Intro to Go

Syntax, Concurreny and Comparison of Golang

Lakshya Singh

- Open Source Contributor
- Software Developer
- Polyglot Gopher, Rustacean, Pythonic, Typescripter
- 3rd Year CSE IIT BHU

Hey everyone 👋

Organization of this workshop

- Part1: Introduction to Go
- Part2: Basic Syntax
- Part3: Tools and Dependencies
- Part4: AMA
- Part5: Obect-oriented Programming
- Part6: Concurrency in Go
- Part7: Error Handling

Motivation

Have

- Incomprehensible and unsafe code.
- Extremely slow builds.
- Missing concurrency support.
- Outdated tools.

Want

- Readable, safe, and efficient code.
- A build system that scales.
- Good concurrency support.
- Tools that can operate at Google-scale.

Meet Go

- Compact, yet expressive
- Statically typed and garbage collected
- Object- but not type-oriented
- Strong concurrency support
- Efficient implementation
- Rich standard library
- Fast compilers
- Scalable tools

Hello World

```go
package main

import "fmt"

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

Syntax

Syntax

Ideas on building Language

> Syntax is not important... - unless you are a programmer.

Rob Pike.

> The readability of programs is immeasurably more important than their writeability.

Hints on Programming Language Design

C. A. R. Hoare 1973

Too Verbose

```
scoped_ptr<logStats::LogStats>
    logStats(logStats::LogStats::NewLogStats(FLAGS_logStats, logStats::LogStats::kFIFO));
```

Too Dense

```
(n: Int) ⇒ (2 to n) ▷ (r ⇒ r.foldLeft(r.toSet)((ps, x) ⇒
    if (ps(x)) ps -- (x * x to n by x) else ps))
```

Just right

```
t := time.Now()
switch {
case t.Hour() < 12:
    return "morning"
case t.Hour() < 18:
    return "afternoon"
default:
    return "evening"
}
```

Structure Go code

Packages

A Go program consists of packages.

A package consists of one or more source files (.go files).

Each source file starts with a package clause followed by declarations.

```
package main

import "fmt"

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

By convention, all files belonging to a single package are located in a single directory.

Declarations, "Pascal-style" (left to right)

Pattern: keyword names [type] [initializer]

```
import "fmt"

const digits = "0123456789abcdef"

type Point struct {
    x, y int
    tag  string
}

var s [32]byte

var msgs = []string{"Hello, Go", "Ciao, Mondo"}

func itoa(x, base int) string
```

Constants

In Go, constants are mathematically precise.

There is no need for qualifiers (-42LL, 7UL, etc.)

```
const (
    MaxUInt = 1<<64 - 1
    Pi      = 3.14159265358979323846264338327950288419716939937510582097494459
    Pi2     = Pi * Pi
    Delta   = 2.0
)
```

Only when used, constants are truncated to size:

```
    var x uint64 = MaxUInt
    var pi2 float32 = Pi2
    var delta int = Delta
```

Huge win in readability and ease of use.

Types

Types

Familiar:

- Basic types, arrays, structs, pointers, functions.

But:

- string is a basic type.
- No automatic conversion of basic types in expressions.
- No pointer arithmetic; pointers and arrays are different.
- A function type represents a function; context and all.

New:

- Slices instead of array pointers + separate length: `` `[]int` ``
- Maps because everybody needs them: `` `map[string]int` ``
- Interfaces for polymorphism: `` `interface{}` ``
- Channels for communication between goroutines: `` `chan int` ``

Variables

Familiar:

```
var i int
var p, q *Point
var threshold float64 = 0.75
```

New: Type can be inferred from type of initializing expression.

```
var i = 42       // type of i is int
var z = 1 + 2.3i // type of z is complex128
```

Shortcut (inside functions):

```
i := 42 // type of i is int
```

It is safe to take the address of any variable:

```
return &i
```

Array and Slice

An array declaration needs to have a fixed size

```
var scoresDeclare [5]int64
scoresInitialize := [5]int{1, 2, 3, 11, 101}
otherScoresInitialize := [...]int{1, 2, 3, 11, 101}
```

A slice need not have a fixed size

```
var declareFruits []strings
fruits := []strings{"apple", "banana", "guava"}
emptyFruits := make([]strings, 3)
```

> make "Initializes and allocates space in memory for a slice, map, or channel."

