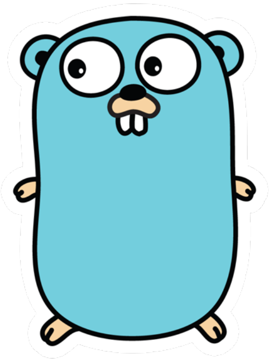
Anonymous Functions

```
func main() {
    s := func(num ...int) int {
        sum := 0
        for _, n := range num {
            sum += n
        }
        return sum
    }(1, 2, 3)
    fmt.Printf("the sum of 1, 2, 3 is %d\n", s)
}
```



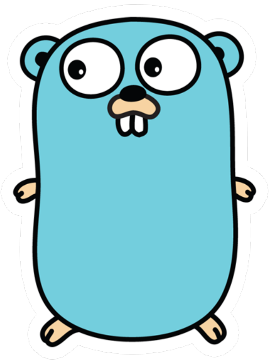
Functions and `defer`

```go
func doWork() error {
    file, err := os.OpenFile("example.txt", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
        return err
    }
    data := map[string]string{"Foo": "Bar", "Hello": "World"}
    buf, err := json.MarshalIndent(data, "", "\t")
    if err != nil {
        file.Close()
        return err
    }
    _, err := file.Write(buf)
    if err != nil {
        file.Close()
        return err
    }
    file.Close()
    return nil
}
```



Functions and `defer`

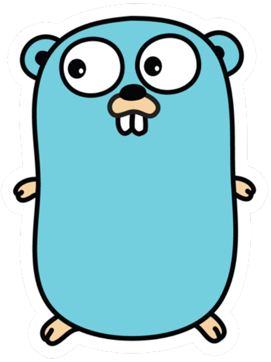
```go
func doWork() error {
    file, err := os.OpenFile("example.txt", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
      return err
    }
    // deferred statement will be executed when surrounding function returns
    // useful for releasing resource when function has many return paths
    defer file.Close()
    data := map[string]string{"Foo": "Bar", "Hello": "World"}
    buf, err := json.MarshalIndent(data, "", "\t")
    if err != nil {
      return err
    }
    _, err := file.Write(buf)
    if err != nil {
      return err
    }
    return nil
}
```



Avoiding deadlocks with `defer`

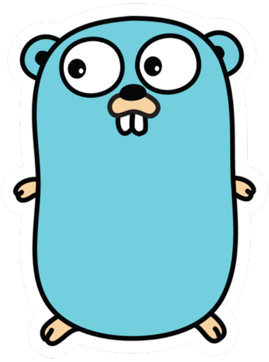
```
func do(m map[string]string, lock sync.RWMutex) error {
   lock.Lock()
   val, ok := m["foo"]
   if !ok {
     lock.Unlock() // this is easily forgotten
     return errors.New("missing key")
   }
   m["foo"] = val + "abc"
   lock.Unlock()
   return nil
}
```

```
func do(m map[string]string, lock sync.RWMutex) error {
   lock.Lock()
   defer lock.Unlock()
   val, ok := m["foo"]
   if !ok {
     return errors.New("missing key")
   }
   m["foo"] = val + "abc"
   return nil
}
```



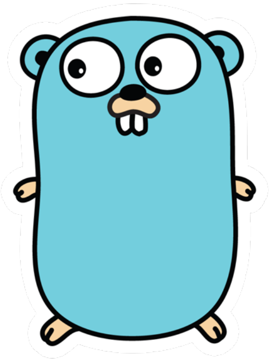
Functional Programming

```
func applyMapper(input []int, mapper func(int) int) []int {
    result := make([]int, len(input))
    for i, v := range input {
        result[i] = mapper(v)
    }
    return result
}


numbers := []int{1, 2, 3, 4, 5}

f := func(x int) int {
    return x * 2
}

doubled := applyMapper(numbers, f)
```



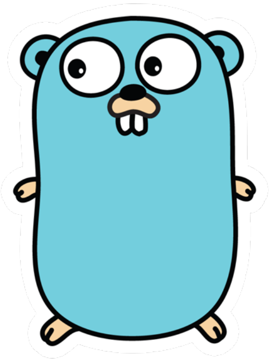
Functional Programming

```
func applyMapper(input []int, mapper func(int) int) []int {
    result := make([]int, len(input))
    for i, v := range input {
        result[i] = mapper(v)
    }
    return result
}


numbers := []int{1, 2, 3, 4, 5}

doubled := applyMapper(numbers, func(x int) int {
    return x * 2
})
```



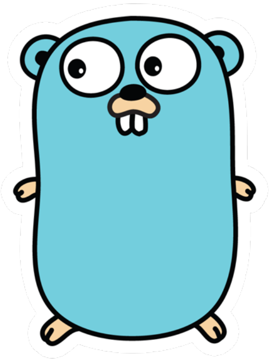
Closures – What’s the Point?

```
sum := 2

func(x int) {
    sum += x
}(3)

fmt.Println(sum)
```



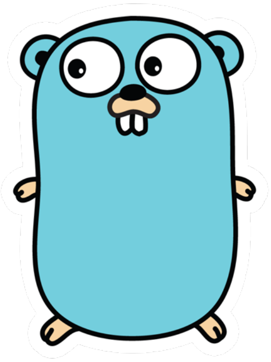
Closures – What’s the Point?

```
sum := 2

sum = func(x int, s int) int {
    s += x
    return s
}(3, sum)

fmt.Println(sum)
```



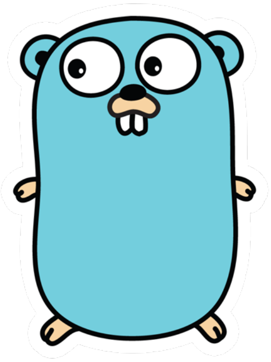
Closures – A Rob Pike Example

```go
package main

import "math"

func Compose(f, g func(x float64) float64) func(x float64) float64 {
    return func(x float64) float64 {
        return f(g(x))
    }
}

func main() {
    print(Compose(math.Sin, math.Cos)(0.5))
}
```



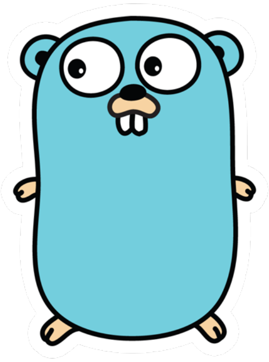
Closures – Rob Pike Example Simplified

```
package main

import "math"

type MathFunc func(x float64) float64

func Compose (f, g MathFunc) MathFunc {
    return func(x float64) float64 {
        return f(g(x))
    }
}


func main() {
    print(Compose(math.Sin, math.Cos)(0.5))
}
```



Pointers in Golang

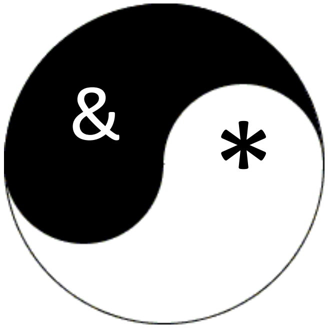
- Pointers are memory addresses
- Pointers can be used as references to variables (pass by reference)
- Use `&` operator to get the address of a variable (“addressing”)
- Use `*` operator to get the variable stored at a memory address (“dereferencing”)
- Uninitialized pointers have the value `nil`
- No pointer needed for reference types
  - Slices and maps are reference types
  - Functions are reference types (functional programming)
  - Channels are reference types



Pass By Reference

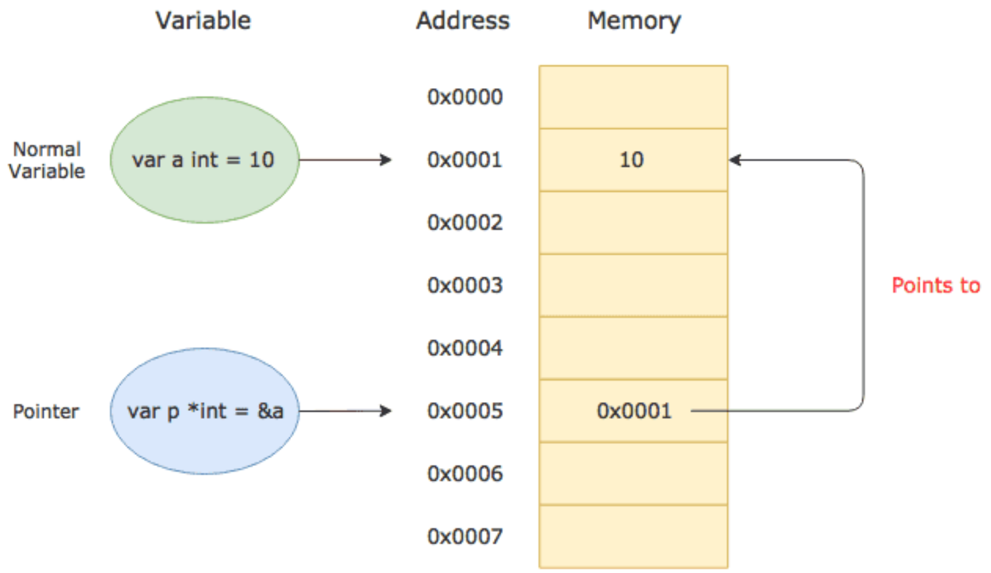
```
func increment(num int) int {
    return num + 1
}

func do() {
    var i int = 10
    i = increment(i)
    fmt.Println(i)
}
```

```
func increment(num *int) {
    *num = *num + 1
}

func do() {
    var i int = 10
    increment(&i)
    fmt.Println(i)
}
```

