



Golang

Parameswari Ettiappan

A black and white photograph of a woman with curly hair, wearing a light-colored shirt, sitting at a desk and working on a laptop. A large orange diagonal bar starts from the top right and extends towards the center of the image.

High performance. Delivered.

consulting | technology | outsourcing



Goals

- Capture core Go language Concepts lab setup, Data Types and Kick Start Coding
- Understanding Functions, Recursion Error Handling and Recover
- Capturing Methods and Interfaces to design modules More on Bit Vector, Port interface and Http Handling
- Designing concurrent applications like Chat Server
- Go Tool Introduction

Software Requirements



-
- Ubuntu or Windows 10 or Mac
 - Golang zip/tar
 - Goland / Nano / Sublime/Notepad++
 - MySQL 5 or above



Golang

Golang

Python

Golang

Statically Typed

Powerful Library

Fast & Powerful

Robert

Rob

Ken



Introduction

- Go is a programming language that was born out of frustration at Google.
- Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production.
- Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.



Introduction

- While Go is a versatile programming language that can be used for many different programming projects, it's particularly well suited for networking/distributed systems programs.
- It has earned a reputation as “the language of the cloud”.
- It focuses on helping the modern programmer do more with a strong set of tooling and making deployment easy by compiling to a single binary.
- Go is easy to learn, with a very small set of keywords, which makes it a great choice for beginners and experienced developers alike.



Why Go

- Go is an open-source but backed up by a large corporation
- Automatic memory management (garbage collection)
- Strong focus on support for concurrency
- Fast compilation and execution
- Statically typed, but feels like dynamically typed
- Good cross-compiling (cross-platform) support
- Go compiles to native machine code
- Rapid development and growing community (Docker/Kubernetes)

Does Golang have a future?



-
- It has a very bright future. In the 6 short years since its birth, Go has skyrocketed to the Top 20 of all language ranking indices:
 - Go is an absolutely minimalist language with only a handful of programming concepts
 - Go has superb built-in support for concurrency



Is Golang better than Python and C++?

- For the readability of code, Golang definitely has the upper hand in most cases and trumps Python as a programming language.
- Go is much easier to learn and code in than C++ because it is simpler and more compact. Go is significantly faster to compile over C++.



Golang vs Python



Performance

Scalability

Applications

Execution

Libraries

Readability





Golang vs Python

Performance



Mandelbrot

⌚ 279.68 SEC

eraser 49344 units

N-Body

⌚ 882.00 SEC

eraser 8212 units

Fasta

⌚ 62.88 SEC

eraser 680736 units

Python

Mandelbrot

⌚ 5.47 SEC

eraser 31280 units

N-Body

⌚ 21.00 SEC

eraser 1532 units

Fasta

⌚ 2.07 SEC

eraser 3168 units

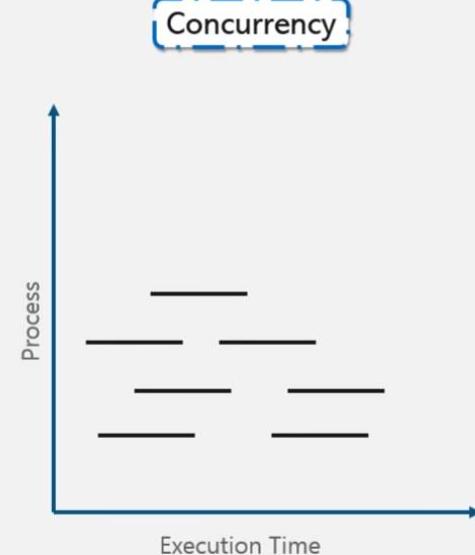
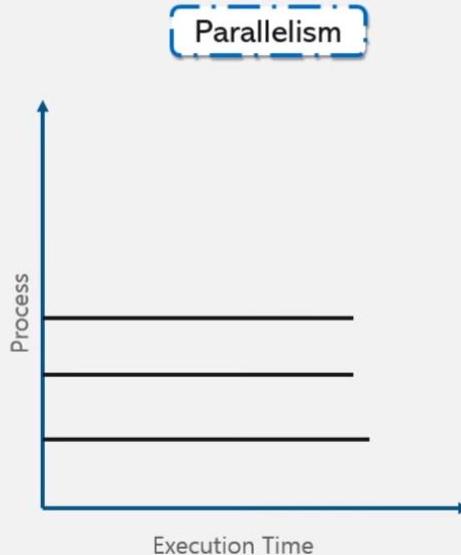
Golang

SUBS



Golang vs Python

Scalability





Golang vs Python

Applications



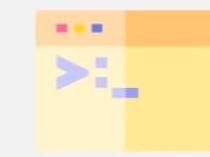
Data Analytics



Web Development



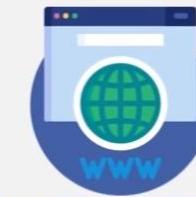
Artificial Intelligence



System Programming



Cloud Computing



Web Development





Golang vs Python

Execution



Dynamically Typed



Interpreter



Statically Typed



Compiler

Golang



Golang vs Python

Library



Scipy



Numpy



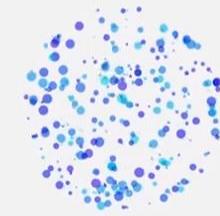
Scikit Learn

pandas
 $y_{it} = \beta x_{it} + \mu_i + \epsilon_{it}$

Pandas



Crypto.go



runtime.go



sql.go

Golang



Golang vs Python

Readability

```
print("hello world")
```

- Excellent Readability
- There are multiple ways to write the same thing, which may lead to confusion when someone else reads your code



Python

```
package main  
  
import "fmt"  
  
func main () {  
  
    fmt.Println("Hello World")  
  
}
```

- Excellent Readability
- Go is strict about how you write your code which means once you write something it is understood by everyone



Golang



Golang vs Java

Java

- Object-oriented language
- Virtual machine
- Bigger community
- Slower

Go

- Supports concurrency
- No error handling
- Easier to read
- Faster

Java & Go

- Server-side programs
- C language family
- Uses garbage collector



Golang vs Java

#1. Architecture

GO



Go does not provide any VM such as Java JVM. This language only compiles to metal like c++/c.

Java



It combines both interpretation and compilation approach. Bytecode is interpreted by Java Virtual Machine. Machine code generated by JVM and executed by the system in which Java program runs.



Golang vs Java

#2. Language

GO



It is an independent programming language and has at least two compilers such as `gccgo` and `go`.

Java



Java is an independent language.



Golang vs Java

#3. Expression Syntax

GO



The syntax on Go is specified by the use of extended Backus-Naur Form (EBNF).

Java



Syntax the same everywhere – independent of an IDE or a compiler.



Golang vs Java

#4. Mobile Support

GO



The Go mobile subrepository includes mobile support for mobile platforms like iOS and Android and offer tools for building mobile apps.

Java



Depends upon the device manufacturers.



