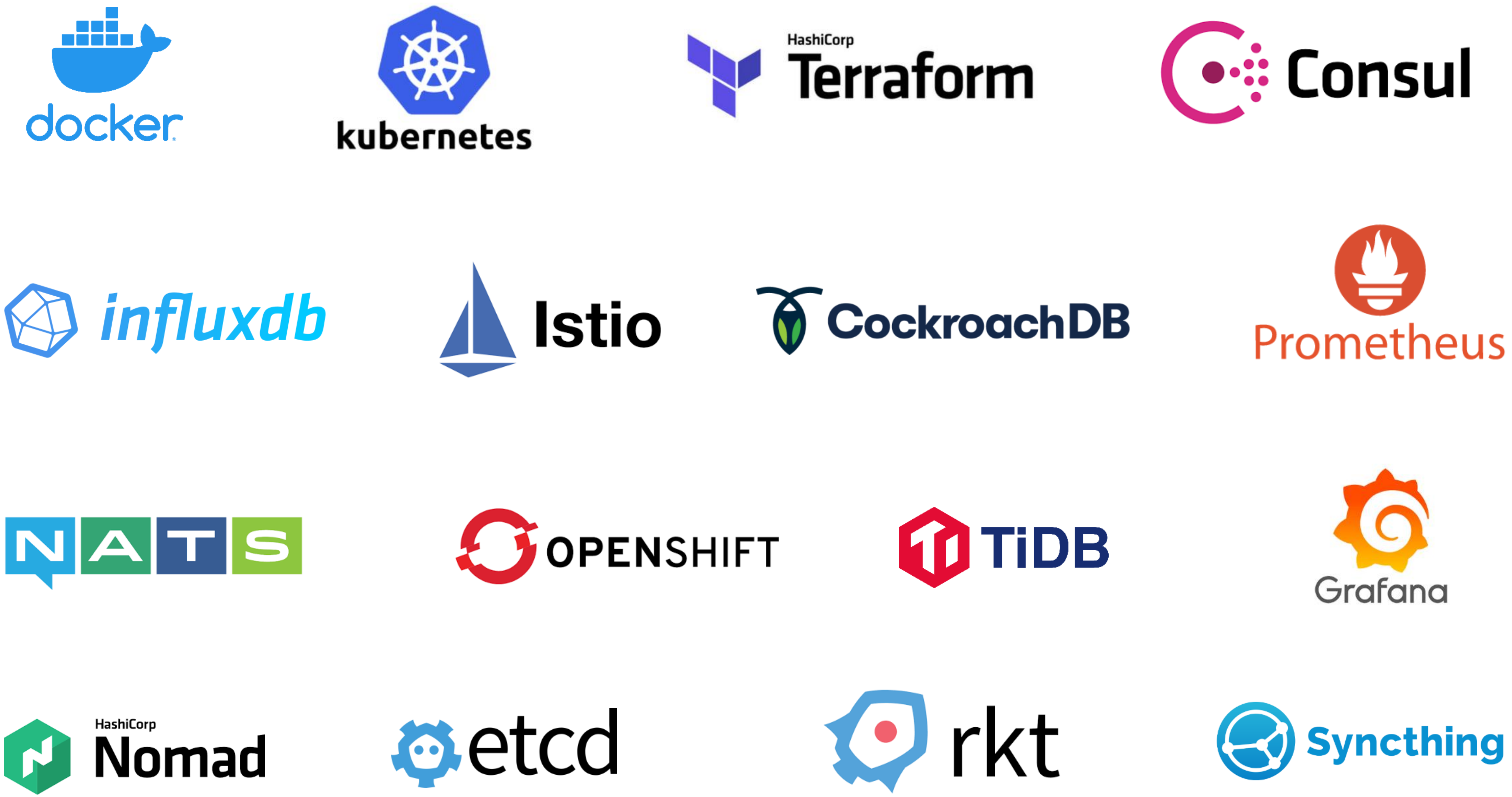
Overview of Go Language

Aniruddha Chakrabarti

Associate Director, EY, Tech Consulting

ani.c@outlook.com | linkedin.com/in/aniruddhac | slideshare.net/aniruddha.chakrabarti | Twitter - anchakra

What is common across all these?



Start with C, remove complex parts

add interfaces, concurrency

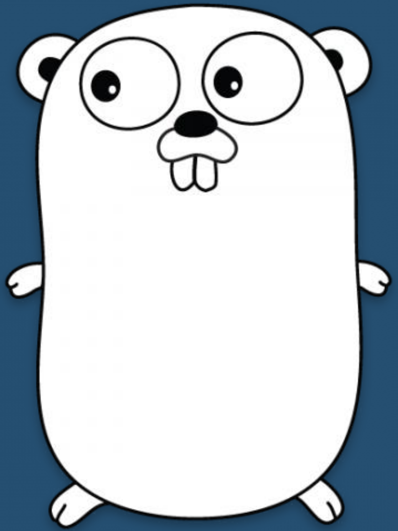
also: garbage collection, closures, reflection, strings, ...

Critics:

There's nothing new in Go

Lack of language features we got used to

- Lack of Function Overloading and Default Values for Arguments.
- Lack of Class based object orientation, inheritance
- **Generics** (yet)
- Dependency Management (somehow later introduced with Go Modules)
- Exceptions for Error Handing
- *Many other advanced features supported by other languages*



Go Language Designers: The task of programming language designer **"is consolidation not innovation"** – *Hoare, 1973*

Less is exponentially more – *Rob Pike, Go Designer*

Do Less, **Enable More** – *Russ Cox, Go Tech Lead*

Agenda

- What is Go
- History of Go
- Who uses Go, where Go is/could be used
- Overview of features
  - Types and Variables
  - Control Structure
  - Array, Slice and Map
  - Functions
  - Structs, Methods and Interfaces
  - Pointers (No pointer arithmetic)
  - Concurrency
  - Testing
  - Modules
  - Packages & Core Packages
- Resources

Why we need another new programming language?

No major systems language has emerged in over a decade, but over that time the computing landscape has changed tremendously. There are several trends:

- Computers are enormously quicker but software development is not faster.
- Dependency management is a big part of software development today but the “header files” of languages in the C tradition are antithetical to clean dependency analysis—and fast compilation.
- There is a growing rebellion against cumbersome type systems like those of Java and C++, pushing people towards dynamically typed languages such as Python and JavaScript.
- Some fundamental concepts such as garbage collection and parallel computation are not well supported by popular systems languages.
- The emergence of multicore computers has generated worry and confusion.

It's worth trying again with a new language, a concurrent, garbage-collected language with fast compilation. Regarding the points above:

- It is possible to compile a large Go program in a few seconds on a single computer.
- Go provides a model for software construction that makes dependency analysis easy and avoids much of the overhead of C-style include files and libraries.
- Go's type system has no hierarchy, so no time is spent defining the relationships between types. Although Go has static types it feels lighter weight than in typical OO languages.
- Go is fully garbage-collected and provides fundamental support for concurrent execution and communication.
- By its design, Go proposes an approach for the construction of system software on multicore machines.

What is Go (golang)

Go (also called golang) is an open source programming language that makes it easy to build simple, reliable, and efficient software.

- Go is **natively compiled** (Go does not use a VM, Go programs gets compiled directly to machine code like C, C++)
- Go is **garbage collected** (No memory management as needed in C/C++)
- Uses **static typing** (types could be inferred though)
- Scalable to large systems
- Though it's general purpose programming languages, but it's targeted towards **System programming** and **server side programming** (similar to C, C++, Rust, D)
- Clean syntax
- Has excellent **support for concurrency and networking**.
- Comes with a **rich standard library**
- gc compiler is available on Linux, OS X, Windows, various BSD & Unix versions
- Go is open source

Essence of Go

Go is **simple**

Go supports a **limited set of very well understood language features** (rather than trying to support everything)

“Go is a wise, clean, insightful, fresh thinking approach to the greatest-hits subset of the well understood” – Michael T.Jones

Programming in Go is productive and fun!

Excellent Tooling

Challenges with Go

- Does not support Generics
- Error handing can become challenging
- Probably too simple!
- Class less object orientation could be difficult to understand

A bit of history

- Originated Go was an experiment by Robert Griesemer, Rob Pike and Ken Thompson at Google, to design a new system programming language in 2007
- Language designers cited their shared dislike of C++'s complexity as a reason to design a new language

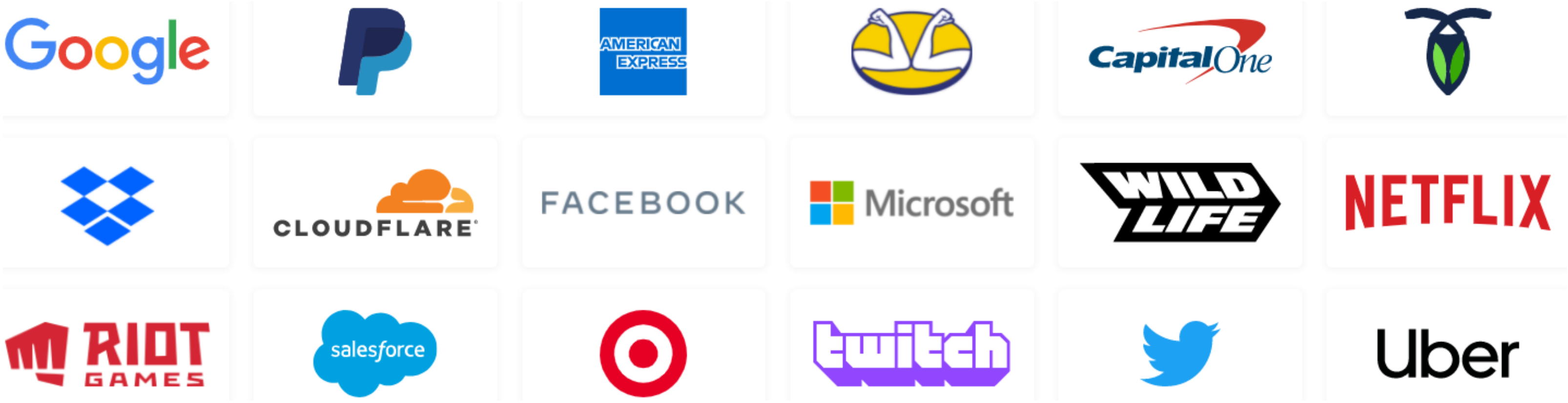
- Was announced officially in **November 2009**; it is used in some of Google's production systems, as well as by other firms.
- Go 1 (March 2012) – Version 1.0 release
- *Go 1.1 (May 2013)*
- *Go 1.2 (December 2013)*
- *Go 1.3 (June 2014)*

…

- Go 1.16 (Feb 2021) – current stable version

```
D:\Work\Play>go version
go version go1.16.7 windows/amd64
```

Who uses Go



- Many Google web properties and systems including YouTube, Kubernetes containers and download server dl.google.com
- Docker, a set of tools for deploying Linux containers
- Dropbox, migrated some of their critical components from Python to Go
- SoundCloud, for many of their systems
- Cloud Foundry, a platform as a service (PaaS)
- Couchbase, Query and Indexing services within the Couchbase Server
- MongoDB, tools for administering MongoDB instances
- ThoughtWorks, some tools and applications around continuous delivery and instant messages
- SendGrid, a transactional email delivery and management service
- The BBC, in some games and internal projects
- Novartis, for an internal inventory system

- Wikipedia - https://en.wikipedia.org/wiki/Go_(programming_language)#Notable_users
- Complete list - https://github.com/golang/go/wiki/GoUsers

Use Cases/ Domains where Go is popular

Distributed systems

- Go is ideal for building distributed systems
- Supports concurrency, memory safely, compiles to native code.
- AKS
- Many CNCF projects
- [Go, for Distributed Systems (golang.org)]

Cloud & Networking

- Many Cloud Native tools are written in Go – Docker, K8s, Prometheus
- Many cloud platforms supports SDKs in Go - GCP, AWS and Azure
- Cloud – Many PaaS Cloud platforms supports hosting Go code - GCP, Cloud Foundry, Heroku.