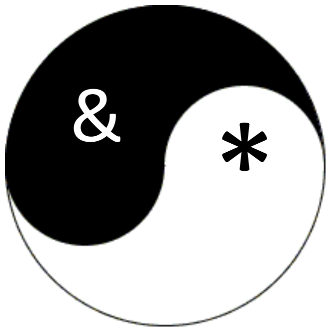


A Pointer is a Memory Addresses

it can be used to reference a variable stored there



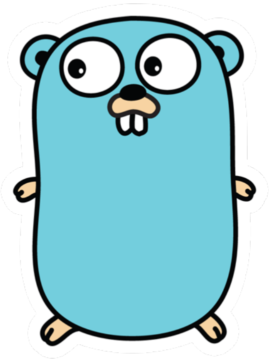
```
var p *int
var a int = 10

p = &a

fmt.Println(*p)

*p = 42

fmt.Println(a)
```



Golang Pointers vs. C Pointers

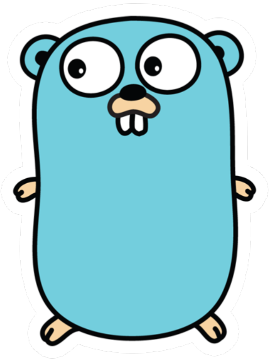
Golang pointers are simpler and safer to use than pointers in C but are therefore also a little less flexible

- No pointer arithmetic (`ptr+1` not allowed)
- No need for `malloc()` and `free()`
- No type casting of pointers
- No buffer overflows or memory leaks



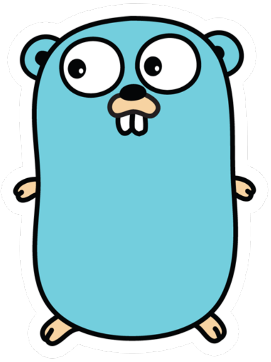
OOP in Golang

- Encapsulation with structs and composition
- Polymorphism with interfaces



Structs are Objects (kind of)

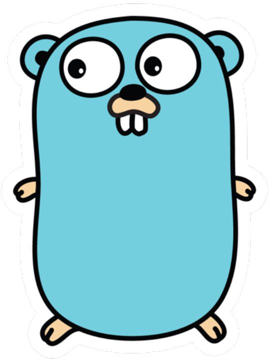
```
// defining a struct

type Foo struct {
   str string
   num int
}


// instantiating a struct

foo := &Foo{
   "abc",
   123,
}


foo = new(Foo)
foo.num = 123
foo.str = "abc"
```



Structs are Objects (kind of)

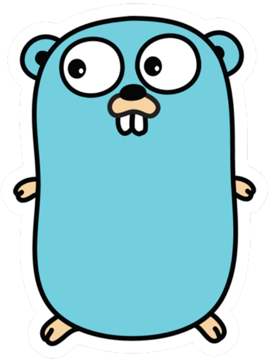
```
// defining a struct

type Foo struct {
   str string
   num int
}


// instantiating a struct

foo := &Foo{
   "abc",
   123,
}


foo = new(Foo)
foo.num = 123
foo.str = "abc"
```

```
func NewFoo(str string, num int) *Foo {
   return &Foo{
      str,
      num,
   }
}
```



Functions on Structs, Pointer Receivers

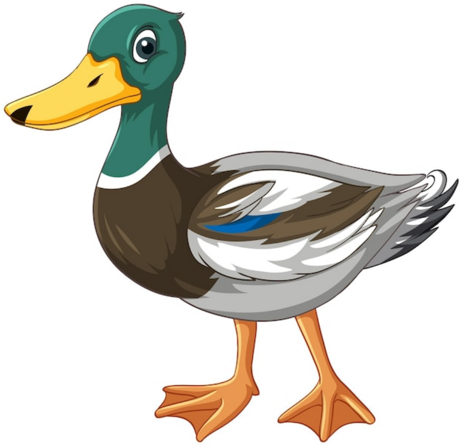
```
// defining functions on structs

func (f *Foo) SetNum(num int) {
   f.num = num
}

func (f *Foo) GetNum() int {
   return f.num
}


// call member function on a struct

foo := NewFoo("abc", 123)

foo.SetNum(42)
```



Interfaces and Duck Typing

if it walks like a duck and quacks like a duck it’s probably a duck…

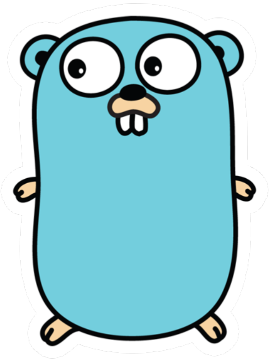
```
type Speaker interface {
    Speak() string
}

type Chicken struct {
}

func (c *Chicken) Speak() string {
    return "i am a chicken"
}

type Duck struct {
}

func (d *Duck) Speak() string {
    return "i am a duck"
}

func main() {
    animals := []Speaker{new(Chicken), new(Duck)}
    for _, a := range animals {
        fmt.Printf("%s\n", a.Speak())
    }
}
```



How to OOP in Go?

composition instead of inheritance

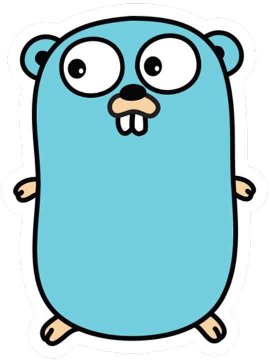
```
type Shape interface {
    Draw() error
}

type Color struct {
    R, G, B int
}

type Point struct {
    X, Y int
}

type Circle struct {
    Color
    Point
    Radius int
}

func (c *Circle) Draw() error {
    fmt.Println("circle with radius " + strconv.Itoa(c.Radius) + " at point " + strconv.Itoa(c.X) + ", " + strconv.Itoa(c.Y))
    return nil
}

func main() {
    var circle Shape
    circle = &Circle{Color{10, 10, 10}, Point{200, 300}, 20}
    circle.Draw()
}
```



How to OOP in Go?

limitation: no overriding

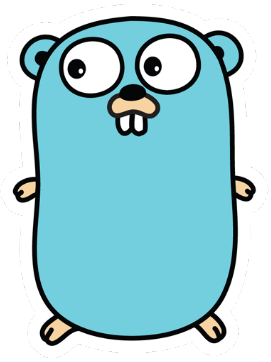
```
type Metric struct {
}

func (m *Metric) Hash() string {
    return ""
}

func (m *Metric) Do() {
    fmt.Println(m.Hash())
}

type Plugin struct {
    Metric
}

func (p *Plugin) Hash() string {
    return "123"
}

func main() {
    p := new(Plugin)
    p.Do()
}
```



How to OOP in Go?

limitation: no overriding – a workaround

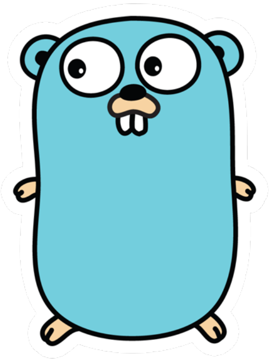
```
type HashFct func() string

type Metric struct {
   Hash HashFct
}

func (m *Metric) Do() {
   fmt.Println(m.Hash())
}

type Plugin struct {
   Metric
}

func (p *Plugin) Hash() string {
   return "123"
}

func main() {
   p := new(Plugin)
   p.Metric.Hash = p.Hash
   p.Do()
}
```



The empty or “anything” `interface{}`

The empty interface `interface{}` can be anything: Any struct, any map or slice, any simple data type!

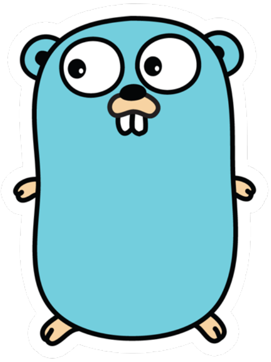
```
 var item interface{}

item = 3

item = map[string]int{"a": 1}

item = new(Foo)
```



The empty or “anything” `interface{}`

type casting and type conversion

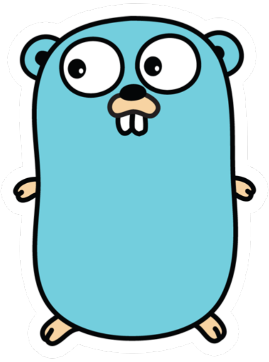
```
func workOnAnything(item interface{}) {

  // type check with switch statement

  switch item.(type) {
  case map[string]string:

    // then safely perform a type cast

    m := item.(map[string]string)
    foo := m["Foo"]
    fmt.Printf("item is a string map and foo is %s\n", foo)

  case []string:
    fmt.Println("item is a string slice")
  default:
    fmt.Println("item is something else")
  }

}
```



The empty or “anything” `interface{}`

type casting and type conversion

```
func workOnAnything(item interface{}) {

  // type check by attempting type cast with comma ok idiom

  if str, ok := item.(string); ok {
    fmt.Println("item is a string”, str)
  } else {
    fmt.Println("item is not a string")
  }

  // some functions happily operate on interfaces

  buf, err := json.MarshalIndent(item, "", "\t")

  if err != nil {
    fmt.Println(err)
  }

  // type conversion with type()

  str := string(buf)

  fmt.Println(str)

}
```