Slice can be extended/appended to have more elements as well

```
fruits = append(fruits, "mango")
```

Maps

A simple hash map

```
var users map[int]string
makeUsers := make(map[int]string)
```

acessing map returns two values first the value and other whether key is present of not

```
val, ok := users[11]
if !ok {
    users[11] = "king-11"
}
```

go has no set container but map can serve as set as well

```
set := map[int]bool
_, ok := set[11]
```

Pointers

A pointer in Go is a variable that holds the memory location of that variable instead of a copy of its value.

Pointer type definitions are indicated with a `*` next to the `type name`

```
var lakshya = "lakshya"
var namePointer *string = &lakshya
```

Pointer variable values are accessed with a `*` next to the `variable name`

```
var name = *namePointer
```

To read through a variable to see the pointer address use a `&` next to the pointer variable name

```
var nameAddress = &namePointer
```

Structs

A method to declare related data together

```
type User struct {
    ID        int
    firstName string
    lastName  string
    email     string
}
```

We can directly declare a struct in memory with local scope

```
user := User{11, "Lakshya", "Singh", "lakshay.singh1108@gmail.com"}
```

we might want to save it in memory and get a pointer back

```
pointerToUser := &User{11, "Lakshya", "Singh", "lakshay.singh1108@gmail.com"}
user := *pointerToUser
```

Functions

Functions may have multiple return values:

```
func atoi(s string) (i int, err error)
```

Functions are first-class citizens (closures):

```
func adder(delta int) func(x int) int {
    f := func(x int) int {
        return x + delta
    }
    return f
}
```

You can store functions as variables and also return then from other functions

```
var inc = adder(1)
fmt.Println(inc(0))
fmt.Println(adder(-1)(10))
```

Variadic Function

```go
func doThings(args ...int) int {
    total := 0
    for _, num := range args {
        total += num
    }
    return total
}

func main() {
    fmt.Println(doThings(1, 11, 101))
}
```

Statements

```
t := x
switch {
case x == 0:
    return "0"
case x < 0:
    t = -x
}
var s [32]byte
i := len(s)
for t != 0 { // Look, ma, no ()'s!
    i--
    s[i] = digits[t%base]
    t /= base
}
if x < 0 {
    i--
    s[i] = '-'
}
return string(s[i:])
```

Go statements

Cleanups:

- No semicolons
- Multiple assignments
- ++ and – are statements
- No parentheses around conditions; curly braces mandatory
- Implicit break in switch; explicit fallthrough

New:

- for range
- type switch
- go, select
- defer

Assignments

Assignments may assign multiple values simultaneously:

```
a, b = x, y
```

Equivalent to:

```
t1 := x
t2 := y
a = t1
b = t2
```

For instance:

```
a, b = b, a       // swap a and b
i, err = atoi(s)  // assign results of atoi
i, _ = atoi(991)  // discard 2nd value
```

Switch statements

Switch statements may have multiple case values, and break is implicit:

```
switch day {
case 1, 2, 3, 4, 5:
    tag = "workday"
case 0, 6:
    tag = "weekend"
default:
    tag = "invalid"
}
```

The case values don’t have to be constants:

```
switch {
case day < 0 || day > 6:
    tag = "invalid"
case day == 0 || day == 6:
    tag = "weekend"
default:
    tag = "workday"
}
```

For loops

```
for i := 0; i < len(primes); i++ {
    fmt.Println(i, primes[i])
}
```

A range clause permits easy iteration over arrays and slices:

```
for i, x := range primes {
    fmt.Println(i, x)
}
```

Unused values are discarded by assigning to the blank (_) identifier:

```
var sum int
for _, x := range primes {
    sum += x
}
```

Dependencies

Dependencies in Go

An import declaration is used to express a dependency on another package:

```
import "net/rpc"
```

Here, the importing package depends on the Go package "rpc".