Golang vs Java

#5. Routing

GO



Uses
http protocol for routing configuration.

Java



Uses
akka.routing.Consistent Hashing Router
and akka.routing.Scatter Gather First
Completed Router for routing
configuration



Golang vs Java

#6. Dependency Injection

GO



Uses
dependency injection.

Java



Uses
dependency injection and allows
modification.



Golang vs Java

#7. Structure

GO



Easily manageable.

Java



Better
structure, user-friendly, easier to create
and maintain large applications.



Golang vs Java

#8. Speed

GO



Relatively faster
than Java.

Java



Java is slower than
Go.



Golang vs Java

The Basis Of Comparison	GO	Java
Architecture	Go does not provide any VM such as Java JVM. This language only compiles to metal like c++/c.	It combines both the interpretation and compilation approach. Java Virtual Machine interprets bytecode. Machine code generated by JVM and executed by the system in which the Java program runs.
Language	It is an independent programming language and has at least two compilers, such as gccgo and go.	Java is an independent language.
Expression Syntax	The syntax on Go is specified by the use of the extended Backus-Naur Form (EBNF).	Syntax the same everywhere – independent of an IDE or a compiler



Golang vs Java

Comparison of Golang and Java Features		
Mobile Support	The Go mobile subrepository includes mobile support for mobile platforms like iOS and Android and offer tools for building mobile apps.	Depends upon the device manufacturers.
Routing	Uses HTTP protocol for routing configuration	Uses Akka.routing.ConsistentHashingRouter and Akka.routing.ScatterGatherFirstCompletedRouter for routing configuration
Dependency Injection	Uses dependency injection	It uses dependency injection and allows modification
Structure	Easily manageable	Better structure, user-friendly, easier to create and maintain large applications.
Speed	Relatively faster than Java	Java is slower than Go



Go improves over Java

- The Go library is smaller, making sifting through it easier.
- It does away with the colon at the ends of lines.
- It doesn't require the use of brackets and parentheses.
- Go has no error handling. (It is built for people who already know how to code.)



Speed vs platform dependency

- Go is faster than Java on almost every benchmark.
This is due to how it is compiled:
- Go doesn't rely on a virtual machine to compile its code.
- It gets compiled directly into a binary file.



Speed vs platform dependency

Go speed:

Copy

```
Output:  
Factorial    Time To calculate factorial  
10000        0.03 seconds  
50000        0.41 seconds  
100000       2.252 seconds  
500000       68.961 seconds  
1000000      224.135 seconds
```

Java speed:

Copy

```
Output:  
Factorial    Time To calculate factorial  
10000        0.112 seconds  
50000        1.185 seconds  
100000       2.252 seconds  
500000       89.500 seconds  
1000000      385.868 seconds
```

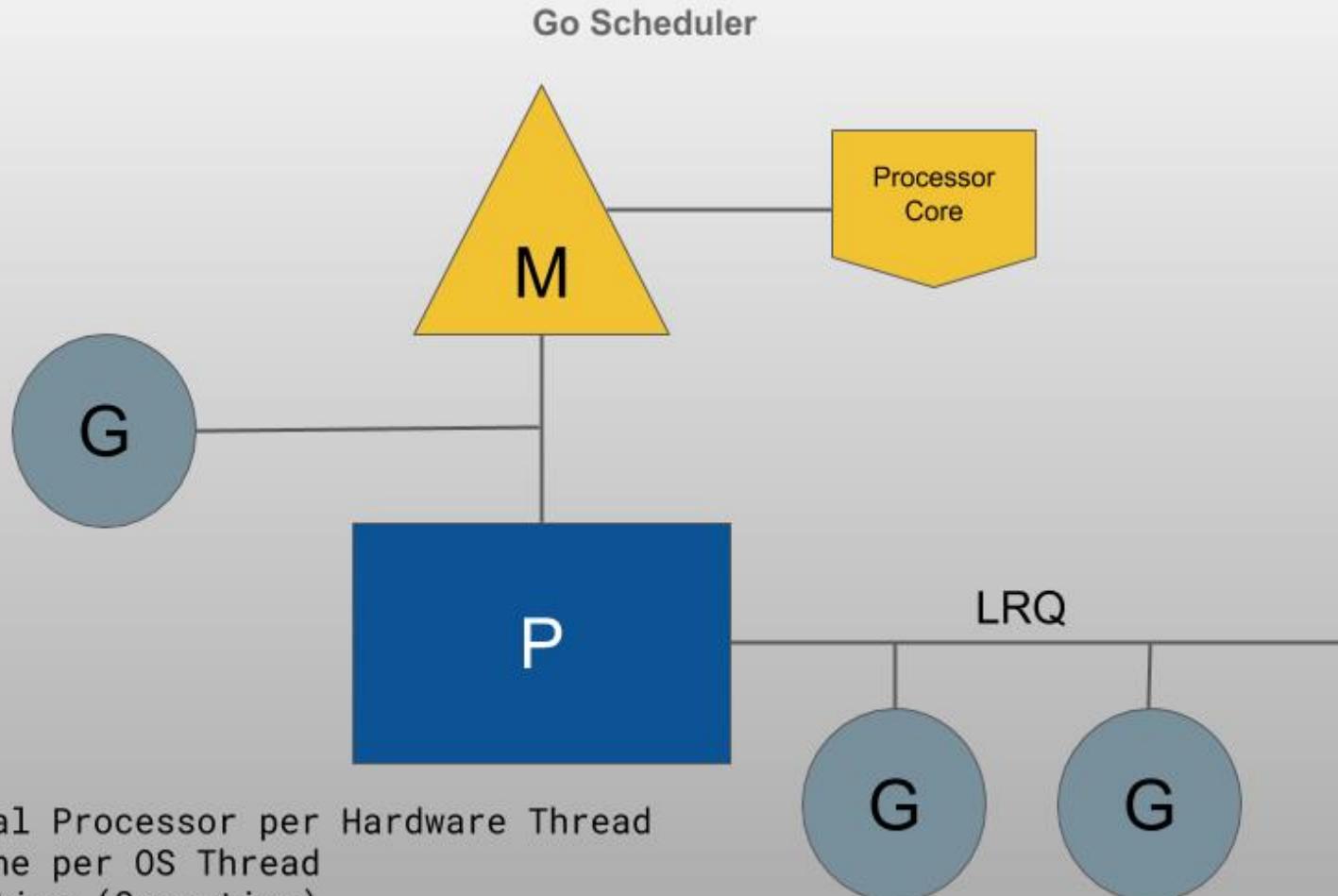


Go internal memory structure

- Each Go program process is allocated some virtual memory by the Operating System(OS), this is the total memory that the process has access to.
- The actual memory that is used within the virtual memory is called Resident Set. This space is managed by the internal memory constructs.