Command Line Tools

- Programs written in Go run on any system without requiring any existing libraries, runtimes, or dependencies.
- Programs written in Go have an immediate startup time
- GitHub command line tool, MongoDB

Web Development

- Go ships with an easy to use, secure and performant web server and includes it own web templating library
- Go has excellent support for all of the latest technologies from HTTP/2, to databases like MySQL, MongoDB and ElasticSearch

DevOps & SRE

- Many popular DevOps and SRE tools are written in Go -
- Docker, K8s, etcd, Istio, Terraform, Drone

Scripting

- Cloud – Many PaaS Cloud platforms supports hosting Go code including GCP, Cloud Foundry (with build pack) and Heroku. Many cloud platforms supports SDKs in Go including GCP, AWS and Azure

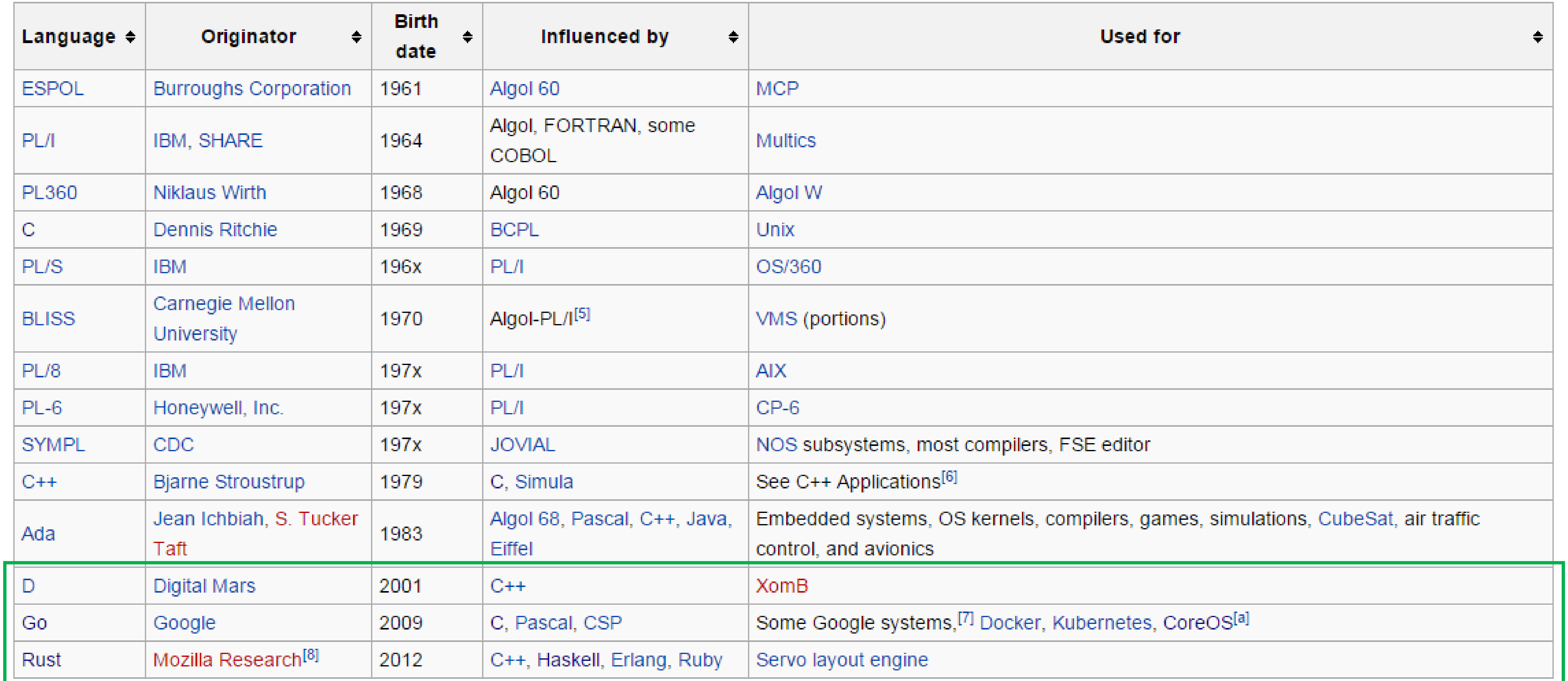
Internet, Open Source

- Google
- Facebook
- Netflix
- Dropbox
- Microsoft
- Twitter
- Uber

Enterprise Systems

- Enterprises are adopting Golang in a big way across domains -
- CapitalOne
- American Express
- Paypal
- Target

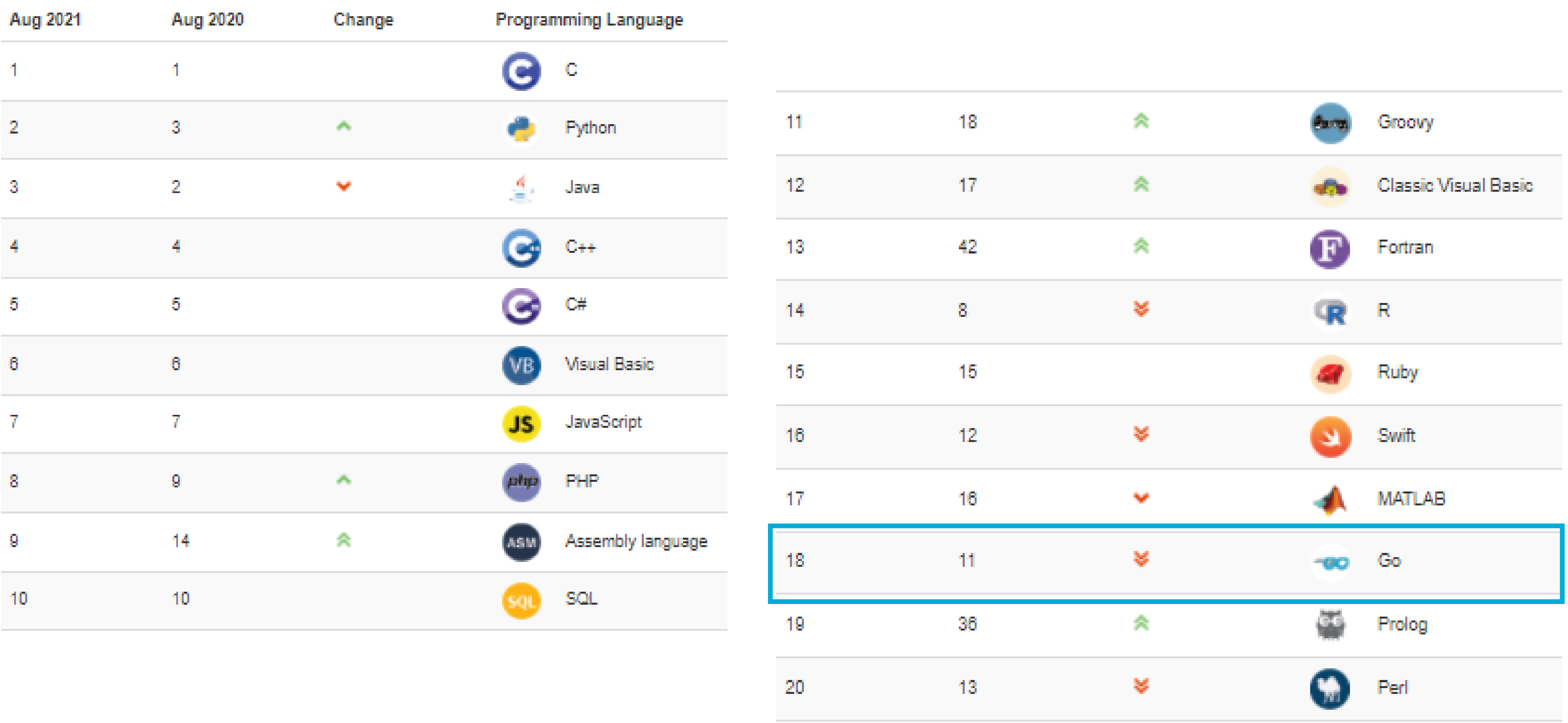
Other System Programming Languages



Language	Originator	Birth date	Influenced by	Used for
ESPOL	Burroughs Corporation	1961	Algol 60	MCP
PL/I	IBM, SHARE	1964	Algol, FORTRAN, some COBOL	Multics
PL360	Niklaus Wirth	1968	Algol 60	Algol W
C	Dennis Ritchie	1969	BCPL	Unix
PL/S	IBM	196x	PL/I	OS/360
BLISS	Carnegie Mellon University	1970	Algol-PL/I[5]	VMS (portions)
PL/8	IBM	197x	PL/I	AIX
PL-6	Honeywell, Inc.	197x	PL/I	CP-6
SYMPL	CDC	197x	JOVIAL	NOS subsystems, most compilers, FSE editor
C++	Bjarne Stroustrup	1979	C, Simula	See C++ Applications[6]
Ada	Jean Ichbiah, S. Tucker Taft	1983	Algol 68, Pascal, C++, Java, Eiffel	Embedded systems, OS kernels, compilers, games, simulations, CubeSat, air traffic control, and avionics
D	Digital Mars	2001	C++	XomB
Go	Google	2009	C, Pascal, CSP	Some Google systems,[7] Docker, Kubernetes, CoreOS[a]
Rust	Mozilla Research[8]	2012	C++, Haskell, Erlang, Ruby	Servo layout engine

System Programming Languages developed after 2000

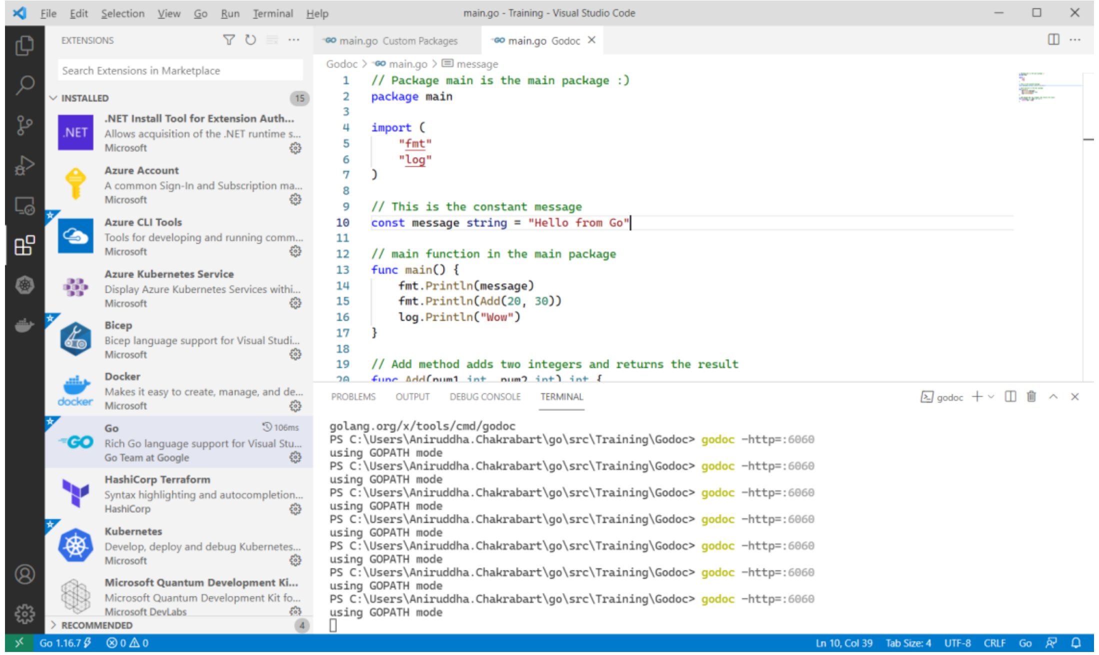
Popularity of Go is on the rise (TIOBE Index for August 2021)