The import path ("net/rpc") uniquely identifies a package; multiple packages may have the same name, but they all reside at different locations (directories).

By convention, the package name matches the last element of the import path (here: "rpc").

Exported functionality of the rpc package is available via the package qualifier (rpc):

```
rpc.Call()
```

A Go import declaration takes the place of a C include.

Naming: An excursion

How names work in a programming language is critical to readability.

Scopes control how names work.

Go has very simple scope hierarchy:

- universe
- package
- file (for imports only)
- function
- block

Locality of names

- Upper case names are exported: Name vs. name.
- The package qualifier is always present for imported names.
- (First component of) every name is always declared in the current package.
- One of the best (and hardest!) decisions made in Go

Back to imports

Importing a package means reading a package's exported API.

This export information is self-contained. For instance:

- A imports B
- B imports C
- B exports contain references to C

B’s export data contains all necessary information about C. There is no need for a compiler to read the export data of C.

This has huge consequences for build times!

Immediate consequences for readability.

Dependencies in C

.h files are not self-contained.

As a result, a compiler ends up reading core header files over and over again.

ifdef still requires the preprocessor to read a lot of code.

No wonder it takes a long time to compile...

At Google scale: dependencies explode, are exponential, become almost non-computable.

A large Google C++ build can read the same header file tens of thousands of times!

Tools

A brief overview

Two compilers: gc, gccgo

Support for multiple platforms: x86 (32/64bit), ARM (32bit), Linux, BSD, OS X, ...

Automatic formatting of source code: gofmt

Automatic documentation extraction: godoc

Automatic API adaption: gofix

All (and more!) integrated into the go command.

Building a Go program

A Go program can be compiled and linked without additional build information (make files, etc.).

By convention, all files belonging to a package are found in the same directory.

All depending packages are found by inspecting the import paths of the top-most (main) package.

A single integrated tool takes care of compiling individual files or entire systems.

The go command

Usage:

```
go command [arguments]
```

Selected commands:

```
build       compile packages and dependencies
clean       remove object files
doc         run godoc on package sources
fix         run go tool fix on packages
fmt         run gofmt on package sources
get         download and install packages and dependencies
install     compile and install packages and dependencies
list        list packages
run         compile and run Go program
test        test packages
vet         run go tool vet on packages
```

golang.org/cmd/go/

AMA / Drinks Break

Object-oriented programming

What is object-oriented programming?

> "Object-oriented programming (OOP) is a programming paradigm using objects – usually instances of a class – consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. Many modern programming languages now support forms of OOP, at least as an option."

(Wikipedia)

OOP requires very little extra programming language support

We only need

- the notion of an Object,
- a mechanism to interact with them (Methods),
- and support for polymorphism (Interfaces).

> Claim: Data abstraction, encapsulation, and modularity are mechanisms in their own right, not OOP specific, and a modern language (OO or not) should have support for them independently.

Object-oriented programming in Go

Methods without classes

Interfaces without hierarchies

Code reuse without inheritance

Specifically:

- Any value can be an object
- Any type can play the role of a class
- Methods can be attached to any type
- Interfaces implement polymorphism.