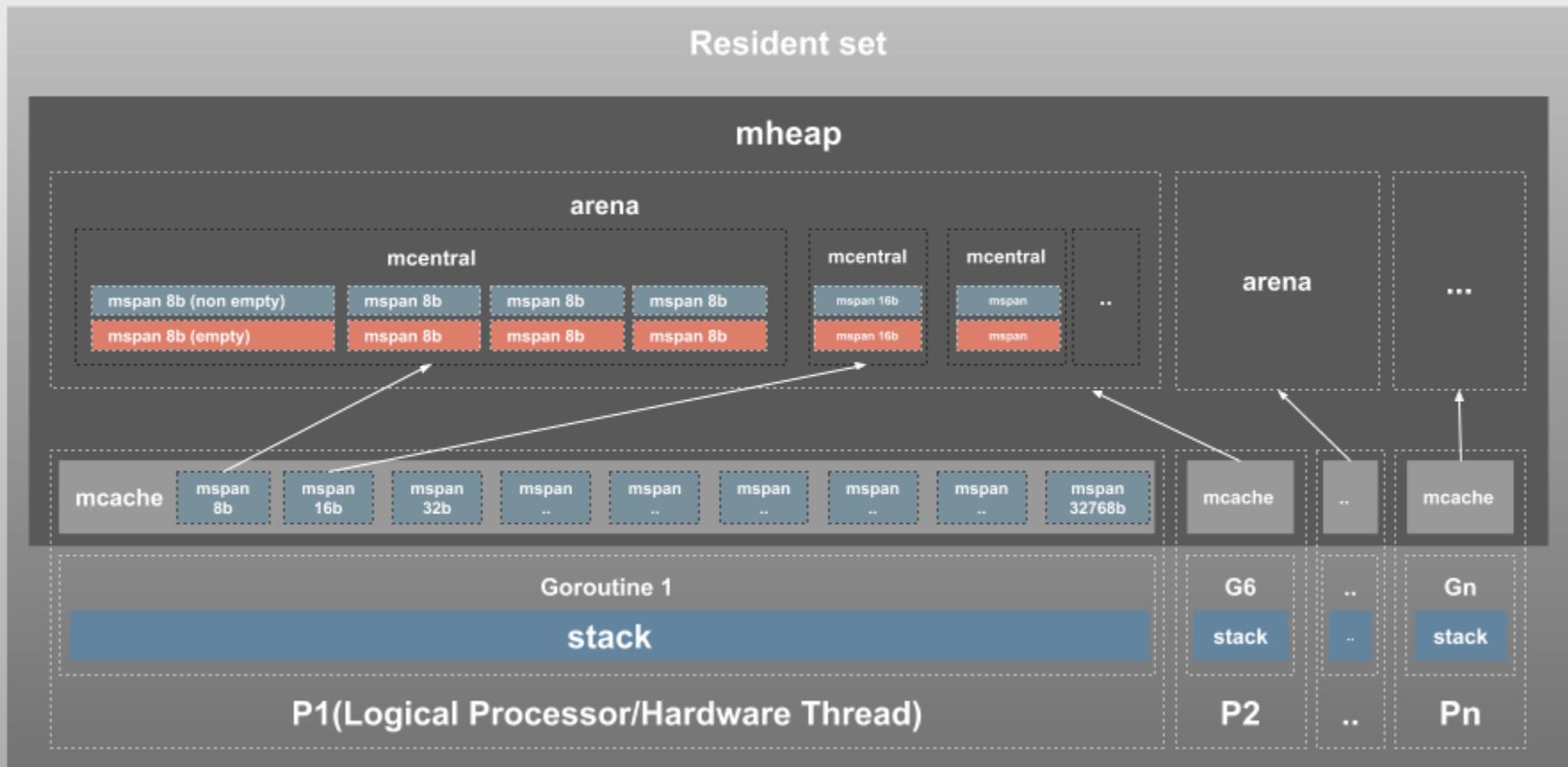
Go internal memory structure





Go internal memory structure

Process Virtual Memory





Page Heap(mheap)

- This is where Go stores dynamic data(any data for which size cannot be calculated at compile time).
- This is the biggest block of memory and this is where Garbage Collection(GC) takes place.
- The resident set is divided into pages of 8KB each and is managed by one global mheap object.
- Large objects(Object of Size > 32kb) are allocated directly from mheap.
- These large requests come at an expense of central lock, so only one P's request can be served at any given point in time.



Page Heap(mheap)

- mheap manages pages grouped into different constructs as below:
 - mspan: mspan is the most basic structure that manages the pages of memory in mheap.
 - It's a double-linked list that holds the address of the start page, span size class, and the number of pages in the span.
 - Like TCMalloc, Go also divides Memory Pages into a block of 67 different classes by size starting at 8 bytes up to 32 kilobytes as in the below image



Page Heap(mheap)





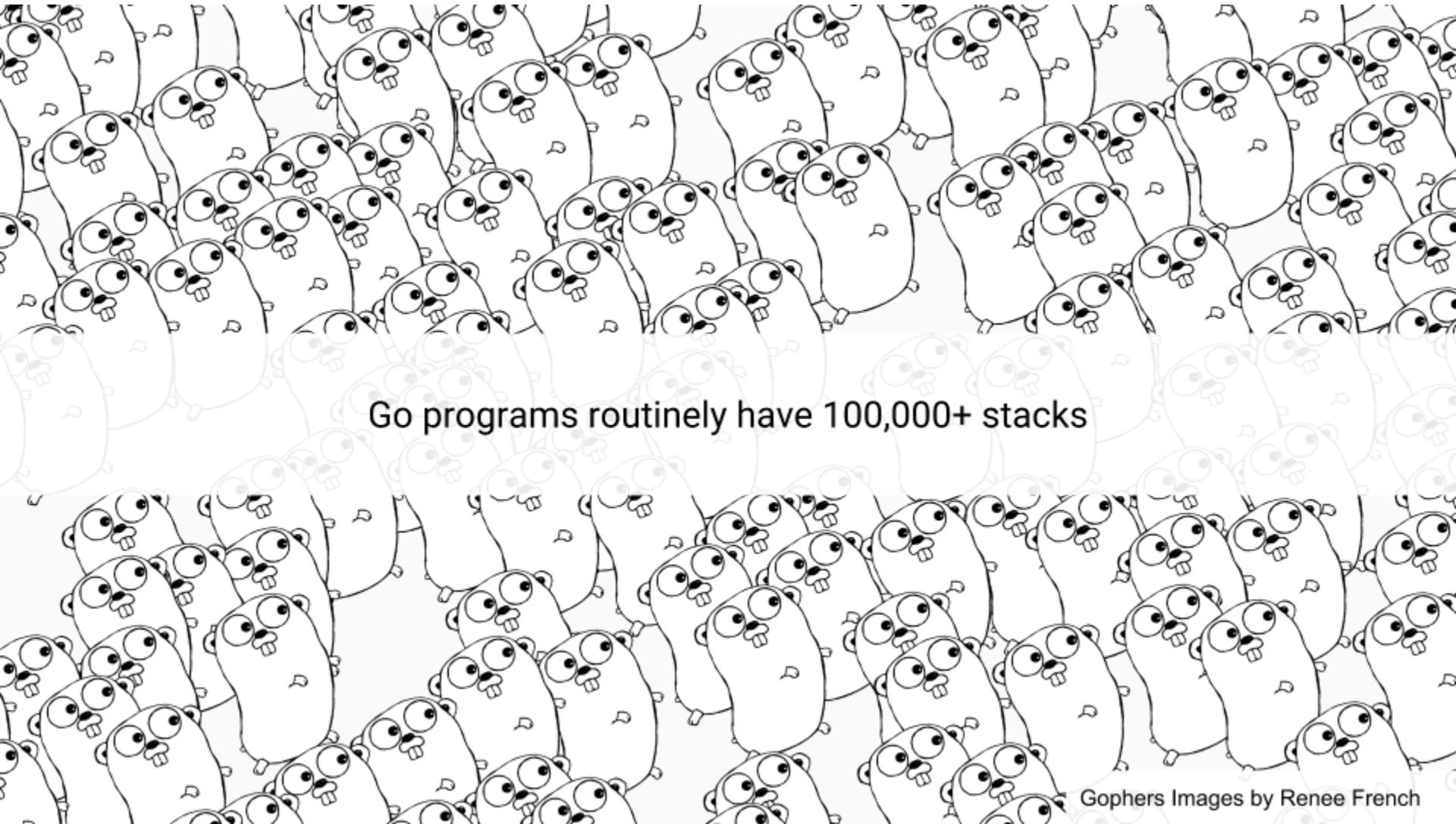
Page Heap(mheap)