Aug 2021	Aug 2020	Change	Programming Language
1	1		C
2	3	↑	Python
3	2	↓	Java
4	4		C++
5	5		C#
6	6		Visual Basic
7	7		JavaScript
8	9	↑	PHP
9	14	↑↑	Assembly language
10	10		SQL
11	18	↑↑	Groovy
12	17	↑↑	Classic Visual Basic
13	42	↑↑	Fortran
14	8	↓↓	R
15	15		Ruby
16	12	↓↓	Swift
17	16	↓	MATLAB
18	11	↓↓	Go
19	38	↑↑	Prolog
20	13	↓↓	Perl

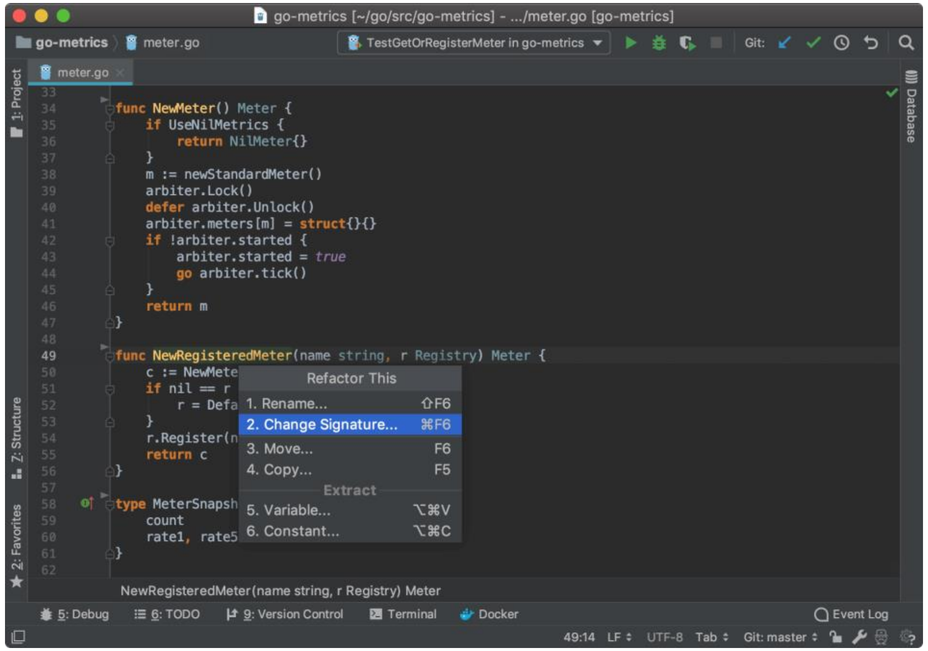
Install and Setup

- Install Go from https://golang.org/dl/ - available in installer or zip file or source code format. Available for Windows, macOS and Linux
- Install additional tools/ frameworks like godoc
- Install an IDE and Go extensions
  - VS Code (with Go extension from Google + others)
  - GoLand (IntelliJ)


main.go - Training - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXTENSIONS
Search Extensions in Marketplace
INSTALLED 15
.NET Install Tool for Extension Auth...
Azure Account
Azure CLI Tools
Azure Kubernetes Service
Bicep
Docker
Go
HashiCorp Terraform
Kubernetes
Microsoft Quantum Development Ki...
RECOMMENDED 4
main.go Custom Packages
main.go Godoc
Godoc > main.go > message
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
godoc
Go 1.16.7
Ln 10, Col 39 Tab Size: 4 UTF-8 CRLF Go

```go
// Package main is the main package :)
package main

import (
    "fmt"
    "log"
)

// This is the constant message
const message string = "Hello from Go"

// main function in the main package
func main() {
    fmt.Println(message)
    fmt.Println(Add(20, 30))
    log.Println("Wow")
}

// Add method adds two integers and returns the result
func Add(num1 int, num2 int) int {
```

```
golang.org/x/tools/cmd/godoc
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Godoc> godoc -http=:6060
using GOPATH mode
```



```go
func NewMeter() Meter {
    if UseNilMetrics {
        return NilMeter{}
    }
    m := newStandardMeter()
    arbiter.Lock()
    defer arbiter.Unlock()
    arbiter.meters[m] = struct{}{}
    if !arbiter.started {
        arbiter.started = true
        go arbiter.tick()
    }
    return m
}

func NewRegisteredMeter(name string, r Registry) Meter {
    c := NewMete
    if nil == r
        r = Defa
    }
    r.Register(n
    return c
}

type MeterSnapsh
    count
    rate1, rate5
}
```

NewRegisteredMeter(name string, r Registry) Meter

- Check GOROOT (where Go is installed) and GOPATH (Go source path) Env variables
- Configure your workspace for dev - How to Write Go Code - The Go Programming Language (golang.org)

First Go Program

- Every Go Program should contain a package called main.
- Package statement should be the first line of any go source file

- Entry point of a Go program should be the main function of main package

```
// name of the source code file is main.go (it could be whatever you like)
// this package is called main
package main

// Entry point of a Go program is main.main i.e. main function of main package
func main (){
    // println in built function is called
    println("Hello from Go")         // prints Hello from Go in console
}
```

- To compile and run the go program use **`go run`** command

```
go run main.go
```

- To generate the binary exe use **`go build`** command

```
go build
```

First Go Program (Cont'd)

- Packages could be imported via import statement

```
// this package is called main
package main

// fmt package contains methods to interact with console like Print and Scan
import "fmt"

// Entry point of a Go program is main.main i.e. main function of main package
func main (){
    // Println method of package fmt is called
    fmt.Println("Hello from Go")            // prints Hello from Go in console
}
```

- The start curly has to be in the same line of method name and paran -

```
func main ()
{
    fmt.Println("Hello from Go")
}
// this code would not compile
```

Few important Go tools

Development

- Running code – go run
- Fetching dependencies / packages – go get
- Formatting / Refactoring code – gofmt
- Static code analysis – go vet
- Linting – go lint
- Viewing documentation – go doc
- Creating documentation - godoc

```
go lint .

go doc fmt
go doc fmt Println

gofmt -w -s -d main.go
```

Testing

- Performing tests – go test
- Profiling Test Coverage – go test –cover
- Testing all Dependencies – go test all

```
go test .

go test -cover ./...

go test all
```

Build & Deployment

- Build an exe – go build
- Cross compilation -

```
go build

GOOS=linux GOARCH=amd64
go build
```

Other Go Commands

- go version – shows version information
- go gopath – prints gopath env variable
- go env – Lists all go env variables
- go list – Lists all go packages / modules installed in the system

An Overview of Go's Tooling - Alex Edwards

Comments and Semicolon

- Go supports C/C++/Java style single line and multi line comments

```
// this package is called main
package main

// fmt package contains methods to interact with console like Print and Scan
import "fmt"

/*
Entry point of a Go program is main.main i.e.
main function of main package
*/
func main (){
    // Println method of package fmt is called
    fmt.Println("Hello from Go")
}
```

- Semicolons are not required in Go

fmt package – interacting with console

- Print – Prints to standard output / console
- Printf – Prints formatted output to console. Similar to C Printf
- Println - Prints to standard output / console. Adds a line break after
- Scan - Scans the input texts which is given in the standard input, reads from there and stores in a variable
- Scanf - Scanf scans text read from standard input, storing successive space-separated values into successive arguments as determined by the format.
- Scanln - Scanln is similar to Scan, but stops scanning at a newline and after the final item there must be a newline or EOF

```
func main(){
    fmt.Printf("Name - %s, Designation - %s, Age - %d \n", "Satya", "CEO", 52)

    var input string
    var err error
    _, err = fmt.Scanln(&input)

    if err != nil {
        fmt.Println("Error - ", err)
    } else{
        fmt.Println("You entered - ", input)
    }
}
```

Packages

```go
// Every Go Program should contain a package called main
// Package statement should be the first line of any go source file

package main

import "fmt"
```

- Every Go program is made up of packages.
- Programs start running in package main.

Import

```
// Every Go Program should contain a package called main
// package statement should be the first line of any go source file
package main

import "fmt"
// fmt package includes functions related to formatting and output to the screen
```

- Other external packages could be imported to be used
- Multiple packages could be imported using a shorthand syntax

```
import (
    "fmt"
    "time"
)
```

Same as

```
import "fmt"
import "time"
```

Core Packages

- Go comes with a large no of packages that provides common functionality like File handling, IO, String handling, Cryptography etc.

Sl. No	Description	Package Name
1	String manipulation	strings
2	Input & Output	io, bytes
3	Files & Folders	os, path/filepath
4	Errors	errors
5	Containers & Sort	container/list
6	Hashes & Cryptography	hash, crypto
7	Encoding	encoding/sob
8	Allows interacting with Go's runtime system, such as functions to control goroutines	runtime
9	Synchronization Primitives	sync
10	Server communication, RPC, HTTP, SMTP etc.	net, http/rpc/jsonrpc
11	Math library	math
12	Zip, Archive, Compress	archive, compress
13	Database related	database / sql
14	Debugging	debug
15	Automated testing	testing

Using popular packages

- Imports popular Go packages like time, os, math and math/rand and utilizes members of these packages
- Alias could be used for imported packages

```
package main

import (
        "fmt"
        "math"
        "math/rand"
        "os"
        s "strings"                      // s is used as alias for package strings
        "time"
)

func main() {
        fmt.Println(time.Now())          // Prints current date and time
        fmt.Println(os.Hostname())       // Prints name of the machine
        fmt.Println(math.Pow(2, 3))      // Prints 2 to the power 3
        fmt.Println(rand.Intn(100))      // Prints a random number between 0 and 100

        fmt.Println(s.ToUpper("bangalore"))        // Alias is used instead of full package name
}
```

Variables

- Go is statically typed
- Implicitly defined variable – type is inferred by Go compiler

```
var message = "Hello from Go"        // message would be of type string
```

- Explicitly defined variable – type is specified explicitly

```
var message string = "Hello from Go"
```

- Multiple variables could be defined together

```
var x, y int          // both x and y are defined as int
```

- Multiple variables could be defined together and initialized

```
var x, y int = 5, 10
```

Or

```
var(
    name = "Go"
    age = 5
    isGood = true
)
```

Type declaration

- Go’s type declaration is different from C style languages (C/ C++/ Java / C#) and is very similar to Pascal – variable / declared name appears before the type

C Style Languages :

```
int ctr = 10
string message = "Hello"
```

Pascal :

```
var ctr : int = 10
var message : string = "Hello"
```

Go :

```
var message string = "Hello from Go"       // Explicit type declaration
var message = "Hello from Go"              // Implicit type declaration - Type inferred
```

Variables (cont'd)