Methods

```go
package main

import "fmt"

type Point struct{ x, y int }

func PointToString(p Point) string {
    return fmt.Sprintf("Point{%d, %d}", p.x, p.y)
}

func (p Point) String() string {
    return fmt.Sprintf("Point{%d, %d}", p.x, p.y)
}

func main() {
    p := Point{3, 5}
    fmt.Println(PointToString(p)) // static dispatch
    fmt.Println(p.String())       // static dispatch
    fmt.Println(p)
}
```

Methods can be attached to any type

```
package main

import "fmt"

type Celsius float32
type Fahrenheit float32

func (t Celsius) String() string            { return fmt.Sprintf("%g°C", t) }
func (t Fahrenheit) String() string         { return fmt.Sprintf("%g°F", t) }
func (t Celsius) ToFahrenheit() Fahrenheit { return Fahrenheit(t*9/5 + 32) }

func main() {
    var t Celsius = 21
    fmt.Println(t.String())
    fmt.Println(t)
    fmt.Println(t.ToFahrenheit())
}
```

Interfaces

```
type Stringer interface {
     String() string
}

type Reader interface {
     Read(p []byte) (n int, err error)
}

type Writer interface {
     Write(p []byte) (n int, err error)
}

type Empty interface{}
```

- An interface defines a set of methods.
- A type that implements all methods of an interface is said to implement the interface.
- All types implement the empty interface interface{}.

Dynamic dispatch

```
type Stringer interface {
    String() string
}

var v Stringer
var corner = Point{1, 1}
var boiling = Celsius(100)

v = corner
fmt.Println(v.String()) // dynamic dispatch
fmt.Println(v)

v = boiling.ToFahrenheit()
fmt.Println(v.String()) // dynamic dispatch
fmt.Println(v)
```

A value (here: corner, boiling) of a type (Point, Celsius) that implements an interface (Stringer) can be assigned to a variable (v) of that interface type.

Composition and chaining

Typically, interfaces are small (1-3 methods).

Pervasive use of key interfaces in the standard library make it easy to chain APIs together.

```
package io
func Copy(dst Writer, src Reader) (int64, error)
```

The `io.Copy` function copies by reading from any Reader and writing to any Writer.

Interfaces are often introduced ad-hoc, and after the fact.

There is no explicit hierarchy and thus no need to design one!

cat

```go
package main

import (
    "flag"
    "io"
    "os"
)

func main() {
    flag.Parse()
    for _, arg := range flag.Args() {
        f, err := os.Open(arg)
        if err ≠ nil {
            panic(err)
        }
        defer f.Close()
        _, err = io.Copy(os.Stdout, f)
        if err ≠ nil {
            panic(err)
        }
    }
}
```

Interfaces in practice

Methods on any types and ad hoc interfaces make for a light-weight OO programming style.

Go interfaces enable post-facto abstraction.

No explicit type hierarchies.

"Plug'n play" in a type-safe way.

Concurrency

What is concurrency?

Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

Concurrency is not parallelism

Concurrency is not parallelism, although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

On the other hand, a well-written concurrent program might run efficiently in parallel on a multiprocessor. That property could be important...

For more on that distinction, see the link below. Too much to discuss here.

golang.org/s/concurrency-is-not-parallelism

A model for software construction

- Easy to understand.
- Easy to use.
- Easy to reason about.
- You don’t need to be an expert!
- (Much nicer than dealing with the minutiae of parallelism (threads, semaphores, locks, barriers, etc.))
- There is a long history behind Go’s concurrency features, going back to Hoare’s CSP in 1978 and even Dijkstra’s guarded commands (1975).

Basic Examples

A boring function

We need an example to show the interesting properties of the concurrency primitives.

To avoid distraction, we make it a boring example.

```
func main() {
    f("Hello, World", 500*time.Millisecond)
}
```

```
func f(msg string, delay time.Duration) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(delay)
    }
}
```

Ignoring it

- The go statement runs the function as usual, but doesn't make the caller wait.
- It launches a goroutine.

The functionality is analogous to the & on the end of a shell command.

```
func main() {
    go f("three", 300*time.Millisecond)
    go f("six", 600*time.Millisecond)
    go f("nine", 900*time.Millisecond)
}
```

Ignoring it a little less

When main returns, the program exits and takes the function f down with it.