- mcentral: mcentral groups spans of same size class together. Each mcentral contains two mspanList:
 - empty: Double linked list of spans with no free objects or spans that are cached in a mcache. When a span here is freed, it's moved to the nonempty list.
 - non-empty: Double linked list of spans with a free object. When a new span is requested from mcentral, it takes that from the nonempty list and moves it into the empty list.
- When mcentral doesn't have any free span, it requests a new run of pages from mheap.



Page Heap(mheap)

- arena: The heap memory grows and shrinks as required within the virtual memory allocated.
 - When more memory is needed, mheap pulls them from the virtual memory as a chunk of 64MB(for 64-bit architectures) called arena.
 - The pages are mapped to spans here.
- mcache: This is a very interesting construct. mcache is a cache of memory provided to a P(Logical Processor) to store small objects(Object size <=32Kb).
 - Though this resembles the thread stack, it is part of the heap and is used for dynamic data.
 - mcache contains scan and noscan types of mspan for all class sizes.
 - Goroutines can obtain memory from mcache without any locks as a P can have only one G at a time.
 - Hence this is more efficient. mcache requests new spans from mcentral when required.



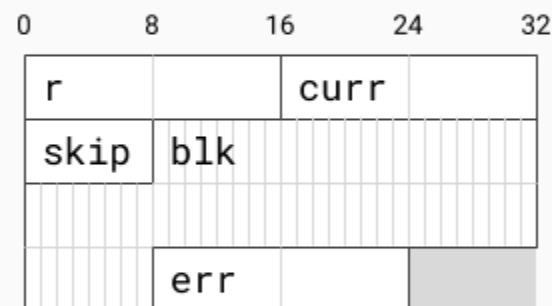
Go programs routinely have 100,000+ stacks

Gophers Images by Renee French



Go is value-oriented

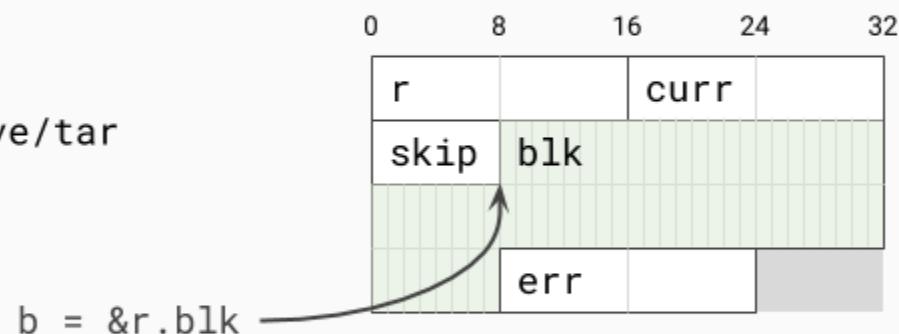
```
type Reader struct { // archive/tar
    r      io.Reader
    curr  numBytesReader
    skip   int64
    blk    block
    err    error
}
type block [64]byte
```





Go allows interior pointers

```
type Reader struct { // archive/tar
    r      io.Reader
    curr  numBytesReader
    skip   int64
    blk    block
    err    error
}
type block [64]byte
```





Static ahead of time compilation

Binary contains entire runtime

No JIT recompilation



Go memory usage (Stack vs Heap)

- Main function is kept in a “main frame” on the Stack
- Every function call is added to the stack memory as a frame-block
- All static variables including arguments and the return value is saved within the function frame-block on the Stack
- All static values regardless of type are stored directly on the Stack. This applies to global scope as well
- All dynamic types are created on the Heap and is referenced from the Stack using Stack pointers. Objects of size less than 32Kb go to the mcache of the P. This applies to global scope as well



Go memory usage (Stack vs Heap)

- The struct with static data is kept on the stack until any dynamic value is added at that point the struct is moved to the heap
- Functions called from the current function is pushed on top of the Stack
- When a function returns its frame is removed from the Stack
- Once the main process is complete, the objects on the Heap do not have any more pointers from Stack and becomes orphan



Garbage collection

- Go manages the heap memory by garbage collection. In simple terms, it frees the memory used by orphan objects, i.e, objects that are no longer referenced from the Stack directly or indirectly(via a reference in another object) to make space for new object creation.
- As of version 1.12, the Golang uses a non-generational concurrent tri-color mark and sweep collector.



Go Cross Compiler

- GOOS and GOARCH
- The environment variables GOOS and GOARCH configure the target platform Go will compile to.
- By default, the GOHOSTOS and GOHOSTARCH, which contain the auto-detected values of the host architecture will be used as the target platform to compile to.
- Type ‘go env’ to check what the target architecture is.



Go Cross Compiler

```
$ go env ... GOARCH="amd64" GOHOSTARCH="amd64"  
GOHOSTOS="darwin" GOOS="darwin" ...
```

```
root@cccd8f40b9eb7:/go# go env ...  
GOARCH="amd64" GOHOSTARCH="amd64"  
GOHOSTOS="linux" GOOS="linux" ...
```



Go Cross Compiler

Compiling for a platform

To compile for a specific platform, you have to set the `GOOS` and `GOARCH` environment variables. Below is a table that shows the available values. Go supports more OS and CPU architectures than the one shown below, but normally you will use these settings the most.