- Within a function variables could be defined using a shorthand syntax without using var keyword
- Shorthand syntax uses := rather than =

Option 1 – Explicit type declaration

```
func main(){
    message string := "Hello World"          // var not used, := is used instead of =
    fmt.Printf(message + "\n")
}
```

Option 2 – Type inferred

```
func main(){
    message := "Hello World"          // var not used, := is used, type inferred as string
    fmt.Printf(message + "\n")
}
```

Constants

- const declares a constant value, that can not change
- A const statement can appear anywhere a var statement can.

```
const PI float32 = 3.14
fmt.Println(PI)

PI = 3.15 // does not compile and shows a compile time error
```

- Variables or Constants declared in a Go program need to be used, otherwise the compiler issues an error (not warning)

```
package main
import (
    "fmt"
)

func main() {
    msg := "Hello"
    fmt.Print("Hello from Go")
}
```

Error - .\main.go:8:2: msg declared but not used

Types

- **Numbers**
  - Integer
    - Signed - int, int16, int32, int64
    - Unsigned - uint8, uint16, uint32, uint64, int8
    - uint means “unsigned integer” while int means “signed integer”.
  - Float
    - float32, float64
  - Complex
    - complex64, complex128
- **String**
- **Boolean**
    - bool (true, false)

```go
package main

import "fmt"

func main() {
	var no1 int = 10
	var no2 int32 = -100000
	var no3 int32 = -1000000000

	fmt.Println(no1)
	fmt.Println(no2)
	fmt.Println(no3)

	var posNo1 uint = 10000
	var posNo2 int64 = 1000000000

	fmt.Print(posNo1)
	fmt.Print(posNo2)

	var float1 float32 = 123456.7890
	var float2 float32 = 99999999999999.99999999999999

	fmt.Print(float1)
	fmt.Print(float2)

	var compl1 complex64 = 123 + 5i
	fmt.Println(compl1)
}
```

Scope

- Go supports package level scoping, function level scoping and block level scoping

Block Level Scoping:

```
y := 20
fmt.Println(y)                 // 20

{
    y := 10
    fmt.Println(y)             // 10
}

fmt.Println(y)                 // 20
```

Collection Types -

Array, Slice and Map

Array (cont'd)

- Array is a numbered sequence of elements of fixed size
- When declaring the array typically the size is specified, though alternately compiler can infer the length

```
var cities[3] string            // Syntax 1 of declaring arrays
var cities [3]string            // Syntax 2 of declaring arrays

cities[0] = "Kolkata"
cities[1] = "Chennai"
cities[2] = "Blore"

fmt.Println(cities[2])          // Blore

cities[2] = "Minneapolis"
fmt.Println(cities[2])          // Minneapolis
```

- Array size is fixed – it could not be changed after declaring it

```
cities[3] = "Amsterdam"
// Does not compile - invalid array index 3 (out of bounds for 3-element array)
```

Array (cont’d)

- Arrays could be declared and initialized in the same line

```
// Option 2 - declaring and initialing the array in the same line

cities := [3]string {"Kolkata", "Chennai", "Blore"}

fmt.Println(cities[2])       // Blore

cities[2] = "Minneapolis"
fmt.Println(cities[2])       // Minneapolis
```

- Go compiler can calculate the length of the array if not specified explicitly

```
cities := [...]string {"Kolkata", "Chennai", "Blore"}

fmt.Println(cities[2])       // Blore
fmt.Println(len(cities))     // 3
```

Array (cont’d)

- Determining the length of the array

```
// builtin len returns the length of an array
fmt.Println(len(cities))                    // 2
```

- Iterating through the array using range – **range** returns index and value of the elements

```
for index,value := range cities {
    fmt.Printf("At position %d, the character %s is present\n", index, value)
}
```

- Using **blank identifier** to ignore values

```
for _,value := range cities {
    fmt.Printf("City - %s \n", value)
}
```

Slices

- An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array.
- Slices are a key data type in Go, giving a more powerful interface to sequences than arrays. Typically Slices are used in Go rather than array.
- Internally Go uses arrays for slices, but slices are easier to use and more effective

```
cities := []string {"Kolkata", "Bangalore", "Mumbai"}
// slices are declared with syntax similar to array. Only the length is not specified.
// cities := [3]string {"Kolkata", "Bangalore", "Mumbai"} // Array declaration with length

fmt.Println(cities)                  // [Kolkata Bangalore Mumbai]
fmt.Println(cities[1])               // Bangalore

cities = append(cities,"Amsterdam")
cities = append(cities,"Den Haag")

fmt.Println(cities)                  // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
```

- Built in len function returns the length of a slice.

```
fmt.Println(len(cities))             // 5 is the length of the slice cities
```

Slices (cont’d)

- Built in cap function returns the length of a slice.

```
fmt.Println(cap(cities))     // 6 is the capacity of the slice, while 5 is the length
```

- Slices could also be defined by built-in make function.

```
cities := make([]string, 3)

cities[0] = "Kolkata"                // Allows to set values like Arrays
cities[1] = "Bangalore"
cities[2] = "Mumbai"

fmt.Println(cities)                  // [Kolkata Bangalore Mumbai]
fmt.Println(cities[1])               // Bangalore

cities = append(cities,"Amsterdam")
cities = append(cities,"Den Haag")

fmt.Println(cities)                  // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
fmt.Println(len(cities))             // 5
```

Slices (cont’d)

- Slices can also be copied/cloned using copy function

Kolkata	Bangalore	Mumbai	Amsterdam	Den Haag

```
fmt.Println(slice)                   // [Kolkata Bangalore Mumbai Amsterdam Den Haag]

duplicateSlice := make([]string, len(slice))
copy(duplicateSlice, slice)

fmt.Println(duplicateSlice)           // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
```

- Slices support a “slice” operator with the syntax slice[low:high] – same as List processing on other languages

```
fmt.Println(slice[0:2])              // [Kolkata Bangalore]
fmt.Println(slice[:3])               // [Kolkata Bangalore Mumbai]
fmt.Println(slice[2:])               // [Mumbai Amsterdam Den Haag]
```

https://blog.golang.org/go-slices-usage-and-internals

Map

- Map is one of the built in data structure that Go provides. Similar to hashes or dicts in other languages

```
employees := map[int]string {
    1 : "Rob Pike",
    2 : "Ken Thompson",
    3 : "Robert Griesemer",
}

fmt.Println(employees)       // map[1:Rob Pike 2:Ken Thompson 3:Robert Griesemer]

// Get a value for a key with name[key]
fmt.Println(employees[2])    // Robert Griesemer

// Set a value for a key with name[key]
emps[2] = "Satya Nadela"
fmt.Println(employees[2])    // Satya Nadela
```

Map

- Maps could be also declared using built in make function

```
// make(map[key-type]val-type)
emps := make(map[int]string)
emps[1] = "Bill"
emps[2] = "Satya"
emps[3] = "Sunder"
emps[4] = "Andrew"

fmt.Println(emps)            // map[1:Bill 2:Satya 3:Sunder 4:Andrew]

// Get a value for a key with name[key]
fmt.Println(emps[2])         // Satya

// Set a value for a key with name[key]
emps[2] = "Satya Nadela"
fmt.Println(emps[2])         // Satya Nadela
```

Map (cont’d)

```
// make(map[key-type]val-type)
emps := make(map[int]string)
emps[1] = "Bill"
emps[2] = "Satya"
emps[3] = "Sunder"
emps[4] = "Andrew"
```

- The builtin len returns the number of key/value pairs when called on a map

```
fmt.Println(len(emps))               // 4
```

- The builtin delete removes key/value pairs from a map

```
delete(emps, 1)                      // remove element with key 1
delete(emps, 2)                      // remove element with key 1

fmt.Println(emps)                    // map[3:Sunder 4:Andrew]
```

Function

Functions

- Functions are declared with `func` keyword
- Functions can take zero or more arguments and can return values

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

func displayMessage(message string){
    fmt.Println(message)
}

displayMessage("Hello")

func add(x int, y int) int {
    return x+y
}

var result int = add(20,10)
fmt.Println(result)
```

Functions (cont’d)

- When function arguments are of same type it could be shortened -

```
// func add(x int, y int) int {

func add(x, y int) int {
    return x+y
}

var result int = add(20,10)
fmt.Println(result)
```

Anonymous Functions / Lambdas

- Go supports Anonymous Functions or Lambdas
- Anonymous Functions could be used in two ways -
    - Assigned to a variable and invoked through variable
    - Could be called immediately after declaration

```
package main

import "fmt"

func main() {
        // anonymous function is assigned to a variable and invoked later
        add := func(x int, y int) int {
                return x + y
        }
        fmt.Println(add(20, 10))

        // anonymous function is invoked immediately after declaration
        res := func(x int, y int) int {
                return x - y
        }(30, 15)

        fmt.Println(res)
}
```

Anonymous Functions / Lambdas (Cont’d)

- Anonymous functions could be passed as an argument or could be returned from a function

```
// Passing anonymous function as argument to a function
func lambdaTest() {
        var res1 = performCalc(func(no1 int, no2 int) int { return no1 + no2 }, 200, 100)
        fmt.Println(res1)

        var res2 = performCalc(func(no1 int, no2 int) int { return no1 - no2 }, 200, 100)
        fmt.Println(res2)
}

func performCalc(f func(no1 int, no2 int) int, x int, y int) int {
        return f(x, y)
}
```

Anonymous Functions (cont'd)

- Anonymous Functions could be called immediately
- Very similar to JavaScript (and other functional languages)

```
func main(){
       func(msg string){
              fmt.Println("Hello " + msg)
       }("Aniruddha")
}
// Displays Hello Aniruddha

func main(){
       func(num1 int, num2 int){
              fmt.Println(num1 + num2)
       }(10, 20)
}
// Displays 30
```

In Brown – Anonymous Function Declaration

In Green - Anonymous Function Invocation

Higher order functions

- Anonymous Functions could be passed as argument to other functions, and could be returned from other functions. Functions behave like values – could be called function values

```
func main () {
        sayHello := func() {
                fmt.Println("Hello from Go")
        }
        doWork(sayHello)
}

func doWork(anonymous func() ){
        anonymous()
}

// Displays "Hello from Go"
```

Higher order functions (cont’d)

```
func main () {
       add := func(x int, y int) int {
             return x+y
       }

       sub := func(x int, y int) int {
             return x-y
       }

       result := doWork(add, 30, 20)
       fmt.Println(result)                   // 1300

       result = doWork(sub, 30, 20)
       fmt.Println(result)                   // 500
}

func doWork(anonymous func(int, int) int, num1 int, num2 int ) int{
       return anonymous(num1 * num1, num2 * num2)
}
```

Higher order functions - User-Defined Function types

```
func main () {
        add := func(x int, y int) int {
                return x+y
        }

        sub := func(x int, y int) int {
                return x-y
        }

        result := doWork(add, 30, 20)
        fmt.Println(result)                   // 1300

        result = doWork(sub, 30, 20)
        fmt.Println(result)                   // 500
}

type HigherFunc func(x int, y int) int        // User defined function type

func doWork(anonymous HigherFunc, num1 int, num2 int ) int{
        return anonymous(num1 * num1, num2 * num2)
}
```

Variadic Functions

- Variadic functions can be called with any number of trailing arguments.