We can hang around a little, and on the way show that both main and the launched goroutine are running.

```
func main() {
    go f("three", 300*time.Millisecond)
    go f("six", 600*time.Millisecond)
    go f("nine", 900*time.Millisecond)
    time.Sleep(3 * time.Second)
    fmt.Println("Done.")
}
```

Goroutines

What are they?

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

Declaring and initializing.

```
var c chan int
c = make(chan int)
// or
c := make(chan int)
```

Sending on a channel.

```
c ← 1
```

Receiving from a channel.

```
// The "arrow" indicates the direction of data flow.
value = ←c
```

Using channels

A channel connects the main and f goroutines so they can communicate.

```
func main() {
    c := make(chan string)
    go f("three", 300*time.Millisecond, c)
    for i := 0; i < 10; i++ {
        fmt.Println("Received", <-c) // Receive expression is just a value.
    }
    fmt.Println("Done.")
}
```

Declartion of `f`

```
func f(msg string, delay time.Duration, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Any suitable value can be sent.
        time.Sleep(delay)
    }
}
```

Synchronization

- When the main function executes `<-c`, it will wait for a value to be sent.
- Similarly, when the function f executes `c<-value`, it waits for a receiver to be ready.

A sender and receiver must both be ready to play their part in the communication. Otherwise we wait until they are.

Thus channels both communicate and synchronize.

Channels[1] can be unbuffered or buffered.

```
buffered := make(chan int, 5)
unbuffered := make(chan int)
```

---

1. Buffered Channels ↩

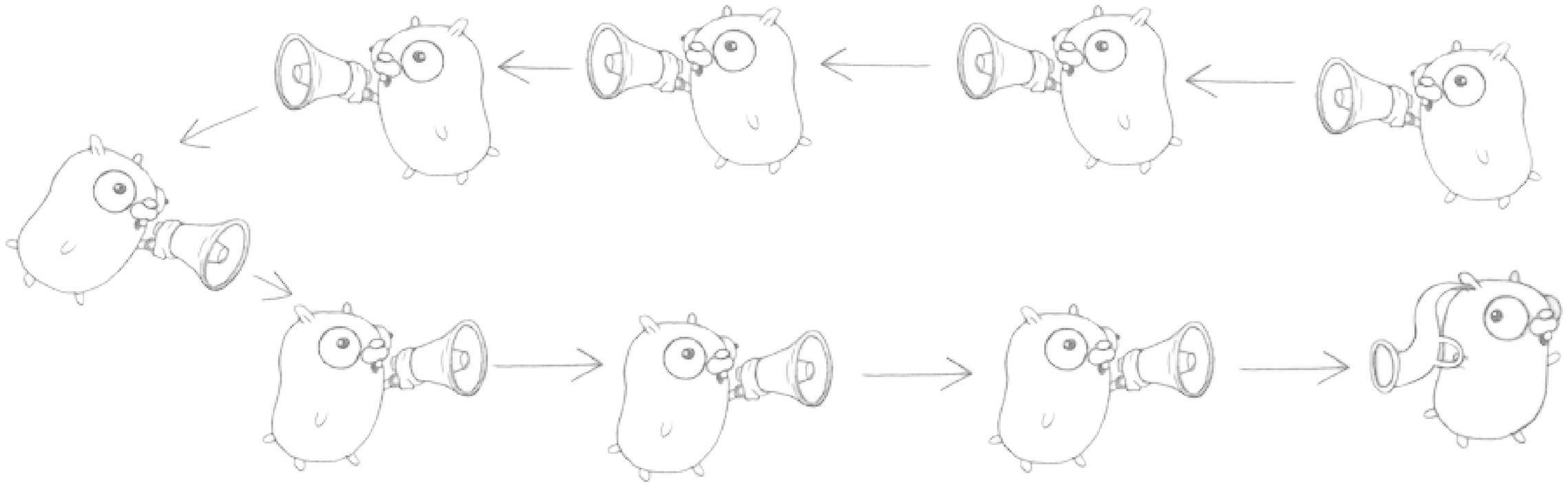
Using channels between many goroutines

```
func main() {
    c := make(chan string)
    go f("three", 300*time.Millisecond, c)
    go f("six", 600*time.Millisecond, c)
    go f("nine", 900*time.Millisecond, c)
    for i := 0; i < 10; i++ {
        fmt.Println("Received", <-c)
    }
    fmt.Println("Done.")
}
```