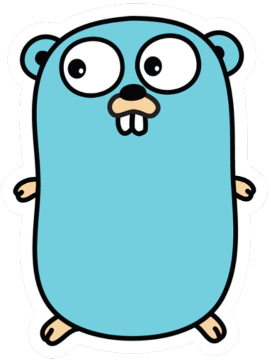


Goroutines And Channels

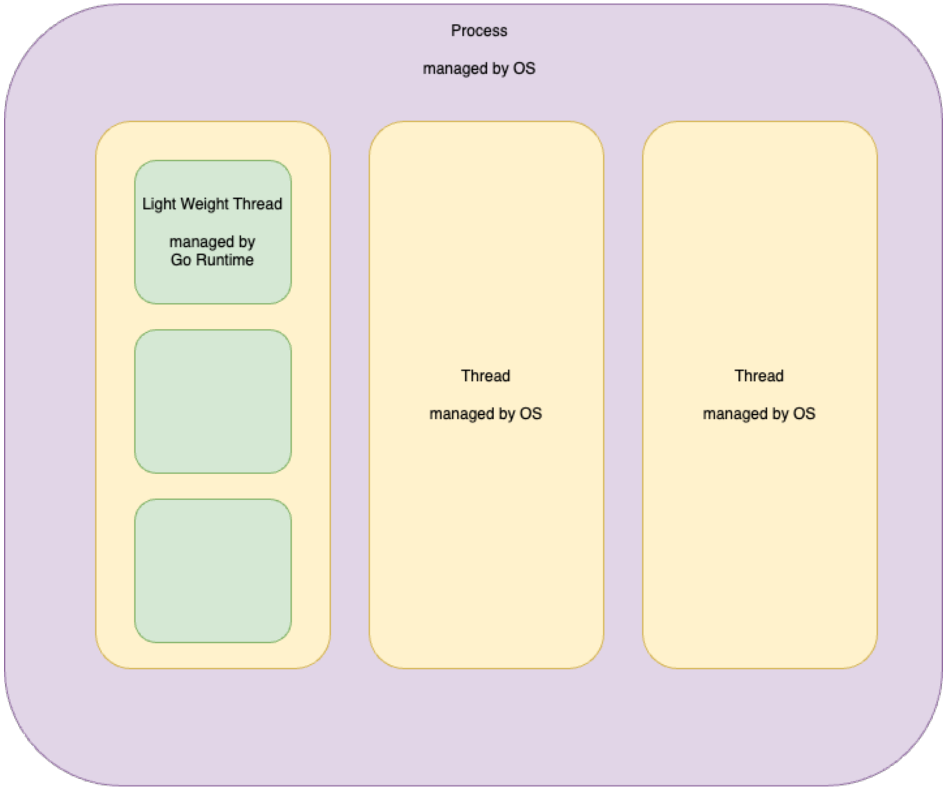
Goroutines

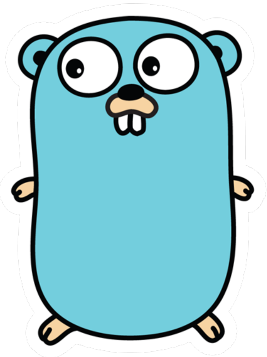
- Goroutines are light-weight threads managed by the Go runtime.
- You can afford to use many Goroutines in your programs (10k no problem)
- Simply prefix a function call with the go keyword to make it run on its own (light-weight) thread.
- *Concurrent execution is not the same as parallel execution!*
  - Rob Pike
    - https://www.youtube.com/watch?v=oV9rvDlIKEg&t=3s
    - https://go.dev/talks/2012/waza.slide#1
- Allows you you to write scalable code (parallelizable code)
- Use GOMAXPROCS environment variable to manage number of OS threads the Go runtime can use – value defaults to number of cores.

```
go myFunc()
```



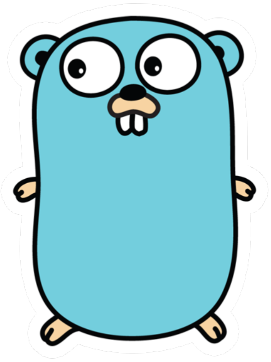
A Goroutines is a Light Weight Thread





Goroutines

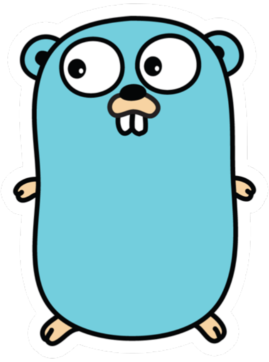
```
func Expensive(num int) {
    fmt.Printf("%d\n", num)
    time.Sleep(1 * time.Second)
}

func SequentialExecution() {
    for i := 0; i < 10; i++ {
       Expensive(i)
    }
}

func ConcurrentExecution() {
    for i := 0; i < 10; i++ {
       go Expensive(i)
    }
}
```



Goroutines

careful when using closures

```
func ConcurrentExecutionAnonymousFlawed() {
   for i := 0; i < 10; i++ {
      // closure over variable i cause problems due to concurrent execution
      go func() {
         fmt.Printf("%d\n", i)
         time.Sleep(1 * time.Second)
      }()
   }
}

func ConcurrentExecutionAnonymous() {
   for i := 0; i < 10; i++ {
      // better to pass data into function as parameter
      go func(num int) {
         fmt.Printf("%d\n", num)
         time.Sleep(1 * time.Second)
      }(i)
   }
}
```



Goroutines

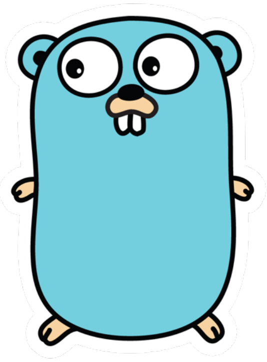
use wait group for synchronization

```
func ConcurrentExecutionAnonymousWithWaitGroup() {
    // if we don't wait, the function will return before all the work is done
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func(num int) {
            fmt.Printf("%d\n", num)
            time.Sleep(1 * time.Second)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

What are Channels?

- Channels are typed pipes through which goroutines can send and receive data using the channel operator <-
- Channels are used to synchronize data passing among go routines (no locks needed!)

```
c := make(chan int)
c <- 42
val := <- c
```

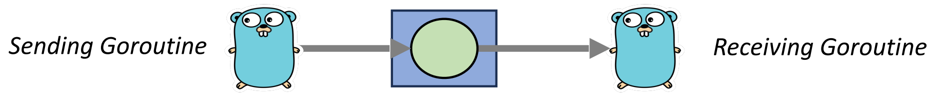


What are Channels?

There are buffered and unbuffered channels

- Use unbuffered channels for synchronous communication and signaling, sending blocks if receiver not ready, receiving blocks if sender not ready
- Use buffered channels for asynchronous communication (queue), sending only blocks when buffer is full, reading only blocks when buffer is empty

Unbuffered Channel

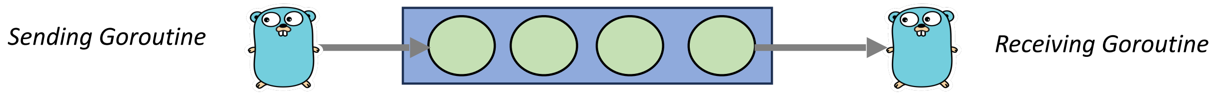


Sending Goroutine

Receiving Goroutine

```
c := make(chan int)
```

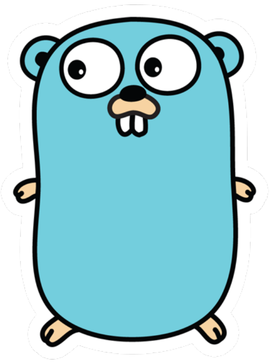
Buffered Channel



Sending Goroutine

Receiving Goroutine

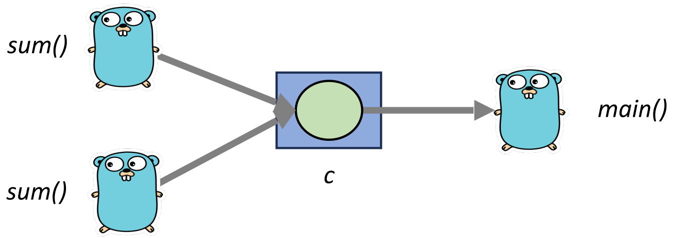
```
c := make(chan int, 4)
```

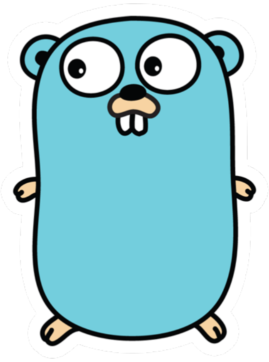


Channels

divide and conquer

```
func sum(a []int, c chan int) {
   sum := 0
   for _, v := range a {
      sum += v
   }
   c <- sum
}

func main() {
   a := []int{7, 2, 8, -9, 4, 0}
   c := make(chan int)
   go sum(a[:len(a)/2], c)
   go sum(a[len(a)/2:], c)
   x := <-c
   y := <-c
   fmt.Println(x, y, x+y)
}
```

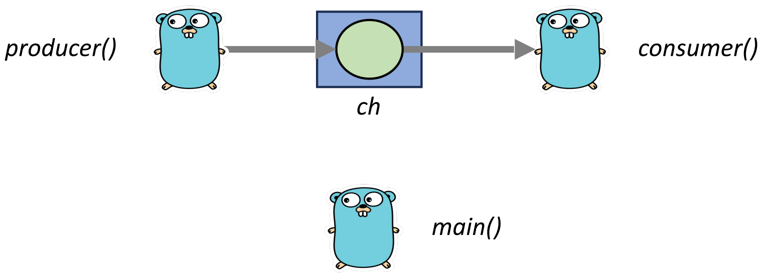


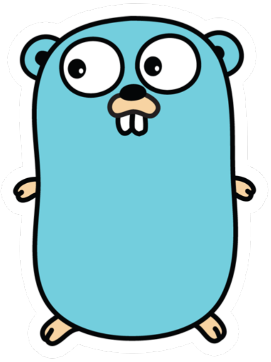


Producer Consumer Pattern

synchronization with wait group