```
func displayMessage(message string, times int, params ...string){
    fmt.Println(message, times, params)
}

displayMessage("Call1", 1)
displayMessage("Call2", 2, "Param1")
displayMessage("Call3", 3, "Param1", "Param2", "Param3", "Param4")
```

Output:

```
Call1 1 []
Call2 2 [Param1]
Call3 3 [Param1 Param2 Param3 Param4]
```

Control Structure -
If, For and Switch

Control Structures

- If
- For
- *For using range*
- Switch
- *Type Switch*

- *goto (infamous ?)*

Go does not supports while or do while keywords, though loops similar to while could be written using for

If, else if, else

- The if statement looks as it does in C or Java, except that the ( ) are gone and the { } are required.

```
var salary int = 100

if salary < 50 {
    fmt.Println("you are underpaid")
} else if salary >= 50 {
    fmt.Println("you are sufficiently paid")
} else {
    fmt.Println("you are overpaid")
}
```

For

- Go has only one looping construct, the for loop.
  - Go does not have while, do while or for each / for in loops
- The basic for loop looks as it does in C or Java, except that the ( ) are gone (they are not even optional) and the { } are required.

```
for ctr := 0; ctr < 10;  ctr++ {
    fmt.Println(ctr)
}
```

- Go does not support while or do while. Same could be achieved using for

```
var ctr int = 0

// same as while
for(ctr < 5) {
    fmt.Println(ctr)
    ctr++
}
```

For (cont’d)

- As in C or Java, you can leave the pre and post statements empty

```
ctr:=0
for ; ctr < 10;  {
    ctr +=1
    fmt.Println(ctr)
}
```

- Semicolons could be dropped: C's while is spelled for in Go

```
ctr:=0
for ctr < 10 {      // behaves in the same way as while ctr < 100 in C or Java
    ctr +=1
    fmt.Println(ctr)
}
```

- Endless or forever loop

```
for {
    // do something - this loop would never end
}
```

Switch

- Go’s switch statement is more general than C's - expressions need not be constants or even integers.

```
city := "Kolkata"

switch city {
case "Kolkata":
       println("Welcome to Kolkata")
       break
case "Bangalore":
       println("Welcome to Bangalore")
       break
case "Mumbai":
       println("Welcome Mumbai")
       break
}
```

```
rating := 2

switch rating {
case 4:
       println("You are rated Excellent")
       break
case 3:
       println("You are rated Good")
       break
case 2:
       println("You are rated Consistent")
       break
case 1:
       println("You need to improve a bit")
       break
}
```

Type switch

- Used to discover the dynamic type of an interface variable. Such a type switch uses the syntax of a type assertion with the keyword type inside the parentheses.
- If the switch declares a variable in the expression, the variable will have the corresponding type in each clause. It's also idiomatic to reuse the name in such cases, in effect declaring a new variable with the same name but a different type in each case.

```
type Human interface {
    Display()
}

type Employee struct {
    name string; designation string
}

func (emp Employee) Display(){
    fmt.Println(emp.name, emp.designation)
}

type Contractor struct {
    name string; weeklyHours int
}

func (cont Contractor) Display(){
    fmt.Println(cont.name, cont.weeklyHours)
}
```

```
func main() {
        var human Human
        human = Employee
{name:"Aniruddha",designation:"AVP"}

        switch human:= human.(type){
                default:
                        fmt.Println("default")
                case Employee:
                        fmt.Println("Human",
human.designation)
                case Contractor:
                fmt.Println("Cont", human.weeklyHours)
        }
}
```

Struct, Method and Interface

Types - struct

- structs are typed collections of named fields. Useful for grouping data together to form records.
- **type** keyword introduces a new type. It's followed by the name of the type (Employee), the keyword struct to indicate that we are defining a **struct** type and a list of fields inside of curly braces.
- **Go does not have Class, it supports Struct and Interfaces**.

```
type Employee struct {
    name string              // field name of type string
    age int                  // field age of type int
    salary float32           // field salary of type float32
    designation string       // field designation of type string
}
```

Struct - initialization

```
type Employee struct {
    name string; age int ; salary float32; designation string
}
```

Initialization Option 1 – Using new function

```
emp := new(Employee)
emp.name = "Ken Thompson"; emp.age = 50
emp.salary = 12345.678; emp.designation = "Distinguished Engineer"
```

Initialization Option 2 (more like JavaScript)

```
emp := Employee{}
emp.name = "Ken Thompson"; emp.age = 50
emp.salary = 12345.678; emp.designation = "Distinguished Engineer"
```

Initialization Option 3 – parameters should be in the same order fields are declared

```
emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished Engineer"}
fmt.Println(emp)
fmt.Println(emp.name)

// age and salary is not known and so not initialized
newEmp := Employee{name:"New Emp", designation:"Engineer"}
fmt.Println(newEmp.designation)
```

Struct (cont'd)

- structs can have arrays and other child structs as fields

```
type Employee struct{
    Name string
    Age int
    Salary float32
    Slills [4]string          // Array field
    HomeAddress Address       // Nested Child struct as property
}
type Address struct{
    StreetAddress string
    City string
    Country string
}

func main(){
    address := Address{"M G Road", "Bangalore", "IN"}
    skills := [4]string {"C","C++","Go","Rust"}
    emp := Employee{"Aniruddha", 40, 123.456, skills, address}

    fmt.Println(emp)   // {Aniruddha 40 123.456 [C Go Rust] {M G Road Bangalore IN}}
    fmt.Println(emp.Skills)                           // [C Go Rust]
    fmt.Println(emp.HomeAddress.StreetAddress)        // MG Road
}
```

Method

- Go supports *methods* defined on struct types.
- **Methods of the struct are actually defined outside of the struct declaration.**

```
type Employee struct {
}

// Method for struct Employee – defined outside of the struct declaration
// Since Display accepts a parameter of type Employee it’s considered a member of Employee
func (emp Employee) Display(){
    fmt.Println("Hello from Employee")
}

func main() {
    emp := Employee{}
    // method invokation
    emp.Display()            // displays "Hello from Employee"
}
```

Method – value receiver type

- Methods can be defined for either pointer or value receiver types.

```
type Employee struct {
    name string
    age int
    salary float32
    designation string
}

// Method for struct Employee – this is value receiver type
func (emp Employee) Display() {
    fmt.Println("Name:", emp.name, ", Designation:", emp.designation)
}

func main() {
    emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished Engineer"}
    emp.Display()
    // displays Name: Ken Thompson, Designation: Distinguished Engineer
}
```

Method parameters

- Methods can accept parameter similar to functions.

```
type Employee struct {
    name string
    age int
    salary float32
    designation string
}

// Method for struct Employee – this is value receiver type
func (emp Employee) Display(message string) {
    fmt.Println(message, emp.name, ", Designation:", emp.designation)
}

func main() {
    emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished Engineer"}
    emp.Display("Hello")
    // displays Hello Ken Thompson, Designation: Distinguished Engineer
}
```

Method - pointer receiver type

- Methods can be defined for either pointer or value receiver types.

```
type Employee struct {
    name string; age int
}

// Methods for struct Employee – this is pointer receiver type
func (emp* Employee) increaseAgeByOne(){
        emp.age++
}
func (emp* Employee) increaseAge(increaseBy int){
        emp.age += increaseBy
}

emp := Employee{"Ken Thompson", 40}
emp.increaseAgeByOne()
emp.display()                   // displays Ken Thompson, 41
emp.increaseAge(5)
emp.display()                   // displays Ken Thompson, 46
```

Method (cont’d)

```
type Rectangle struct{
    Height float32
    Width float32
}

// Method that returns a result
func (rect Rectangle) Area() float32{
    return rect.Height * rect.Width
}


rect := Rectangle {Height: 25.5, Width: 12.75}
fmt.Println(rect.Area())
```

Interface

- Interfaces are named collections of method signatures.

```
type Human interface {
    Display()
}

type Employee struct {
    name string; designation string
}

func (emp Employee) Display(){
    fmt.Println("Name - ", emp.name, ", Designation - ", emp.designation)
}

type Contractor struct {
    name string; weeklyHours int
}

func (cont Contractor) Display(){
    fmt.Println("Name - ", cont.name, ", Weekly Hours - ", cont.weeklyHours)
}

func main(){
    var emp Human = Employee{name:"Rob Pike", designation:"Engineer"}
    emp.Display()
    var cont Human = Contractor{name:"XYZ", weeklyHours:35}
    cont.Display()
}
```

Pointer

- Go supports pointers, allowing you to pass references to values and records within your program
- A pointer holds the memory address of a value. The type *T is a pointer to a T value
- The & operator generates a pointer to its operand.
- The * operator denotes the pointer's underlying value.

```
package main
import "fmt"
func main() {
        var ptr *int             // p is declared as a pointer to type int
        var no int = 100

        ptr = &no

        fmt.Println(ptr)         // prints memory address ptr points to
        fmt.Println(*ptr)        // prints the value ptr point to i.e. 100

        *ptr = *ptr / 2
        fmt.Println(no)          // no is 50 now
}
```

- **Unlike C, Go has no pointer arithmetic.**

Using pointer to pass value by ref

- Pointers are used to pass value to functions by ref, so that changes made in called function are reflected in callee/ parent function.

```
func main(){
    var message string = "Hello World"
    fmt.Println("Before function call - " + message)
    displayMessagePointer(&message)
    fmt.Println("After function call - " + message)
}

func displayMessagePointer(message *string){
    fmt.Println("Before update - " + *message)
    *message = "Hello World from Go"
    fmt.Println("After update - " + *message)
}
```

Output

```
Before function call - Hello World
Before update - Hello World
After update - Hello World from Go
After function call - Hello World from Go
```

```
func main(){
    var message = "Hello World"
    fmt.Println("Before function call - " + message)
    displayMessage(message)
    fmt.Println("After function call - " + message)
}

func displayMessage(message string){
    fmt.Println("Before update - " + message)
    message = "Hello World from Go"
    fmt.Println("After update - " + message)
}
```

Output

```
Before function call - Hello World
Before update - Hello World
After update - Hello World from Go
After function call - Hello World
```

Type Composition/Embedding, Packages and Export

Type Composition/ Embedding - struct

- Go does not have the typical, type-driven notion of subclassing, but it does have the ability to “borrow” pieces of an implementation by *embedding* types within a struct or interface.
- structs can have fields borrowed from another struct (parent)

```go
package main
import "fmt"

type Human struct {
    Name    string
    Address string
    Age     int
}
type Employee struct {
    Human       // Embedded anonymous field that points to another struct
    EmployeeNo  string
    Salary      float32
    Designation string
}

func main() {
    human := Human{"Rob Pike", "USA", 45}
    fmt.Println(human) // {Rob Pike USA 45}

    emp := Employee{human, "3423434", 200456.78, "Chief Architect"}
    fmt.Println(emp) // {{Rob Pike USA 45} 3423434 200456.78 Chief Architect}
    fmt.Printf("Name %s, Age %d \n", emp.Name, emp.Age)
}
```

Type Composition/ Embedding – struct (Cont’d)

```go
package main
import "fmt"

type Human struct {
    Name    string
    Address string
    Age     int
}

type Employee struct {
    Human       // Embedded anonymous field that points to another struct
    EmployeeNo  string
    Salary      float32
    Designation string
}

func (human Human) Display() {
    fmt.Printf("Name %s, Age %d \n", human.Name, human.Age)
}

func main() {
    human := Human{"Rob Pike", "USA", 45}
    fmt.Println(human)         // {Rob Pike USA 45}

    emp := Employee{human, "3423434", 200456.78, "Chief Architect"}
    fmt.Println(emp)           // {{Rob Pike USA 45} 3423434 200456.78 Chief Architect}
    fmt.Printf("Name %s, Age %d \n", emp.Name, emp.Age)

    emp.Display()              // Prints Name Rob Pike, Age 45
}
```

Type Composition/ Embedding – struct (Cont’d)