Platform	GOOS	GOARCH
Mac	darwin	amd64
Linux	linux	amd64
Windows	windows	amd64



Go Cross Compiler

Cross Compilation

Lets create a simple 'helloworld.go' example and cross compile it to Linux, Mac and Windows.

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("Hello World!")
}
```

COPY

To compile it type:

```
$ GOOS=darwin GOARCH=amd64 go build -o helloworld-mac helloworld.go
$ file helloworld-mac
helloworld: Mach-O 64-bit executable x86_64

$ GOOS=linux GOARCH=amd64 go build -o helloworld-linux helloworld.go
$ file helloworld-linux
helloworld-linux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, not stripped

$ GOOS=windows GOARCH=amd64 go build -o helloworld-windows.exe helloworld.go
$ file helloworld-windows.exe
helloworld-windows.exe: PE32+ executable (console) x86-64 (stripped to external PDB),
for MS Windows
```

COPY



Go Cross Compiler

Compile for Windows

Here's the command you need to run to compile your Go project for a 64-bit Windows machine:

```
$ GOOS=windows GOARCH=amd64 go build -o bin/app-amd64.exe app.go
```

In this scenario, `GOOS` is windows, and `GOARCH` is `amd64` indicating a 64-bit architecture. If you need to support a 32-bit architecture, all you need to do is change `GOARCH` to `386`.

```
$ GOOS=windows GOARCH=386 go build -o bin/app-386.exe app.go
```

Compile for macOS

The `GOARCH` values for Windows are also valid for macOS, but in this case the required `GOOS` value is `darwin`:



Go Cross Compiler

```
Terminal Local
operable program or batch file.

F:\ciscogolangws\day3\crosscompilation>go tool dist list
aix/ppc64
android/386
android/amd64
android/arm
android/arm64
darwin/amd64
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
freebsd/arm64
illumos/amd64
js/wasm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
linux/mips64
linux/mips64le
linux/mipsle
```

Structure

...

Favorites

★



What is Clean Architecture?

- Independent of Frameworks. The architecture does not depend on the existence of some library or feature laden software.
- This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
- Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.



What is Clean Architecture?

- Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
- Independent of any external agency. In fact your business rules simply don't know anything at all about the outside world.



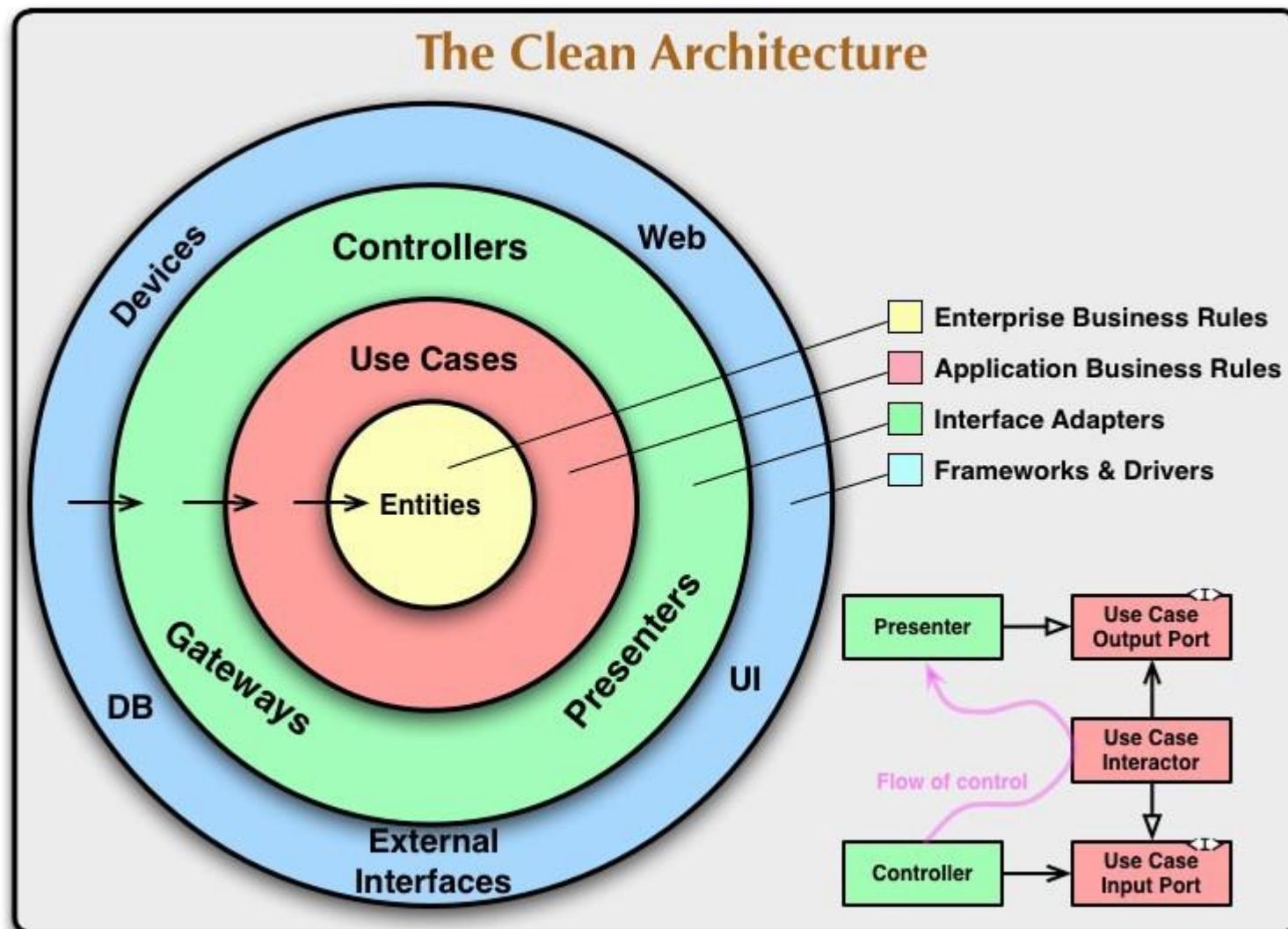
What is Clean Architecture?

- From Uncle Bob's Architecture we can divide our code in 4 layers :
- Entities: encapsulate enterprise wide business rules.
An entity in Go is a set of data structures and functions.
- Use Cases: the software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system.



What is Clean Architecture?

- Controller: the software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web
- Framework & Driver: this layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc.





Golang Architecture

- The top directory contains three directories:
 - app: application package root directory
 - cmd: main package directory
 - vendor: several vendor packages directory