```
func producer(ch chan int, wg *sync.WaitGroup) {
    defer close(ch)
    defer wg.Done()
    for i := 0; i < 5; i++ {
        fmt.Printf("produced %d\n", i)
        ch <- i
    }
}

func consumer(ch chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for num := range ch {
        fmt.Printf("consumed %d\n", num)
    }
}

func main() {
    var wg sync.WaitGroup
    ch := make(chan int)
    wg.Add(2)
    go producer(ch, &wg)
    go consumer(ch, &wg)
    wg.Wait()
```

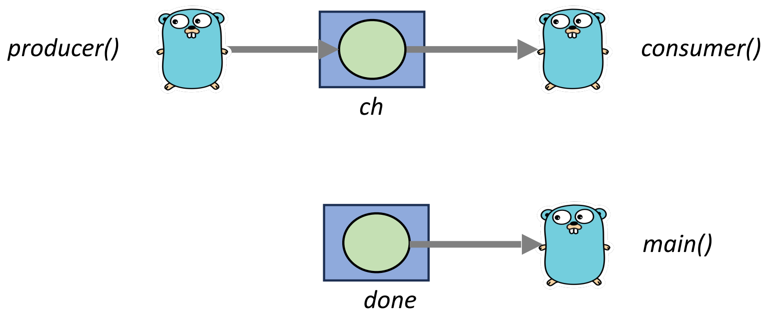


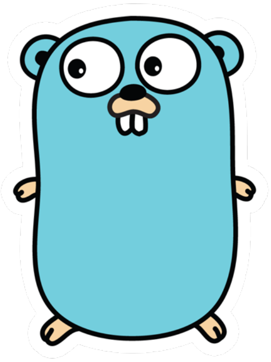


Producer Consumer Pattern

synchronization with extra channel

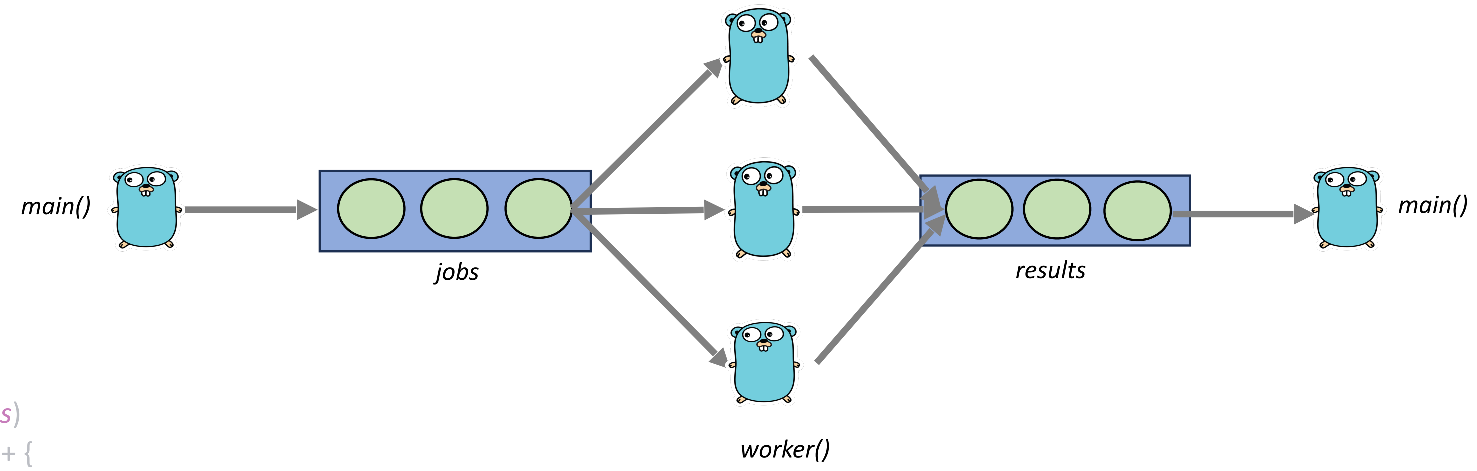
```
func producer(ch chan int) {
    defer close(ch)
    for i := 0; i < 5; i++ {
      fmt.Printf("produced %d\n", i)
      ch <- i
    }
}

func consumer(ch chan int, done chan struct{}) {
    for num := range ch {
      fmt.Printf("consumed %d\n", num)
    }
    close(done)
}

func main() {
    ch := make(chan int)
    done := make(chan struct{})
    go producer(ch)
    go consumer(ch, done)
    <-done
}
```

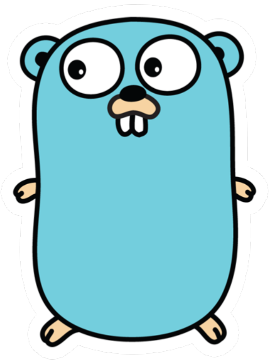




Fan-Out and Fan-In / Load Balancer

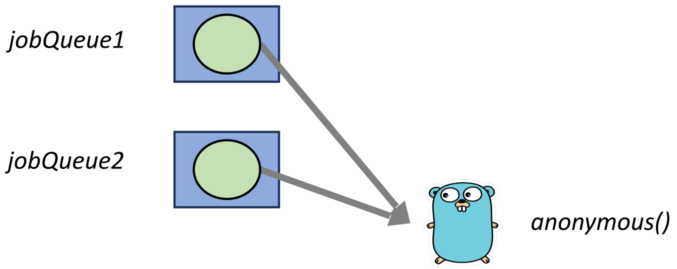
```
func worker(id int, jobs <-chan int, results chan<- int) {
  for j := range jobs {
    time.Sleep(time.Second)
    results <- j * 2
  }
}

func main() {
  const numJobs = 10
  const numWorkers = 3
  jobs := make(chan int, numJobs)
  results := make(chan int, numJobs)
  for w := 1; w <= numWorkers; w++ {
    go worker(w, jobs, results)
  }
  for j := 1; j <= numJobs; j++ {
    jobs <- j
  }
  close(jobs)
  sum := 0
  for a := 1; a <= numJobs; a++ {
    sum += <-results
  }
  fmt.Println("sum of all jobs is ", sum)
```

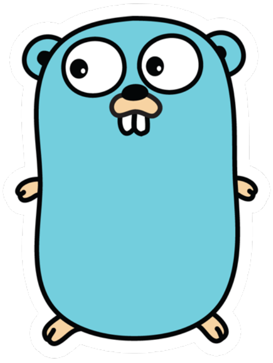




Selecting from Multiple Channels

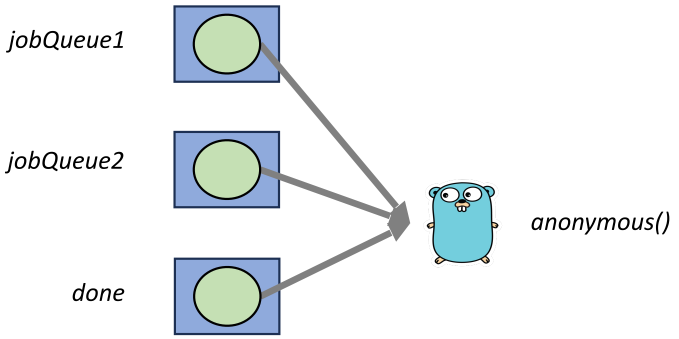
```
jobQueue1 := make(chan string)
jobQueue2 := make(chan string)

go func() {
   for {
      select {
      case j := <-jobQueue1:
         doJob(j)
      case j := <-jobQueue2:
         doJob(j)
      }
   }
}()
```

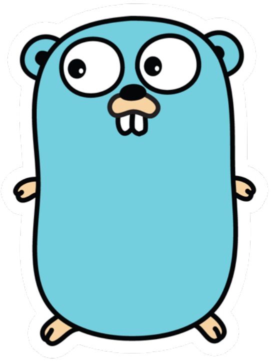




Selecting from Multiple Channels

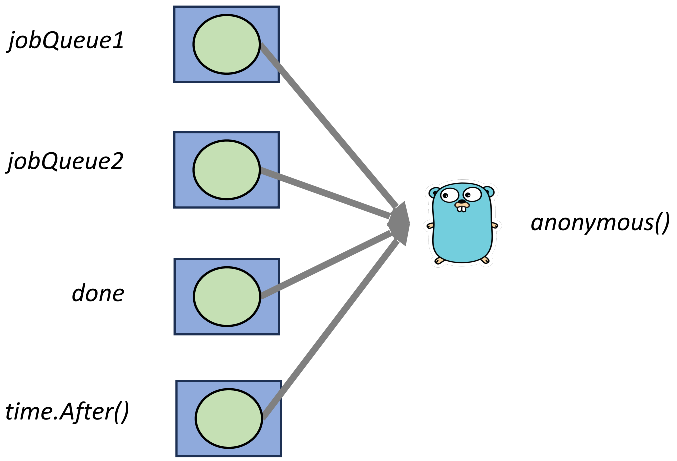
```
jobQueue1 := make(chan string)
jobQueue2 := make(chan string)
done := make(chan struct{})

go func() {
   for {
     select {
     case j := <-jobQueue1:
        doJob(j)
     case j := <-jobQueue2:
        doJob(j)
     case <-done:
        fmt.Println("shutting down")
        return
     }
   }
}()
```

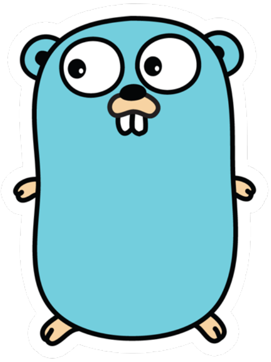




Selecting from Multiple Channels