```go
package main
import "fmt"

type Human struct {
    Name    string
    Address string
    Age     int
}
type Employee struct {
    Human       // Embedded anonymous field that points to another struct
    EmployeeNo  string
    Salary      float32
    Designation string
}

func (human Human) Display() {
    fmt.Printf("Name %s, Age %d \n", human.Name, human.Age)
}

func (emp Employee) Display() {
    fmt.Printf("Name %s, Age %d Emp No %s Designation %s \n", emp.Name, emp.Age, emp.EmployeeNo, emp.Designation)
}

func main() {
    human := Human{"Rob Pike", "USA", 45}
    fmt.Println(human)         // {Rob Pike USA 45}

    emp := Employee{human, "3423434", 200456.78, "Chief Architect"}
    fmt.Println(emp)           // {{Rob Pike USA 45} 3423434 200456.78 Chief Architect}
    fmt.Printf("Name %s, Age %d \n", emp.Name, emp.Age)

    emp.Display()              // prints Name Rob Pike, Age 45 Emp No 3423434 Designation Chief Architect
}
```

Embed by value vs by ref

- Embedding could be by values or by ref

```
type Human struct {
    Name string
    Age int
}
```

```
type Employee struct {
    Human       // Embed by value
    EmployeeNo string
    Designation string
}
```

```
type Employee struct {
    *Human      // Embed by ref
    EmployeeNo string
    Designation string
}
```

```
human := Human {"Rob Pike", 45}
fmt.Println(human)           // {Rob Pike 45}

emp := Employee{human, "3423434", "Chief Architect" }
fmt.Println(emp)             // {{Rob Pike 45} 3423434 Chief Architect}
```

Type Composition/Embed - interface

- Embedding could be done for interface also.

```
type CanWalk interface {
    Walk()
}
type CanFly interface {
    Fly()
}
type CanWalkAndFly interface
{
    CanWalk      // Embed
    CanFly       // Embed
}

type Human struct {
    Name string
}
func (human Human) Walk() {
    fmt.Println("Human walk")
}
func (human Human) Fly() {
    fmt.Println("Human fly")
}
```

```
var h1 CanWalk = Human{"Aniruddha"}
h1.Walk()

var h2 CanFly = Human{"Aniruddha"}
h2.Fly()

var h3 CanWalkAndFly = Human{"Aniruddha"}
h3.Walk()
h3.Fly()
```

Exporting from package

- Methods whose name start with Upper case are exported. Methods whose name do not start with Upper case are not exported (private).

```
package library

import "fmt"

// Exported methods
func SayHello() {
        fmt.Println("Hello from Library")
}

func SayHello2() {
        fmt.Println("Hello2 from Library")
}

// Non exported method
func sayHelloPvt() {
        fmt.Println("sayHelloPvt from Library")
}
```

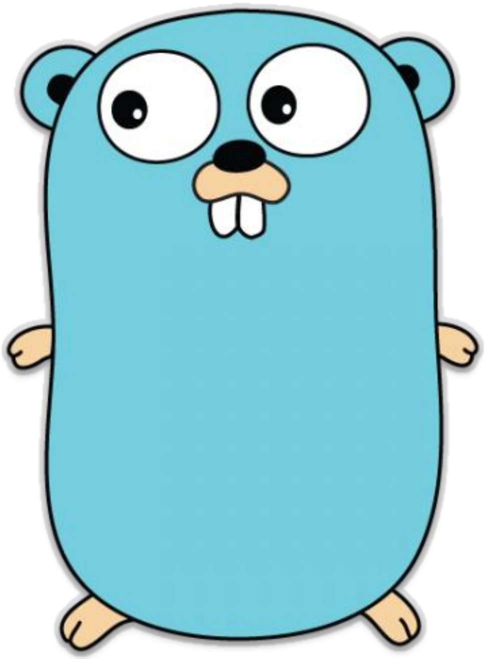
Custom Package

- Custom packages are searched in $GOPATH location.
- Vendoring process

```
package main

import (
        "employees"                   // Package
        "employees/salary"            // Sub package
        "fmt"
)

func main() {
        fmt.Println("Hello from Go")
        var empName string = employees.GetEmployee()
        fmt.Println(empName)

        var salary int = salary.GetSalary()
        fmt.Println(salary)
}
```

Folder Structure

```
bin
pkg
src
-- <Project base location>
   main.go
   -- vendor
      -- employees
          employees.main
          -- salary
             salary.main
```



Unique Go Features

Returning multiple values from function

- Supports multiple return value from functions like Tuples supported in other languages
- Could be used to return a pair of values / return value and error code

```
func swap(x, y int) (int, int){
    return y, x
}

var x,y int = 10,20
var p,q int = swap(x,y)
fmt.Println(x,y)             // 10 20
fmt.Println(p,q)             // 20 10

func addSubMultiDiv(x,y int) (int,  int, int, int){
    return x+y, x-y, x*y, x/y
}

fmt.Println(addSubMultiDiv(20,10))   // 30 10 200 2
```

Returning multiple values from function

- More than two values could be returned

```
func getEmployee() (string, int, float32) {
    return "Bill", 50, 6789.50
}
func main() {
    name, age, salary := getEmployee()
    fmt.Println(name)
    fmt.Println(age)
    fmt.Println(salary)
    fmt.Scanln()
}
```

- Return values not required could be ignored by using _

```
func main() {
    name, _, salary := getEmployee()
    fmt.Println(name)
    fmt.Println(salary)
    fmt.Scanln()
}
```

Named return values

```
func concat(str1, str2 string) (res string){
    res = str1 + " " + str2
    return
}

result := concat("Aniruddha","Chakrabarti")
fmt.Println(result)          // Aniruddha Chakrabarti

func getEmployee() (name string, age int, salary float32)
{
    name = "Bill"
    age = 50
    salary = 6789.50
    return
}

func main() {
    name, age, salary := getEmployee()
    fmt.Println(name)
    fmt.Println(age)
    fmt.Println(salary)
    fmt.Scanln()
}
```

Defer

- A defer statement defers the execution of a function until the surrounding function returns.
- The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.
- Defer is commonly used to simplify functions that perform various clean-up actions.
- Defer statements could be stacked up and would be executed in First In Last Out order i.e. the first Defer statement would be executed the last, and last Defer statement would be executed the first.

```go
package main

import "fmt"

func main() {
    defer fmt.Print("World ")
    fmt.Print("Hello ")
}

// prints Hello World
```

```go
package main

import "fmt"

func main() {
    defer fmt.Print("Happy learning Go! ")
    defer fmt.Print("from Golang! ")
    defer fmt.Print("World ")

    fmt.Print("Hello ")
}

// prints Hello World from Golang! Happy learning Go!
```

Defer

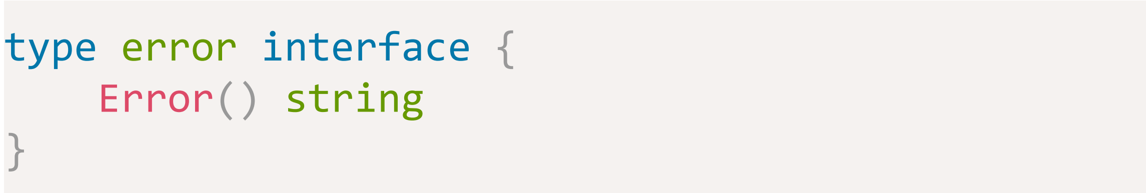
- Defer is commonly used to simplify functions that perform various clean-up actions.

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

Error Handing in Go

- Errors in Go are plain old values. Errors are represented using the built-in error type.
- Just like any other built-in type such as int, float64, ... error values can be stored in variables, passed as parameters to functions, returned from functions, and so on.

```
func Open(name string) (file *File, err error)
```



```
type error interface {
    Error() string
}
```

- If the file has been opened successfully, then the Open function will return the file handler and error will be nil. If there is an error while opening the file, a non-nil error will be returned.
- If a function or method returns an error, then by convention it has to be the last value returned from the function. Hence the Open function returns err as the last value

```
package main

import (
    "fmt"
    "os"
)

func main() {
    file, err := os.Open("/test.txt")
    if err != nil {
        fmt.Println(err.Error())
    } else {
        fmt.Println(file.Name())
    }
}
```

Error Handing using error object

- Update the function so that it returns two values: a string and an error. The caller function will check the second value to see if an error occurred.
- Import the Go standard library errors package so you can use its errors.New function.
- Add an if statement to check for an invalid request (an empty string where the name should be) and return an error if the request is invalid.
- Add nil (meaning no error) as a second value in the successful return. That way, the caller can see that the function succeeded.

```go
func main() {
    res1, err := divide(10, 0)
    if err != nil {
        log.Println(err)
    } else {
        fmt.Printf("result is %d \n", res1)
    }

    res2, err := divide(10, 2)
    if err != nil {
        log.Println(err)
    } else {
        fmt.Printf("result is %d \n", res2)
    }
}
```

```go
package main
import (
    "errors"
    "fmt"
    "log"
)

func divide(x int, y int) (int, error) {
    if y == 0 {
        return 0, errors.New("y can not be zero, as a no
 could not be divided by zero")
    }

    return x / y, nil
}
```

Panic

- There are certain operations in Go that automatically return panics and stop the program. Common operations include
  - Indexing an array beyond its capacity
  - Performing type assertions
  - Calling methods on nil pointers
  - Incorrectly using mutexes
  - Attempting to work with closed channels.
- Most of these situations result from mistakes made while programming that the compiler has no ability to detect while compiling your program.

Raising Panic

Out of the box panics

```
package main

import "fmt"

func main() {

    cities := []string{"Kolkata", "Delhi", "Chennai"
, "Bangalore"}

    fmt.Println(cities[4])

}
```

```
panic: runtime error: index out of range [4] with length 4

goroutine 1 [running]:
main.main()

C:/Users/Aniruddha.Chakrabart/go/src/Training/Panic/main.go:7
+0x1b
exit status 2
```

Raising custom panics

```
package main

import "fmt"

func main() {
    fmt.Println("Hello")
    div(10, 0)
}

func div(x int, y int) {
    if y == 0 {
        panic("Error: no can not be divided by zero")
    }
}
```

```
Hello
panic: Error: no can not be divided by zero

goroutine 1 [running]:
main.div(...)

C:/Users/Aniruddha.Chakrabart/go/src/Training/Panic/main.go:12
main.main()

C:/Users/Aniruddha.Chakrabart/go/src/Training/Panic/main.go:7 +0xa5
exit status 2
```

Panic and Recover

- Panics have a single recovery mechanism - the **recover** builtin function.
- This function allows you to intercept a panic on its way up through the call stack and prevent it from unexpectedly terminating your program.
- It has strict rules for its use, but can be invaluable in a production application.