A single channel may be used to communicate between many (not just two) goroutines; many goroutines may communicate via one or multiple channels.

This enables a rich variety of concurrency patterns.

Chain of gophers



Ok, I’m just bragging here

Chain of gophers

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    start := time.Now()
    const n = 1000
    leftmost := make(chan int)

    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }

    go func(c chan int) { c <- 0 }(right)

    fmt.Println(<-leftmost, time.Since(start))
}
```

A matching producer and consumer

```
func producer(out chan int) {
    for order := 0; ; order++ {
        out ← order // Produce a work order.
    }
}
```

```
func consumer(in chan []int, n int) {
    for i := 0; i < n; i++ {
        result := ←in // Consume a result.
        fmt.Println("Consumed", result)
    }
}
```

The producer produces and endless supply of work orders and sends them out.

The consumer receives n results from the in channel and then terminates.

Putting it all together

```
func main() {
    start := time.Now()

    in := make(chan int)    // Channel on which work orders are received.
    out := make(chan []int) // Channel on which results are returned.
    go producer(in)
    // Launch 10 workers.
    for i := 0; i < 10; i++ {
        go worker(in, out)
    }
    consumer(out, 100)

    fmt.Println(time.Since(start))
}
```

A ready worker will read the next order from the in channel and start working on it. Another ready worker will proceed with the next order, and so forth.

Because we have many workers and since different orders take different amounts of time to work on, we see the results coming back out-of-order.

On a multi-core system, many workers may truly run in parallel.

The Go approach

> Don’t communicate by sharing memory, share memory by communicating.

Error Handling

Errors & Panics

Error indicates that something bad happened, but it might be possible to continue running the program.

```
func openScanner(filename string) (*bufio.Scanner, error) {
    f, err := os.Open(filename)
    if err ≠ nil {
        return nil, err
    }

    return bufio.NewScanner(f)
}
```

Panic happens at run time, something happened that was fatal to your program and program stops execution.

```
func main() {
    filename := os.Args[0]
    f, err := os.Open(filename)
    if err ≠ nil {
        panic(err)
    }
    defer f.Close()
}
```

Recovering

Panic is called during a run time error and fatally kill the program

When the function F calls panic, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller.

Recover[1] is a built-in function that regains control of a panicking goroutine. It tells Go what to do when that happens. Returns what was passed to panic. Recover must be paired with defer, which will fire even after a panic

```
func f() {
    defer func() {
        if r := recover(); r ≠ nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}
```

1. Defer-Panic-Recover ↩

In conclusion

- Go is simple, consistent, readable, and fun.
- All types are equal: methods on any type.
- Implicit satisfaction of interfaces makes code easier to reuse.
- Use composition instead of inheritance.
- Struct embedding is a powerful tool.
- Concurrency is awesome, and you should check it out.

What to do next?

- Complete the tour of Go tour.golang.org
- Find more about Go on golang.org
- Try out some examples https://gobyexample.com/
- Follow along tutorials https://go.dev/doc/
- Read blogs https://go.dev/blog/
- Explore the standard library https://pkg.go.dev/std
- Build something using awesome tools https://github.com/avelino/awesome-go

Uncover Truth About

- Ranging and Closing Channels
- `New` Keyword
- Blocking Calls
- Importand Packages like `net/http`, `bufio`, `os`, `fmt`, `io`, etc
- Testing in Go
- Modules

Thank You