```
jobQueue1 := make(chan string)
jobQueue2 := make(chan string)
done := make(chan struct{})

go func() {
   for {
     select {
     case j := <-jobQueue1:
       doJob(j)
     case j := <-jobQueue2:
       doJob(j)
     case <-done:
       fmt.Println("shutting down")
       return
     case <-time.After(60 * time.Second):
       fmt.Println("alert: no new job within 60 seconds")
     }
   }
}()
```

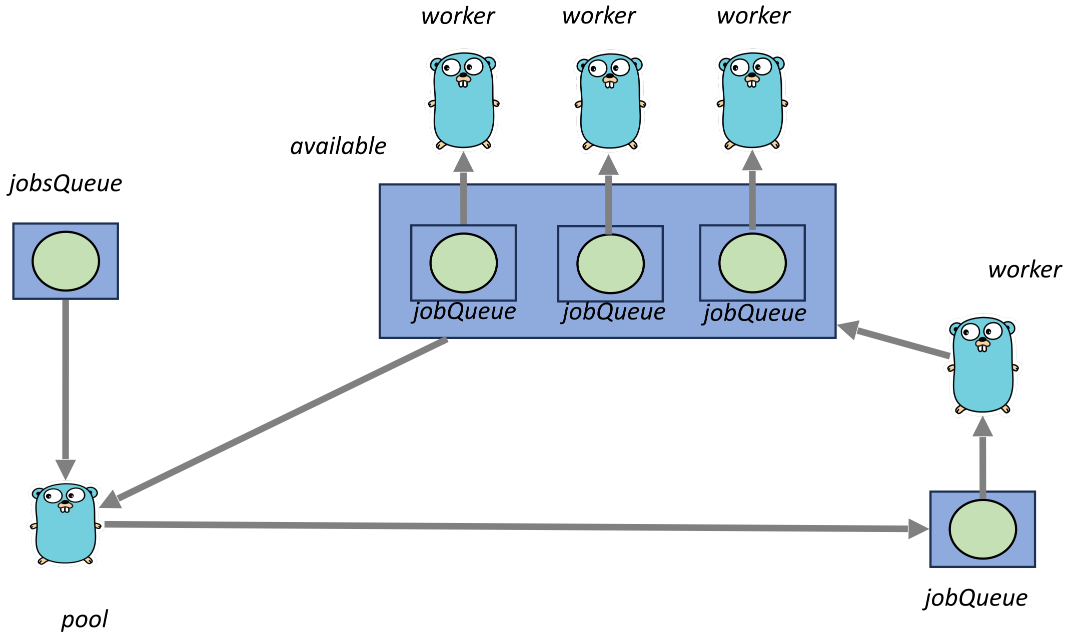


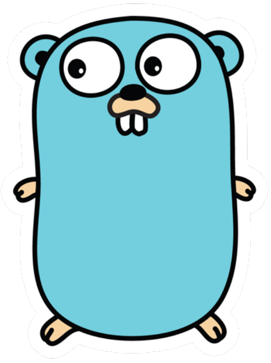


Worker Pool

```go
type DoJob func(*Job) string

type Worker struct {
   jobQueue chan *Job
   done     chan struct{}
   id       string
   do       DoJob
   pool     *Pool
}

func NewWorker(id string, do DoJob, pool *Pool) *Worker {
   return &Worker{
      jobQueue: make(chan *Job),
      done:     make(chan struct{}),
      id:       id,
      do:       do,
      pool:     pool,
   }
}

func (w *Worker) Run() *Worker {
   go func() {
      for {
         select {
         case j := <-w.jobQueue:
            w.do(j)
            w.pool.available <- w
         case <-w.done:
            fmt.Println("shutdown", w.id)
            return
         }
      }
   }()
   return w
}
```

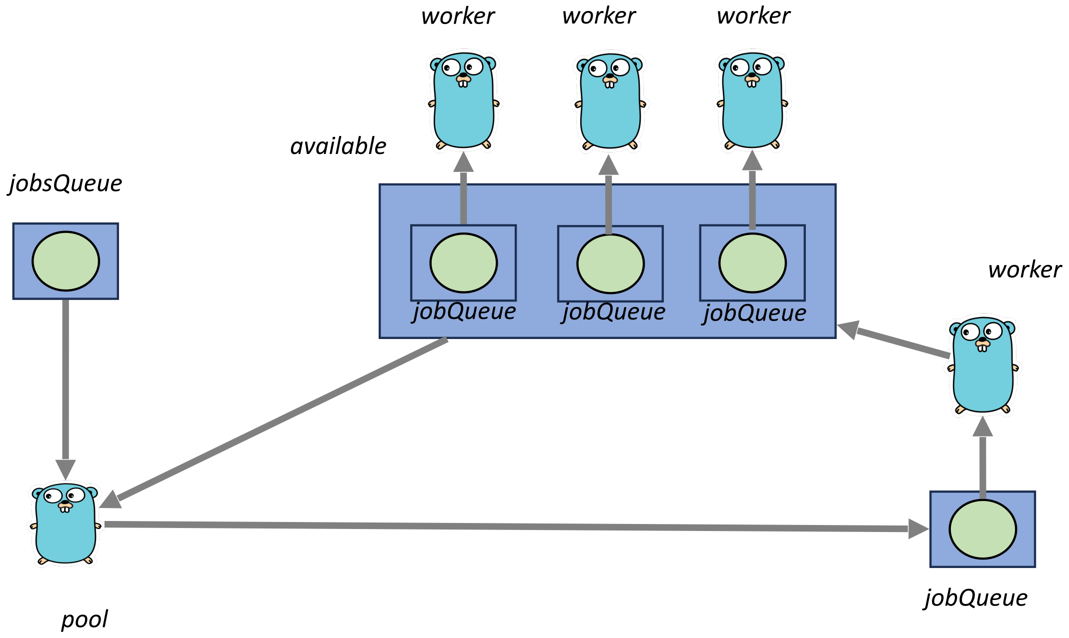
main()

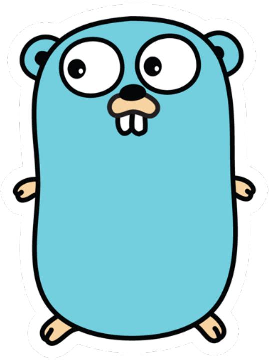




Worker Pool

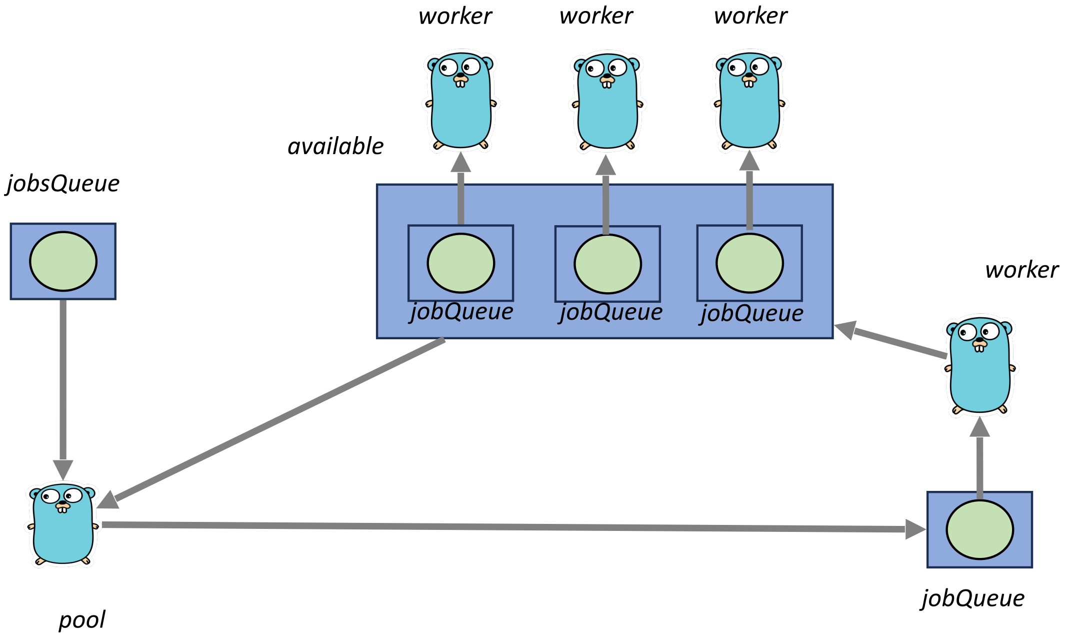
```
type Pool struct {
   available chan *Worker
   jobsQueue chan *Job
}