```go
package main
import (
    "fmt"
    "log"
)

func main() {
    defer recoverFromPanic()
    div(10, 0)
    fmt.Println("we survived dividing by zero!")
}

func div(x int, y int) int {
    if y == 0 {
        panic("Error: no can not be divided by zero")
    }
    return x / y
}

func recoverFromPanic() {
    if err := recover(); err != nil {
        log.Println("panic occurred:", err)
    }
}
```

Concurrency in Go

Concurrency in Go

- Goroutines
- Channels

- Synchronization - sync package : To avoid race condition
- WaitGroup
- Once

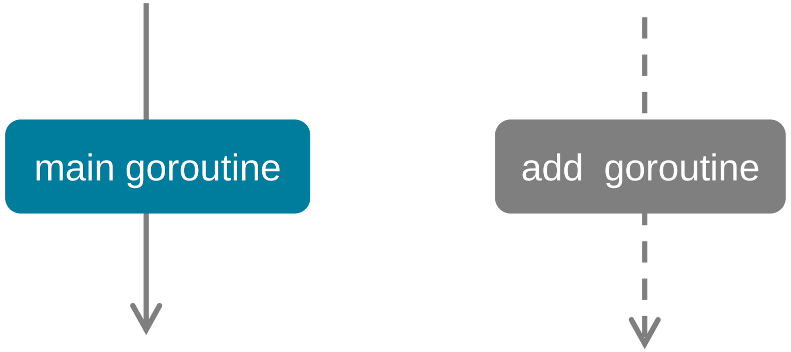
Apart from WaitGroup and Once the below are typically used by low-level library routines. **Higher-level synchronization is better done via channels and communication**

- Mutex
- RWMutex
- Pool

Goroutines

- Goroutines are lightweight threads managed by Go runtime – since they are lightweight creating them is fast, and does not impact performance
- A goroutine is a function that is capable of running concurrently with other functions.
- To create a goroutine, use the keyword **go** followed by a function invocation

```
func main(){
    add(20, 10)
    // add function is called as goroutine - will execute concurrently with calling one
    go add(20, 10)
    fmt.Scanln()
}

func add(x int, y int) {
    fmt.Println(x+y)
}
```



- Goroutines do not have names; they are just anonymous workers. They expose no unique identifier, name, or data structure to the programmer.
- Go statement does not return some item that can be used to access and control the goroutine later like Threads do. This is done on purpose.

Goroutines

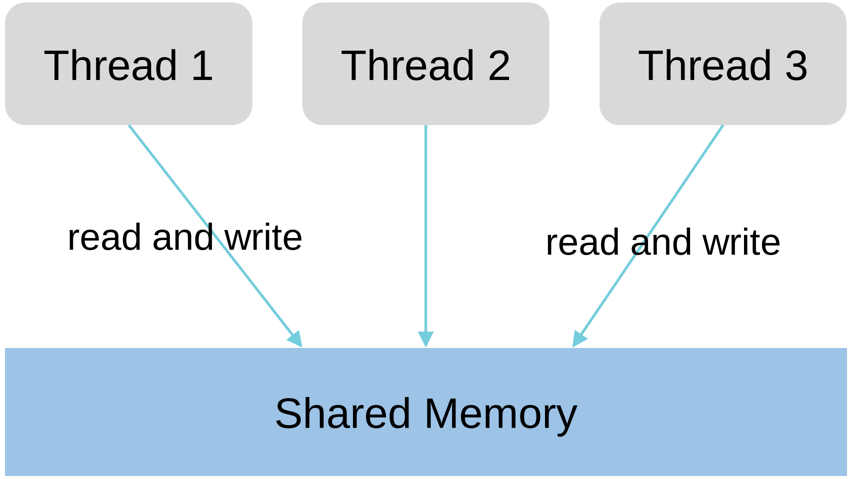
- Goroutine could be started for anonymous function call also

```
// Anonymous function that does not take parameter - called as Goroutine
go func(){
    fmt.Println("Another Goroutine")
}()            // last pair of parenthesis is the actual function invocation


// Anonymous function that takes parameters - called as Goroutine
go func(msg string){
    fmt.Println("Hello " + msg)
}("World")            // last pair of parenthesis is the actual function invocation
```

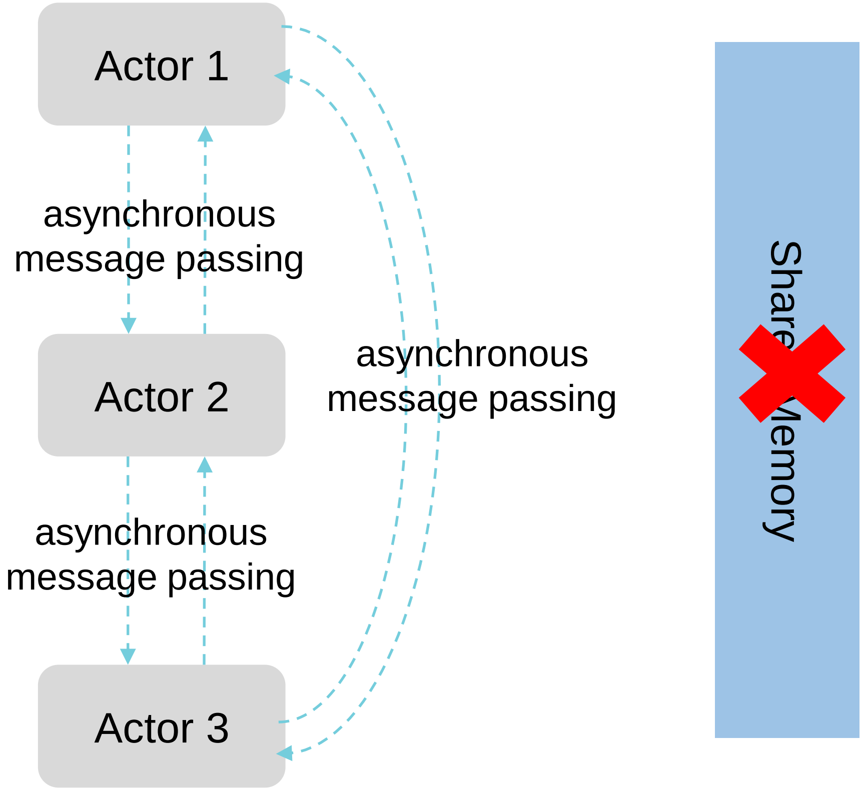
Concurrency - Actor model

Principle - Don't communicate by sharing memory; share memory by communicating

- Actors pass asynchronous messages among each other for communication and does not use shared memory to communicate.



Thread based concurrency model with shared memory

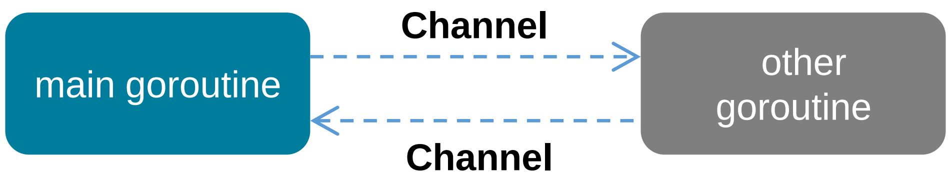

Actor 1
asynchronous message passing
Actor 2
asynchronous message passing
asynchronous message passing
Actor 3
Shared Memory

Actor based concurrency model – no shared memory

Channel

- Channels are the pipes that connect concurrent goroutines - Channels are **typed** by the values they convey
- You can send values into channels from one goroutine and receive those values into another goroutine.
- Channels are created using make operator

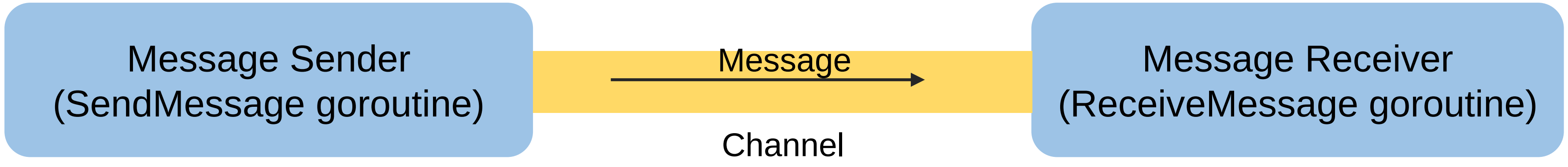
```
ch := make(chan int)
```



- Values are received and sent using the channel operator, **<-**

```go
package main

import "fmt"

func main() {
    ch := make(chan string)
    go func() { ch <- "ping" }()

    msg := <-ch
    fmt.Println(msg)
}
```

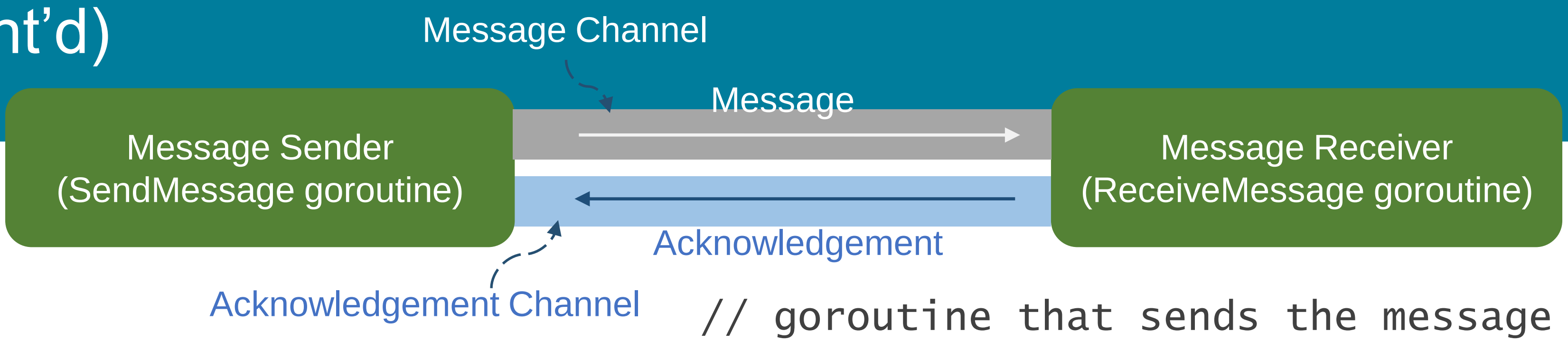
Comminating b/w Goroutines using Channel



```
package main

import (
    "fmt"
    "time“
)

func main(){
    channel := make(chan string)

    go SendMessage(channel)
    go ReceiveMessage(channel)

    fmt.Scanln();
}
```

```
// goroutine that sends the message
func SendMessage(channel chan string) {
    for {
        channel <- "sending message @" + time.Now().String()
        time.Sleep(5 * time.Second)
    }
}


// goroutine that receives the message
func ReceiveMessage(channel chan string) {
    for {
        message := <- channel
        fmt.Println(message)
    }
}
```

Channels (Cont'd)



Message Channel
Message
Message Sender (SendMessage goroutine)
Message Receiver (ReceiveMessage goroutine)
Acknowledgement
Acknowledgement Channel

```go
package main

import (
    "fmt"
    "time"
)

func main(){
    // Channel to send message from
sender to receiver
    msgChnl := make(chan string)

    // Channel to acknowledge message
receipt by receiver
    ackChnl := make(chan string)

    go SendMessage(msgChnl, ackChnl)
    go ReceiveMessage(msgChnl, ackChnl)

    fmt.Scanln();
}
```

```go
// goroutine that sends the message
func SendMessage(msgChannel chan string,
ackChannel chan string) {
    for {
        msgChannel <- "sending message @" +
time.Now().String()
        time.Sleep(2 * time.Second)
        ack := <- ackChannel
        fmt.Println(ack)
    }
}

// goroutine that receives the message
func ReceiveMessage(msgChannel chan string,
ackChannel chan string) {
    for {
        message := <- msgChannel
        fmt.Println(message)
        ackChannel <- "message received @" +
time.Now().String()
    }
}
```

WaitGroup

- To wait for multiple goroutines to finish, we can use a wait group.
- **WaitGroup.Add()** is called to set the number of goroutines we want to wait for, and subsequently, **WaitGroup. Done()** is called within any goroutine to signal the end of its' execution
- **WaitGroup.Wait()** is called to block the execution of main() function until the goroutines in the WaitGroup is successfully completed

```go
package main

import "fmt"

func main() {
    go displayMessage("Hello")
    go displayMessage(“World")
}

func displayMessage(msg string) {
    fmt.Println(msg + " ")
    // execution does not reach this line
}

// Does not display any message
// You can add a Scan statement or add a sleep,
// but that is not the right solution
```

```go
package main
import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go displayMessage("Hello", &wg)

    wg.Add(1)
    go displayMessage("World", &wg)
    wg.Wait()
}

func displayMessage(msg string, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println(msg + " ")
}
```

Once

- Helps you to run your code only once – behaves like Singleton pattern.