```
% tree
.
├── Makefile
└── README.md
app
├── domain
│   ├── model
│   └── repository
│       └── service
├── interface
│   ├── persistence
│   │   └── rpc
│   └── registry
│       └── usecase
└── cmd
    └── 8am
        └── main.go
vendor
└── vendor packages
|...
```



Golang Architecture

- There are 4 layers, blue, green, red and yellow layers there in order from the outside. App directory has 3 layers except blue:
 - interface: the green layer
 - usecase: the red layer
 - domain: the yellow layer

```
% tree
.
├── Makefile
├── README.md
└── app
    ├── domain
    │   ├── model
    │   ├── repository
    │   └── service
    ├── interface
    │   ├── persistence
    │   └── rpc
    ├── registry
    └── usecase
└── cmd
    └── 8am
        └── main.go
vendor
└── vendor packages
|...
```



DOMAIN

```
$GOPATH/src/interfaces/repositories.go
package interfaces

type DbUserRepo DbRepo

func NewDbUserRepo(dbHandlers map[string]DbHandler) *DbUserRepo {
    dbUserRepo := new(DbUserRepo)
    dbUserRepo.dbHandlers = dbHandlers
    dbUserRepo.dbHandler = dbHandlers["DbUserRepo"]
    return dbUserRepo
}

func (repo *DbUserRepo) Store(user usecases.User) {
    isAdmin := "no"
    if user.isAdmin {
        isAdmin = "yes"
    }
    repo.dbHandler.Execute(fmt.Sprintf(`INSERT INTO users (id,
customer_id, is_admin)
VALUES (%d, %d, %v)`,
        user.Id, user.Customer.Id, isAdmin))
    customerRepo := NewDbCustomerRepo(repo.dbHandlers)
    customerRepo.Store(user.Customer)
}

func (repo *DbUserRepo) FindById(id int) usecases.User {
    row := repo.dbHandler.Query(fmt.Sprintf(`SELECT is_admin,
customer_id
FROM users WHERE id = '%d' LIMIT 1`,
        id))
    var isAdmin string
    var customerId int
    row.Next()
    row.Scan(&isAdmin, &customerId)
    customerRepo := NewDbCustomerRepo(repo.dbHandlers)
    u := usecases.User{Id: id, Customer:
        customerRepo.FindById(customerId)}
    u.isAdmin = false
    if isAdmin == "yes" {
        u.isAdmin = true
    }
    return u
}
```

```
$GOPATH/src/usecases/usecases.go
package usecases

import (
    "domain"
    "fmt"
)

type UserRepository interface {
    Store(user User)
    FindById(id int) User
}

type User struct {
    Id     int
    isAdmin bool
    Customer domain.Customer
}

type Item struct {
    Id     int
    Name   string
    Value  float64
}

type Logger interface {
    Log(message string) error
}

$GOPATH/src/domain/domain.go
package domain

import (
    "errors"
)

type ItemRepository interface {
    Store(item Item)
    FindById(id int) Item
}

type OrderRepository interface {
    Store(order Order)
    FindById(id int) Order
}

type CustomerRepository interface {
    Store(customer Customer)
    FindById(id int) Customer
}
```

USE
CASES

```
$GOPATH/src/interfaces/repositories.go
package interfaces

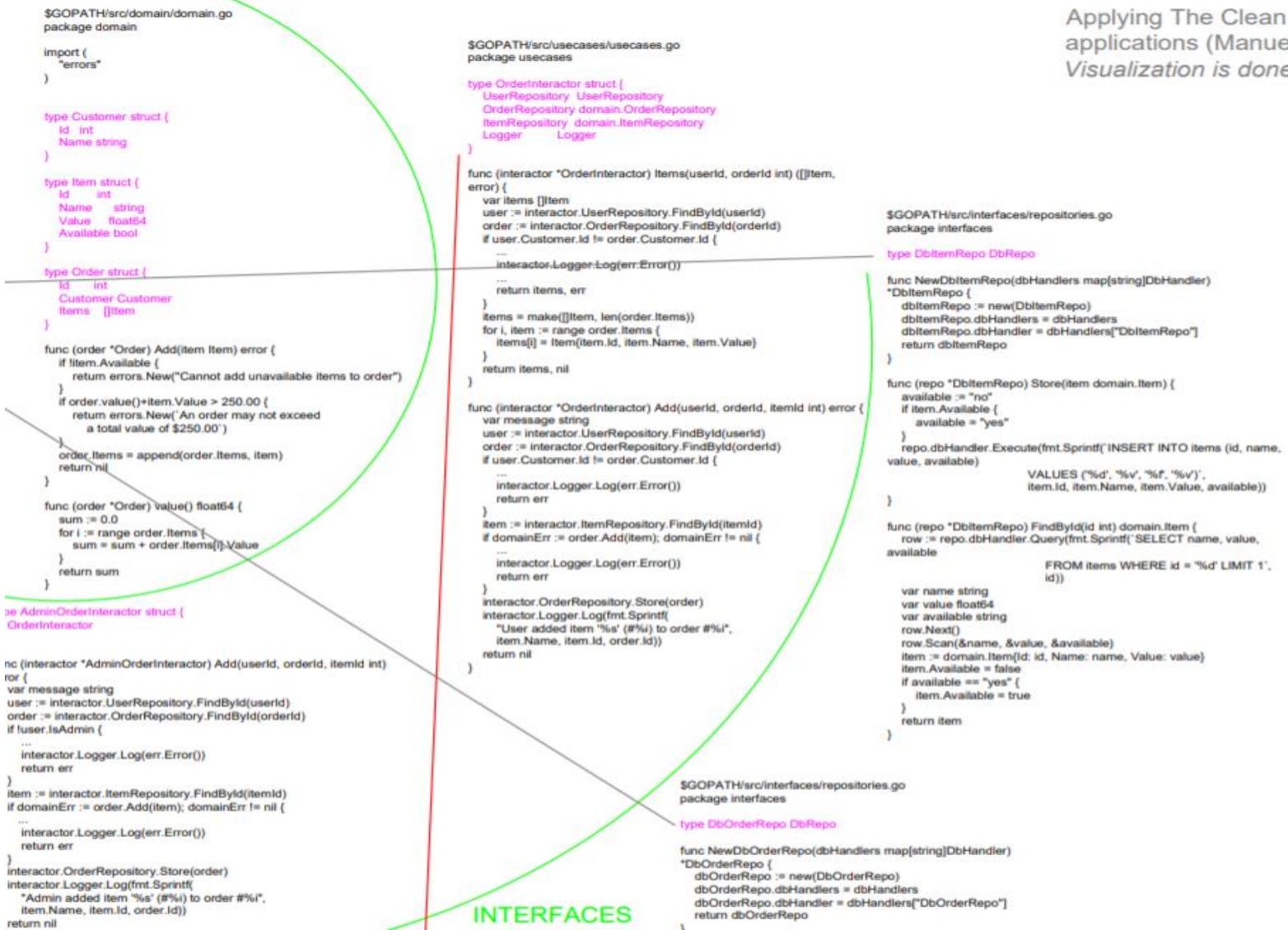
import (
    "domain"
    "fmt"
    "usecases"
)
```

\$GOPATH/src/interfaces/repositories.go
package interfaces



Applying The Clean Architecture to Go applications (Manuel Kiessling)

Visualization is done by Eduard Sesigin



Explicit Architecture

Primary/Driving Adapters

Secondary/Driven Adapters





Go Installation

Download and install

1. Go download.
2. Go install.
3. Go code.

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- [Managing Go installations](#) – How to install multiple versions and uninstall.
- [Installing Go from source](#) – How to check out the sources, build them on your own machine, and run them.

1. Go download.

Click the button below to download the Go installer.

[Download Go for Windows](#)

go1.15.6.windows-amd64.msi (115 MB)



Go Installation

Linux

Mac

Windows

If you have a previous version of Go installed, be sure to [remove it](#) before installing another.

1. Download the archive and extract it into /usr/local, creating a Go tree in /usr/local/go.

For example, run the following as root or through sudo:

```
tar -C /usr/local -xzf go1.15.6.linux-amd64.tar.gz
```

2. Add /usr/local/go/bin to the PATH environment variable.

You can do this by adding the following line to your \$HOME/.profile or /etc/profile (for a system-wide installation):

```
export PATH=$PATH:/usr/local/go/bin
```

Note: Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as source \$HOME/.profile.

3. Verify that you've installed Go by opening a command prompt and typing the following command:

```
$ go version
```

4. Confirm that the command prints the installed version of Go.



Go Installation



Download GoLand

[Windows](#) [Mac](#) [Linux](#)

GoLand includes an evaluation license key for a [free 30-day trial](#).

[Download](#)

Version: 2020.3.1
Build: 203.6682.164
30 December 2020

Installation Instructions

1. Unpack the `goland-2020.3.1.tar.gz` file to an empty directory using the following command: `tar -xzf goland- 2020.3.1.tar.gz`
2. Note: A new instance MUST NOT be extracted over an existing one. The target folder must be empty.
3. Run `goland.sh` from the `bin` subdirectory

Questions!

How do we run the code in our project?

What does '*package main*' mean?

What does '*import "fmt"*' mean?

What's that '*func*' thing?

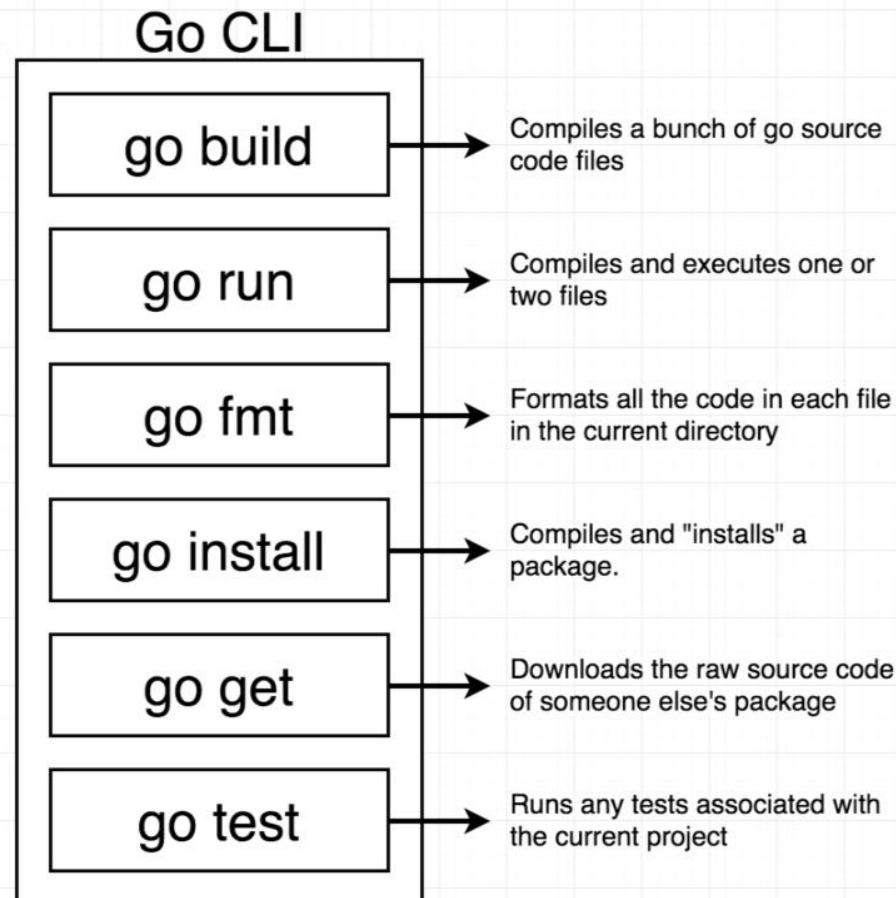
How is the `main.go` file organized?





Go

How do we run the code in our project?





Go

What does
'package main'
mean?

Package == Project == Workspace

+

What does
'package main'
mean?

Package Main

main.go

```
package main

import"fmt"