func NewPool(size int, Do DoJob, JobQueue chan *Job) *Pool {
   p := &Pool{
     available: make(chan *Worker, size),
     jobsQueue: JobQueue,
   }
   for i := 0; i < size; i++ {
     p.available <- NewWorker(strconv.Itoa(i), Do, p).Run()
   }
   return p
}

func (p *Pool) Launch() {
   go func() {
     for j := range p.jobsQueue {
       w := <-p.available
       w.jobQueue <- j
     }
   }()
}

func (p *Pool) Shutdown() {
   go func() {
     for w := range p.available {
       w.done <- struct{}{}
     }
   }()
```

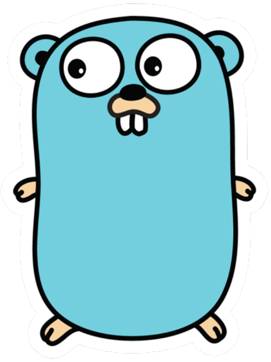




Worker Pool

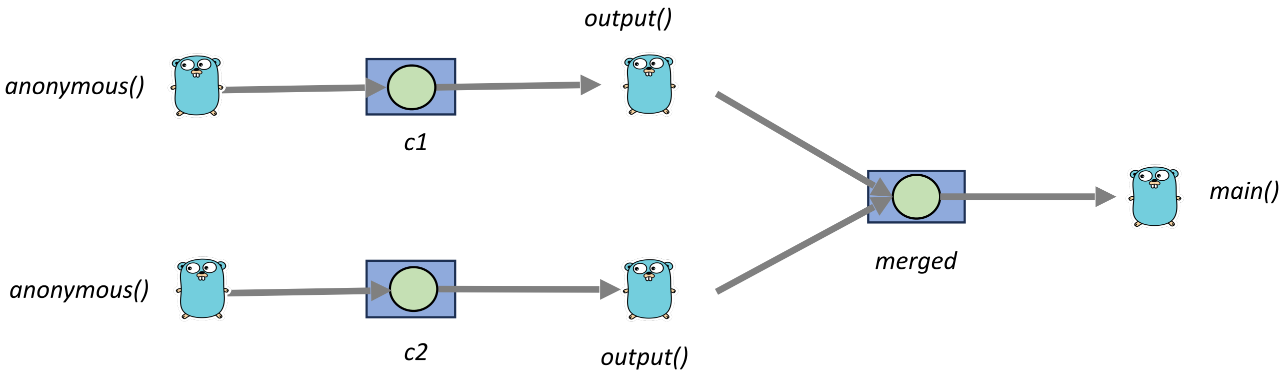
```go
func main() {
    const NUM_JOBS = 25
    const NUM_WORKERS = 10
    jobs := make(chan *Job)
    do := func(j *Job) string {
        pauseSec := 1 //rand.Intn(5) + 1
        time.Sleep(time.Duration(pauseSec) * time.Second)
        fmt.Println("task", j.Task, "duration", pauseSec)
        return ""
    }
    pool := NewPool(NUM_WORKERS, do, jobs)
    pool.Launch()
    fmt.Println("sending work")
    for i := 0; i < NUM_JOBS; i++ {
        jobs <- NewJob(strconv.Itoa(i))
    }
    fmt.Println("shutting down")
    pool.Shutdown()
    time.Sleep(5 * time.Second)
}
```





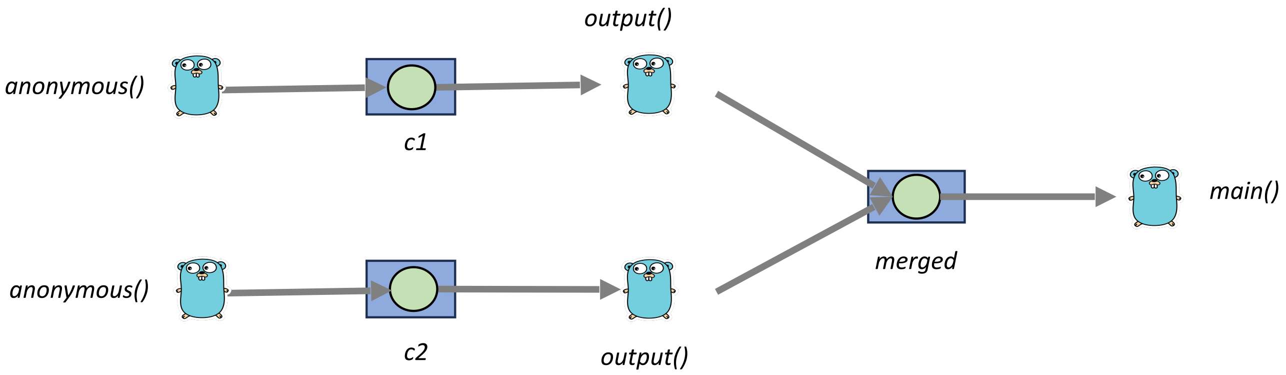
Merge Multiple Channels Into One

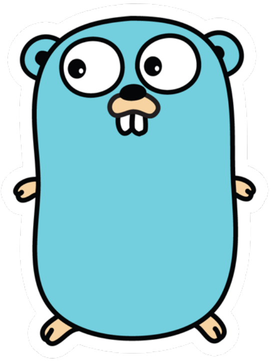
```
func main() {

  c1 := make(chan int)
  c2 := make(chan int)

  merged := merge(c1, c2)

  go func() {
    for _, x := range []int{1, 2, 3} {
      c1 <- x
    }
    close(c1)
  }()

  go func() {
    for _, x := range []int{4, 5, 6} {
      c2 <- x
    }
    close(c2)
  }()

  for n := range merged {
    fmt.Println(n)
  }
```



Merge Multiple Channels Into One

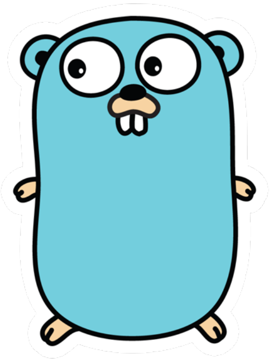
```go
func merge(channels ...chan int) chan int {

  var wg sync.WaitGroup
  merged := make(chan int)

  output := func(c chan int) {
    for n := range c {
      merged <- n
    }
    wg.Done()
  }

  wg.Add(len(channels))

  for _, c := range channels {
    go output(c)
  }

  go func() {
    wg.Wait()
    close(merged)
  }()

  return merged
}
```





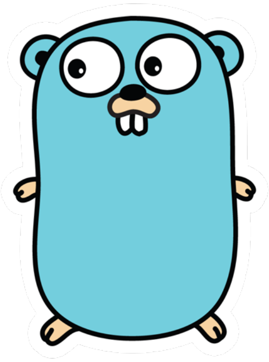
Closing Multiple Workers with One Done Channel

```
func worker(done chan struct{}) {
    for {
        select {
        case <-done:
            fmt.Println("shutting down")
            return
        default:
            time.Sleep(1 * time.Second)
            fmt.Println("working")
        }
    }
}

func main() {
    done := make(chan struct{})
    go worker(done)
    go worker(done)
    time.Sleep(3 * time.Second)
    close(done)
    //done <- struct{}{}
    //done <- struct{}{}
    time.Sleep(3 * time.Second)
}
```



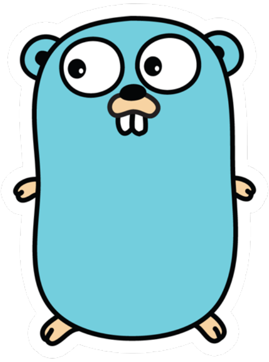
Go AWS SDK

```
func receive (ch chan Message) error {
  logger := log.New(os.Stdout, "INFO: ", log.Ldate|log.Ltime)
  sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region),
  })
  if err != nil {
    return err
  }
  svc := sqs.New(sess)
  go func() {
    for {
      result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
        QueueUrl:           aws.String(queueURL),
        MaxNumberOfMessages: aws.Int64(1),
        VisibilityTimeout:   aws.Int64(30),
        WaitTimeSeconds:     aws.Int64(20),
      })
      if err != nil {
        logger.Println(err)
      }
      for _, message := range result.Messages {
        var msg Message
        err = json.Unmarshal([]byte(*message.Body), &msg)
        if err != nil {
          logger.Println(err)
        }
        ch <- msg
      }
    }
  }()
  return nil
}
```



Go AWS SDK

```
func receive(ch chan Message, done chan struct{}) error {
  logger := log.New(os.Stdout, "INFO: ", log.Ldate|log.Ltime)
  sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region),
  })
  if err != nil {
    return err
  }
  svc := sqs.New(sess)
  go func() {
    for {
      select {
      case <-done:
        return
      default:
        result, err := svc.ReceiveMessage(&sqs.ReceiveMessageInput{
          QueueUrl:            aws.String(queueURL),
          MaxNumberOfMessages: aws.Int64(1),
          VisibilityTimeout:   aws.Int64(30),
          WaitTimeSeconds:     aws.Int64(20),
        })
        if err != nil {
          logger.Println(err)
        }
        for _, message := range result.Messages {
          var msg Message
          err = json.Unmarshal([]byte(*message.Body), &msg)
          if err != nil {
            logger.Println(err)
          }
          ch <- msg
        }
      }
    }
  }()
  return nil
}
```

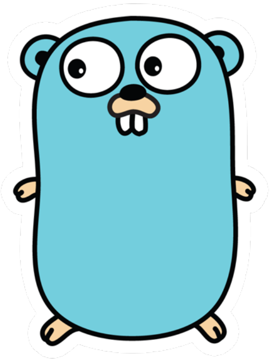


Miscellaneous Techniques & Patterns

Golang Strings

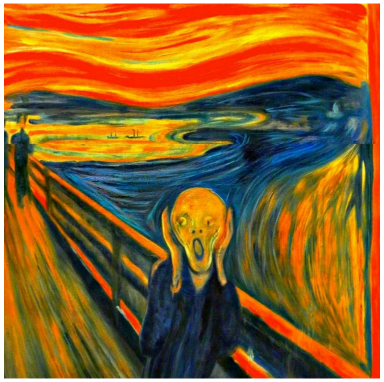
strings are byte slices, strings are UTF-8 encoded