```
package main
import (
        "fmt"
        "sync"
)

func main() {
        var once sync.Once
        for i := 0; i < 5; i++ {
                once.Do(welcomeUser)   // Gets invoked only once and not 5 times
        }
}

func welcomeUser() {
        fmt.Println("Hello")
}
```

Mutex

- A Mutex, or a mutual exclusion is a mechanism that allows us to prevent concurrent processes from entering a critical section of data whilst it’s already being executed by a given process.
- Mutex in Go is implemented as sync.Mutex
- sync.Mutext has only two methods –
  - Lock()
  - Unlock()

```go
package main
import "fmt"

// Account Type
type Account struct {
	balance int
	name    string
}

func (acnt *Account) Deposit(amount int) {
	acnt.balance += amount
}

func (acnt *Account) Withdraw(amount int) {
	acnt.balance -= amount
}

func WithoutSync() {
	var acnt = Account{name: "XYZ", balance: 1000}
	for i := 0; i < 100; i++ {
		go acnt.Deposit(100)
		go acnt.Withdraw(100)
	}

	fmt.Printf("Account Balance: %d \n", acnt.balance)
}
```

Mutex

```
package main

import (
    "fmt"
    "sync"
)

func WithSync() {
    var mutex *sync.Mutex
    var wg sync.WaitGroup

    var acnt = AccountSafe{name: "XYZ", balance: 1000,
mutex: mutex}
    wg.Add(200)
    for i := 0; i < 100; i++ {
        go acnt.Deposit(100, &wg)
        go acnt.Withdraw(100, &wg)
    }
    wg.Wait()
    fmt.Printf("Account Balance: %d \n", acnt.balance)
}
```

```
// Safe Account Type that uses locks
type AccountSafe struct {
    balance int
    name    string
    mutex   *sync.Mutex
}

func (acnt *AccountSafe) Deposit(amount int,
wg *sync.WaitGroup) {
    acnt.mutex.Lock()
    acnt.balance += amount
    acnt.mutex.Unlock()
    wg.Done()
}

func (acnt *AccountSafe) Withdraw(amount int,
 wg *sync.WaitGroup) {
    acnt.mutex.Lock()
    acnt.balance -= amount
    acnt.mutex.Unlock()
    wg.Done()
}
```

RWMutex

- RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer.

```go
package main

import (
    "fmt"
    "sync"
)

func WithRWMutex() {
    var rwmux *sync.RWMutex
    var wg sync.WaitGroup

    var acnt = AccountRWSafe{name: "XYZ", balance: 1000, rwmux: rwmux}
    wg.Add(200)
    for i := 0; i < 100; i++ {
        go acnt.Deposit(100, &wg)
        go acnt.Withdraw(100, &wg)
    }
    wg.Wait()

    wg.Add(1)
    fmt.Printf("Final Account Balance: %d \n", acnt.GetBalance(&wg))
    wg.Wait()
}
```

```go
// Safe Account Type that uses locks
type AccountSafe struct {
    balance int
    name    string
    mutex   *sync.Mutex
}
func (acnt *AccountSafe) Deposit(amount int, wg *sync.
WaitGroup) {
    acnt.mutex.Lock()
    acnt.balance += amount
    acnt.mutex.Unlock()
    wg.Done()
}
func (acnt *AccountSafe) Withdraw(amount int, wg *sync
.WaitGroup) {
    acnt.mutex.Lock()
    acnt.balance -= amount
    acnt.mutex.Unlock()
    wg.Done()
}
func (acnt *AccountRWSafe) GetBalance(wg *sync.WaitGro
up) int {
    var bal int
    acnt.rwmux.RLock()
    bal = acnt.balance
    acnt.rwmux.RUnlock()
    wg.Done()
    return bal
}
```

net/http package

net/http package – Writing a simple web server

- net/http packages is used for building a simple web server that listens to a port and sends a text response

```go
package main

import (
    "fmt"
    "net/http"
)

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

func handler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "Hi there, I love Go")
}
```

net/http package – Improving the simple web server

- Error handling is added
- Error is logged
- Request path is retrieved from request objects and sent as output

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler)
    err := http.ListenAndServe(":8080", nil)
    log.Fatal(err)
}

func handler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "Hi there, I love %s!", request.URL.Path[1:])
}
```

Testing

Unit testing in Go

- testing package is used for unit testing in Go
- Unit test file should be named as <file to be tested>_test.go
- Test functions should accept a pointer to testing.T - T is a type passed to Test functions to manage test state and support formatted test logs
- Tests are run using go test

main.go

```go
package main

import "fmt"

func main() {
    fmt.Println(Divide(20, 4))
}

func Divide(x int, y int) int {
    return x / y
}
```

main_test.go

```go
package main

import (
    "testing"
)

func TestDivide(t *testing.T) {
    if Divide(20, 10) != 2 {
        t.Error("Expected result is 2")
    }
}
```

```
> go run main.go
5
> go test
PASS
ok      Training/Testing        1.609s
PS C:\Users\Aniruddha.Chakrabart\go\src\Training\Testing>
```

Unit testing in Go (Cont'd)

- To get verbose output use **–v** flag

```
> go test -v

=== RUN   TestDivideForPositiveNo
--- PASS: TestDivideForPositiveNo (0.00s)
=== RUN   TestDivideForNegativeNo
--- PASS: TestDivideForNegativeNo (0.00s)
```

- To check test coverage use **–cover** flag

```
> go test -cover

PASS
coverage: 50.0% of statements
ok      Training/Testing        3.993s
```

- Test coverage could be visualized using go tool `(go tool cover)`

Modules

Modules

Enable Modules if they are not already enabled	`go env -w GO111MODULE=on`

Initialize a Go module in a folder	`go mod init`
	`go mod githib.com/ani/<package_name>`

Download dependencies / additional modules to be used	`go get`
	`go get github.com/nats-io/nats.go/`

⇩ Code/ Develop the module functionality	
Format, code analysis	`go fmt / go vet / go lint`
Test the code	`go test`
Build the code, generate exe	`go build`
Remove unused dependencies	`go mod tidy -v`

Modules

Initialize a Go module in a folder

```
go mod init
go mod github.com/ani/<package_name>
```

Download dependencies / additional modules to be used

```
go get
go get github.com/nats-io/nats.go/
```

go init and go get generates two files

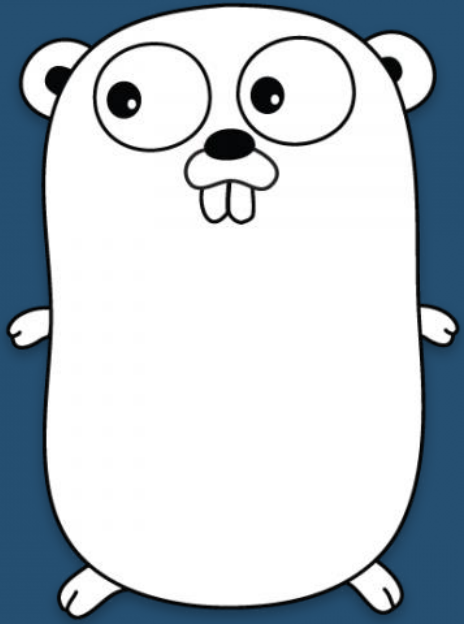
go.mod

```
module github.com/ani/<package_name>

go 1.16

require github.com/nats-
io/nats.go v1.11.0
```

go.sub

```
github.com/nats-io/nats.go v1.11.0 h1:L263PZkrmkRJRJT2YHU8GwWWvEvmr9/LUKuJTXsF32k=
github.com/nats-
io/nats.go v1.11.0/go.mod h1:BPko4oXsySz4aSWeFgOHLZs3G4Jq4ZAyE6/zMCxRT6w=
github.com/nats-io/nkeys v0.3.0 h1:cgM5tL53EvYRU+2YLXIK0G2mJtK12Ft9oeooSZMA2G8=
github.com/nats-
io/nkeys v0.3.0/go.mod h1:gvUNGjVcM2IPr5rCsRsC6Wb3Hr2CQAm08dsxtV6A5y4=
github.com/nats-io/nuid v1.0.1 h1:5iA8DT8V7q8WK2EScv2padNa/rTESc1KdnPw4TC2paw=
github.com/nats-
io/nuid v1.0.1/go.mod h1:19wcPz3Ph3q0Jbyiqsd0kePYG7A95tJPxeL+1OSON2c=
golang.org/x/crypto v0.0.0-20210314154223-
e6e6c4f2bb5b h1:wSOdpTq0/eI46Ez/LkDwIsAKA71YP2SRKBODiRWM0as=
golang.org/x/crypto v0.0.0-20210314154223-
e6e6c4f2bb5b/go.mod h1:T9bdIzuCu7OtxOm1hfPfRQxPLYneinmdGuTeoZ9dtd4=
golang.org/x/net v0.0.0-20210226172049-
e18ecbb05110/go.mod h1:m0MpNAwzfU5UDzcl9v0D8zg8gWTRqZa9RBIspLL5mdg=
golang.org/x/sys v0.0.0-20201119102817-
f84b799fce68/go.mod h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
golang.org/x/term v0.0.0-20201126162022-
7de9c90e9dd1/go.mod h1:bj7SfCRtBDWHUb9snDiAeCFNEtKQo2Wmx5Cou7ajbmo=
golang.org/x/text v0.3.3/go.mod h1:5Zoc/QRtKVWzQhOtBMvqHzDpF6irO9z98xDceosuGiQ=
golang.org/x/tools v0.0.0-20180917221912-
90fa682c2a6e/go.mod h1:n7NCudcB/nEzxVGmLbDWY5pfWTLqBcC2KZ6jyYvM4mQ=
```

Go – Key Takeaway



- Support for a limited but well thought out set of language constructs *(Less is More)*
- Package
- Support for struct and interface only, does not support classes.
- Concurrency through goroutines and channels
- Defer & Panic
- Type composition through embedding
- Multiple return values from functions
- Access level of functions depending on casing
- Compiles to native machine code, but has garbage collection and automatic memory management.
- Go Tools
- Modules for Dependency management, versioning, new way of development / build

Resources

- Go Lang website - https://golang.org
- Go Dev website - go.dev

- Go By Example - https://gobyexample.com/
- An Introduction to Programming in Go - https://www.golang-book.com/books/intro
- Little Go Book - http://openmymind.net/assets/go/go.pdf
- Effective Go - https://golang.org/doc/effective_go.html

- Less is exponentially more: Rob Pike - http://commandcenter.blogspot.in/2012/06/less-is-exponentially-more.html
- Communicating Sequential Processes (CSP) by Tony Hoare https://en.wikipedia.org/wiki/Communicating_sequential_processes
- Go Cheatsheet - Go cheatsheet (devhints.io)