```
const s = "你好世界"

fmt.Println("len:", len(s))

for i := 0; i < len(s); i++ {
    fmt.Printf("%x ", s[i])
}

fmt.Println()

for idx, rune := range s {
    fmt.Printf("%c starts at %d\n", rune, idx)
}
```



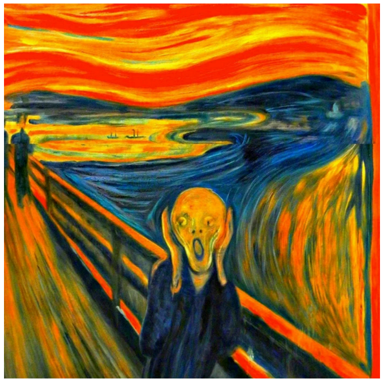
Catching Panics

```
func doNilPointer() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("recovered from panic:", r)
        }
    }()
    var m map[string]string
    m["foo"] = "bar"
}
```



Catching Panics

```go
func doExplicitPanic() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("recovered from panic:", r)
        }
    }()
    panic("explicit panic")
}
```



Errors: errors.As() errors.Is()

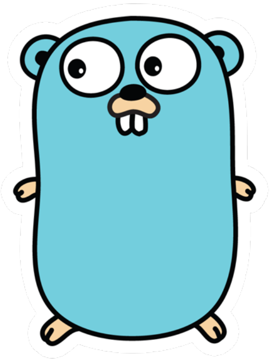
```go
type CustomError struct {
    Code int
}

func (e CustomError) Error() string {
    return fmt.Sprintf("Custom Error with code: %d", e.Code)
}

func processFile(filename string) error {
    _, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed to open file: %w", err)
    }
    return CustomError{Code: 42}
}

func main() {

    filename := "file.txt"
    err := processFile(filename)
    var customErr CustomError

    if errors.As(err, &customErr) {
        fmt.Printf("Custom Error: Code %d\n", customErr.Code)
    } else {
        fmt.Println("Not a custom error")
    }

    if errors.Is(err, os.ErrNotExist) {
        fmt.Println("File not found")
    } else {
        fmt.Println("Not a file not found error")
    }
}
```

Errors: `errors.As()` `errors.Is()`

```
type CustomFileError struct {
   msg string
   err error
}

func (e *CustomFileError) Error() string {
   return fmt.Sprintf("%s: %v", e.msg, e.err)
}

func (e *CustomFileError) Unwrap() error {
   return e.err
}

func WrapErrNotExist(msg string) error {
   return &CustomFileError{msg, os.ErrNotExist}
}
```

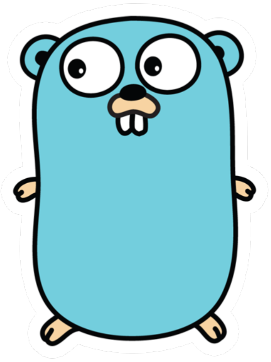
Functional Options

```
type Server struct {
    Host    string
    Port    int
    Timeout int
}

type ServerOption func(*Server)

func WithHost(host string) ServerOption {
    return func(s *Server) {
      s.Host = host
    }
}

func WithPort(port int) ServerOption {
    return func(s *Server) {
      s.Port = port
    }
}

func WithTimeout(timeout int) ServerOption {
    return func(s *Server) {
      s.Timeout = timeout
    }
```



Functional Options

```go
func NewServer(options ...ServerOption) *Server {
    server := &Server{
        Host:    "localhost",
        Port:    8080,
        Timeout: 30,
    }
    for _, option := range options {
        option(server)
    }
    return server
}

func main() {
    server := NewServer(
        WithHost("example.com"),
        WithPort(9090),
        WithTimeout(60),
    )
}
```



Generics

since version 1.18

```
func SumInts(a []int64) int64 {
    var s int64
    for _, v := range a {
       s += v
    }
    return s
}

func SumFloats(a []float64) float64 {
    var s float64
    for _, v := range a {
       s += v
    }
    return s
}
```

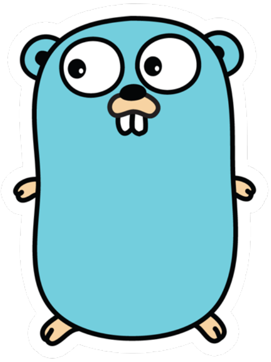
Generics

since version 1.18

```
func Sum[V int64 | float64](a []V) V {
    var s V
    for _, v := range a {
        s += v
    }
    return s
}
```

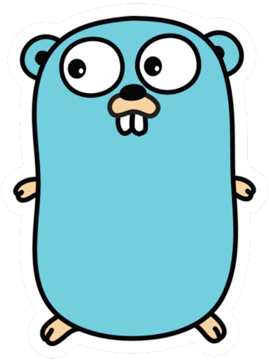
Threadsafe Maps

```
type StringMap interface {
   Get(string) string
   Set(string, string)
}

type SafeStringMap struct {
   sync.Mutex
   m map[string]string
}

func NewSafeStringMap(size int) StringMap {
   return &SafeStringMap{
      m: make(map[string]string, size),
   }
}

func (s *SafeStringMap) Get(key string) string {
   s.Lock()
   defer s.Unlock()
   return s.m[key]
}

func (s *SafeStringMap) Set(key string, value string) {
   s.Lock()
   defer s.Unlock()
   s.m[key] = value
}
```



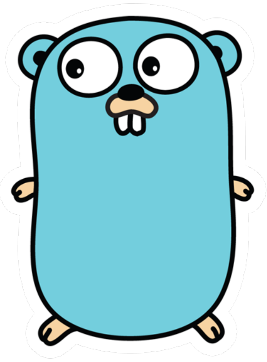
Threadsafe Maps

```
func main() {
    m := NewSafeStringMap(100)
    var wg sync.WaitGroup
    wg.Add(200)
    for i := 0; i < 100; i++ {
        go func() {
            m.Set("foo", "bar")
            wg.Done()
        }()
    }
    for i := 0; i < 100; i++ {
        go func() {
            m.Get("foo")
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println("done")
}
```



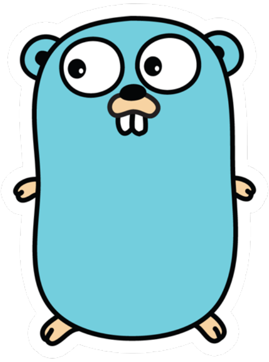
JSON In JSON Out

```
str :=
  `{
    "Foo" : "Bar"
  }`

// parse json string

var m map[string]string

err := json.Unmarshal([]byte(str), &m)

if err != nil {
  fmt.Println(err)
}

// now you can reach inside the object

if m["Foo"] != "Bar" {
  fmt.Println("unexpected data")
}

// serialize json to get the original string back

buf, err := json.MarshalIndent(m, "", "\t")

if err != nil {
  fmt.Println(err)
} else {
  fmt.Printf("%s\n", string(buf))
}
```



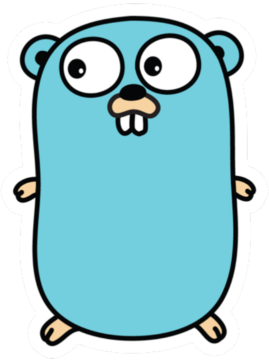
JSON with Serialization Hints

```
type Message struct {
   Foo string `json:"foo,omitempty"`
}
str :=
   `{
     "foo" : "Bar"
    }`

var msg Message

err = json.Unmarshal([]byte(str), &msg)

if err != nil {
   fmt.Println(err)
} else {
   fmt.Println(msg.Foo)
}
```



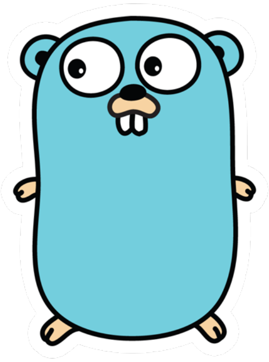
Go Context

```go
func longRunningTask(ctx context.Context) {
   val := ctx.Value("foo")
   select {
   case <-time.After(5 * time.Second):
     fmt.Println("task completed successfully ", val)
   case <-ctx.Done():
     fmt.Println("task was canceled", val)
   }
}

func main() {
   ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
   ctx = context.WithValue(ctx, "foo", "bar")
   defer cancel()
   fmt.Println("starting task")
   go longRunningTask(ctx)
   time.Sleep(2 * time.Second)
   cancel()
   time.Sleep(1 * time.Second)
}
```



Shell Commands

```
cmd := exec.Command("ls", "-l")

var stdout, stderr bytes.Buffer
cmd.Stdout = &stdout
cmd.Stderr = &stderr

err := cmd.Run()
if err != nil {
    fmt.Printf("Error: %s\n", err)
}

fmt.Printf("%s", stdout.String())
fmt.Printf("%s", stderr.String())
```



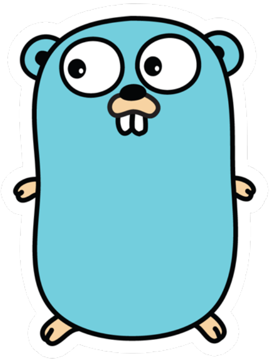
Capturing CTRL-C and other OS-Signals

```
done := make(chan bool, 0)
work := make(chan string, 0)

go func() {
  for {
    fmt.Println("waiting for work")
    select {
    case w := <-work:
      // do work
      fmt.Println("doing work: ", w)
    case <-done:
      // cleanup and exit go routine in a clean way
      fmt.Println("cleaning up")
      return
    }
  }
}()

// create a channel to receive os.Signal values

sigs := make(chan os.Signal, 1)

// notify sigs when a SIGINT (Ctrl-C) is received

signal.Notify(sigs, syscall.SIGINT)

// wait for signal to come in

<-sigs

// now cleanup

done <- true
```

Web Service Framework

https://github.comcast.com/bwolf200/goworkshop/blob/main/webservice/main.go

https://github.com/rs/zerolog - structured logging
https://github.com/gorilla/mux - http web services
https://github.com/spf13/cobra - cli commands
https://github.com/spf13/viper - configuration

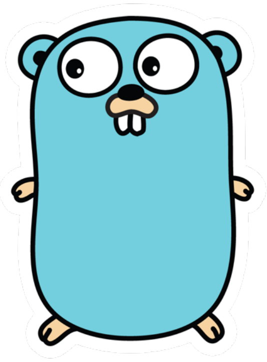


Some Fun Talks

- Rob Pike, Lexical Scanning in Go (channels)
  - https://www.youtube.com/watch?v=HxaD_trXwRE
- Liz Rice, Building a docker like container system from scratch in Go (Unix system commands)
  - https://www.youtube.com/watch?v=Utf-A4rODH8

Some Resources

- Go Playground https://go.dev/play/
- A Tour Of Go https://go.dev/tour/list
- Go By Example https://gobyexample.com/
- How to write Go code https://go.dev/doc/code
- Effective Go https://go.dev/doc/effective_go
- Go Blog https://go.dev/blog/strings
- Downloads https://go.dev/dl/
- Go Debugger https://github.com/go-delve/delve



THANK YOU

BACKUP

Go Tool Chain & Testing

Compiling Go Code

cross compilation, packages, modules

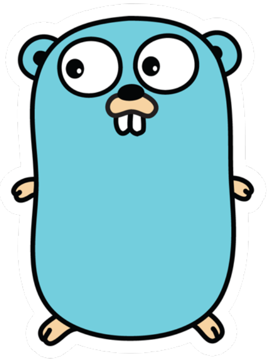
```
go build
```

```
GOOS=linux GOARCH=386 go build -o myprogram myprogram.go
```

```
go mod init
```

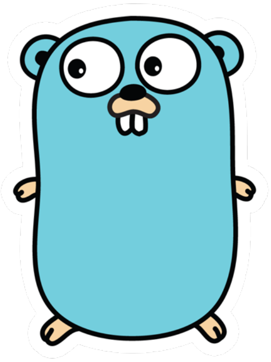
```
go mod tidy
```

```
go get somedomain.com/somepackage
```



Unit Tests

```
func FactorialA(n int) int {
   if n <= 1 {
      return 1
   }
   return n * FactorialA(n-1)
}

func FactorialB(n int) int {
   result := 1
   for i := 2; i <= n; i++ {
      result *= i
   }
   return result
}

func FactorialC(cache map[int]int, n int) int {
   if result, ok := cache[n]; ok {
      return result
   }
   result := FactorialB(n)
   cache[n] = result
   return result
}
```



Unit Tests

convention: test function must start with Test and file name must end with _test.go

```
func TestFactorialA(t *testing.T) {
    got := FactorialA(10)
    want := 3628800
    if got != want {
        t.Errorf("Factorial(10 = %d; want %d", got, want)
    }
}

func TestFactorialB(t *testing.T) {
    got := FactorialB(10)
    want := 3628800
    if got != want {
        t.Errorf("Factorial(10 = %d; want %d", got, want)
    }
}
```

```
go test

go test -race
```

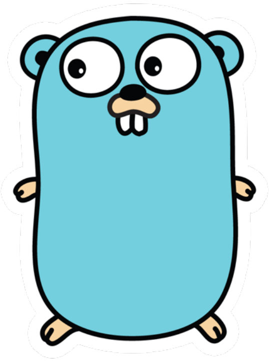


Table Driven Testing

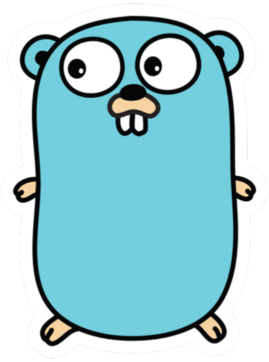
```go
func add(a, b int) int {
   return a + b
}

func TestAdd(t *testing.T) {
   tests := []struct {
      name     string
      a, b     int
      expected int
   }{
      {"add 1 + 2", 1, 2, 3},
      {"add 10 + 20", 10, 20, 30},
      {"add 0 + 0", 0, 0, 0},
   }

   for _, tt := range tests {
      t.Run(tt.name, func(t *testing.T) {
         result := add(tt.a, tt.b)
         if result != tt.expected {
            t.Errorf("expected %d, got %d", tt.expected, result)
         }
      })
   }
}
```

Benchmarks

careful: execution time must converge, otherwise benchmarking never ends

```
func BenchmarkFactorialA(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
       r = FactorialA(100)
    }
    result = r
}

func BenchmarkFactorialB(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
       r = FactorialB(100)
    }
    result = r
}
```

```
go test -bench=.
```



Golden Files with goldie

```
package main

import (
    "encoding/json"
    "github.com/sebdah/goldie/v2"
    "testing"
)

func TestMyFunction(t *testing.T) {
    g := goldie.New(t)
    myOutput := MyFunction()
    buf, _ := json.MarshalIndent(myOutput, "", "\t")
    g.Assert(t, "my_function_output", buf)
}
```

```
go test -update
```

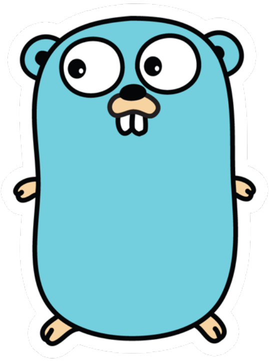
Profiling with pprof

```
import (
_ "net/http/pprof"
)
```

```
go tool pprof http://localhost:8080/debug/pprof/heap
```

https://github.comcast.com/bwolf200/goworkshop/blob/main/pprof/main.go

https://jvns.ca/blog/2017/09/24/profiling-go-with-pprof/



Linting

```
go install github.com/golangci/golangci-lint/cmd/golangci-lint@latest
golangci-lint run
```

https://golangci-lint.run/usage/configuration/#config-file

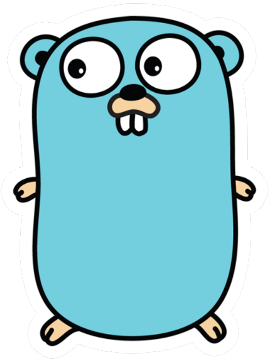
```
m := map[string]string{"Hello": "Golang"}

if m != nil && len(m) > 0 {
   m["Hello"] = "World"
}

buf, _ := json.Marshal(m)

fmt.Printf("%d\n", string(buf))
```

Random Stuff

Pointers to Pointers

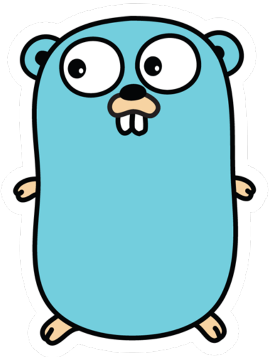
passing a pointer by reference

```
type Node struct {
  Value int
  Next  *Node
}

func InsertNode(node **Node, value int) {
  newNode := &Node{Value: value}
  if *node != nil {
    newNode.Next = (*node).Next
    (*node).Next = newNode
  } else {
    *node = newNode
  }
}
```



Struct Composition and Shadowing

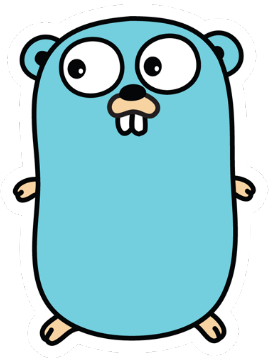
```go
type Inner struct {
    Name string
}

type Outer struct {
    Inner
    Name string
}

func main() {

    o := Outer{
      Inner: Inner{Name: "Inner Name"},
      Name:  "Outer Name",
    }

    fmt.Println(o.Name)
    fmt.Println(o.Inner.Name)
}
```



Using C Libraries With CGO

it works, but it’s pretty ugly…

```
package main

/*
#include <stdio.h>
#include <stdlib.h>
*/
import "C"
import "unsafe"

func main() {
    cs := C.CString("i can c you\n")
    C.puts(cs)
    C.free(unsafe.Pointer(cs))
}
```



go:embed

```go
//go:embed web/*
var content embed.FS

func main() {
   http.Handle("/", http.FileServer(getFileSystem()))
   http.ListenAndServe(":8080", nil)
}

func getFileSystem() http.FileSystem {
   fsys, err := fs.Sub(content, "web")
   if err != nil {
      panic(err)
   }
   return http.FS(fsys)
}
```