funcmain() {
    fmt.Println("Hi
there!")
}
```

support.go

```
package main

func support() {
    fmt.Println("I help!")
}
```

helper.go

```
package main

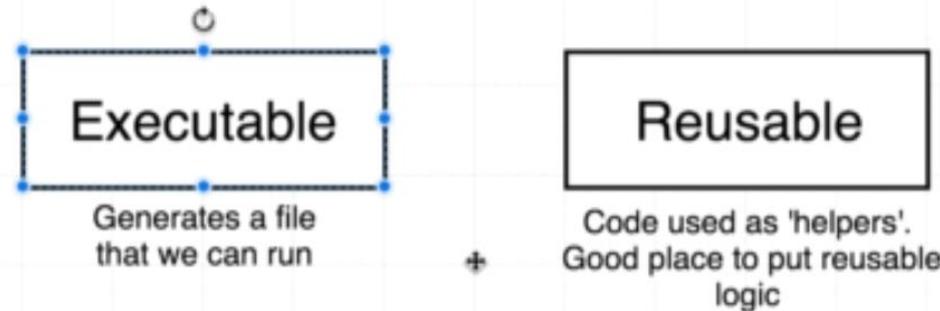
func help() {
    fmt.Println("I help too")
}
```



Go

What does
'package main'
mean?

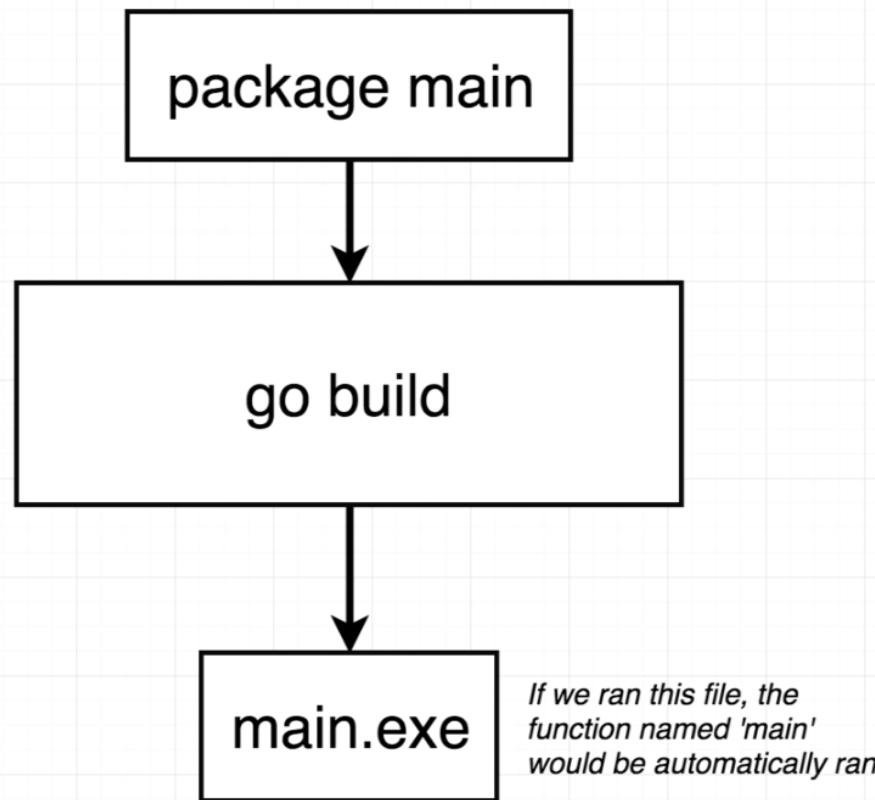
Types of Packages



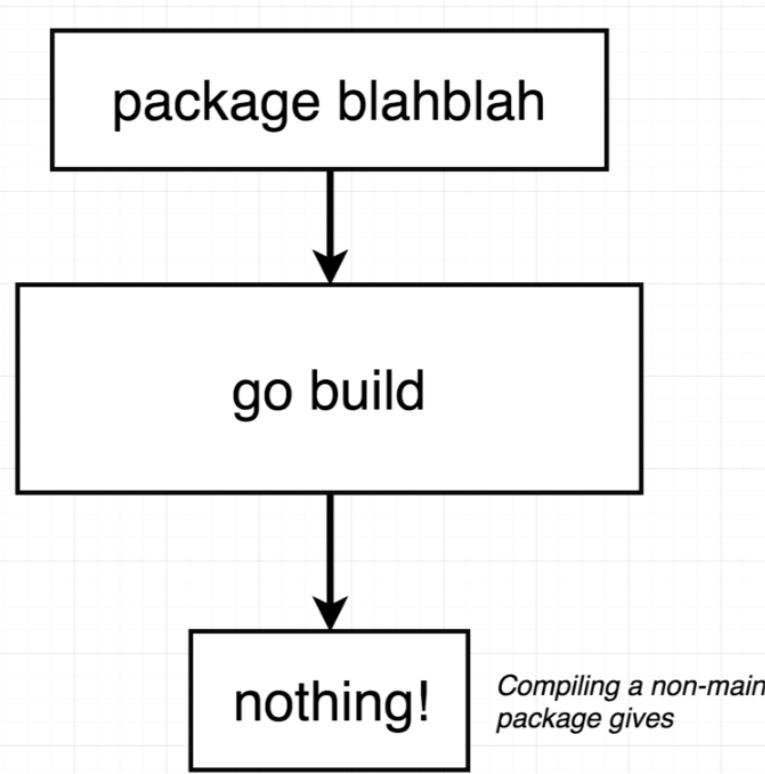


Go

What does
'package main'
mean?

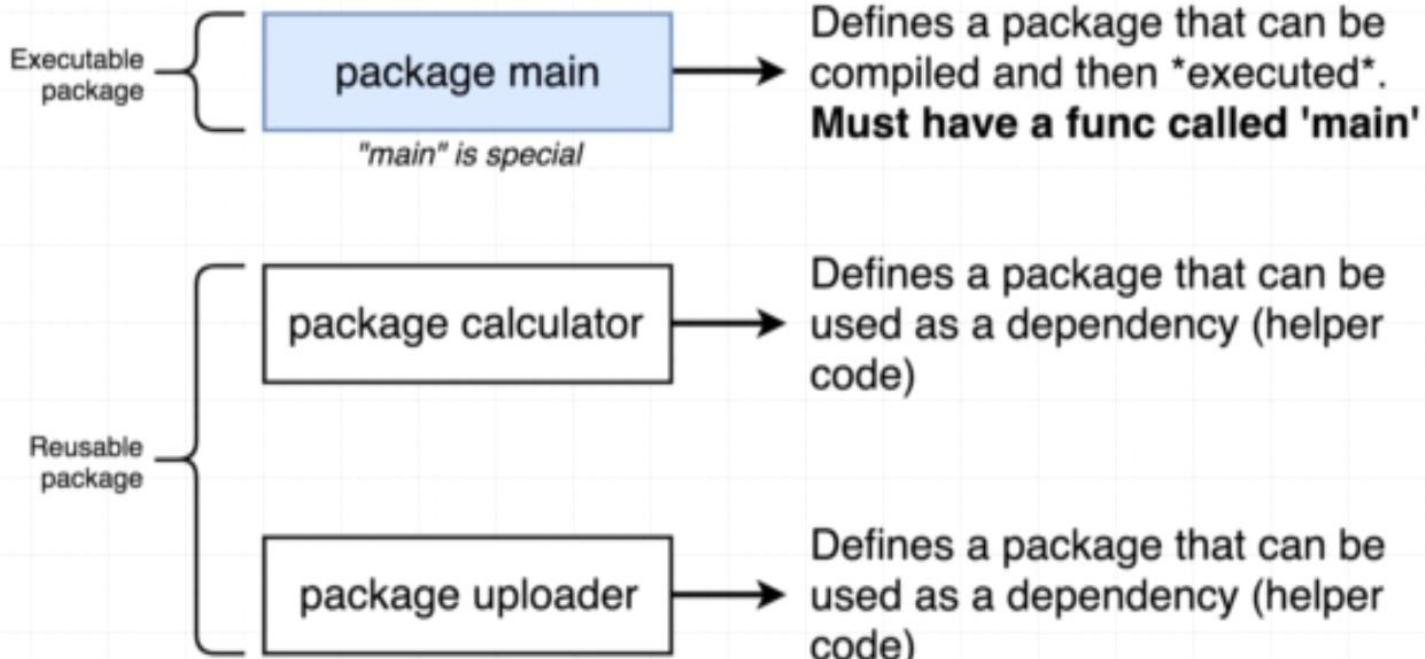


What does
'package main'
mean?



*Compiling a non-main
package gives*

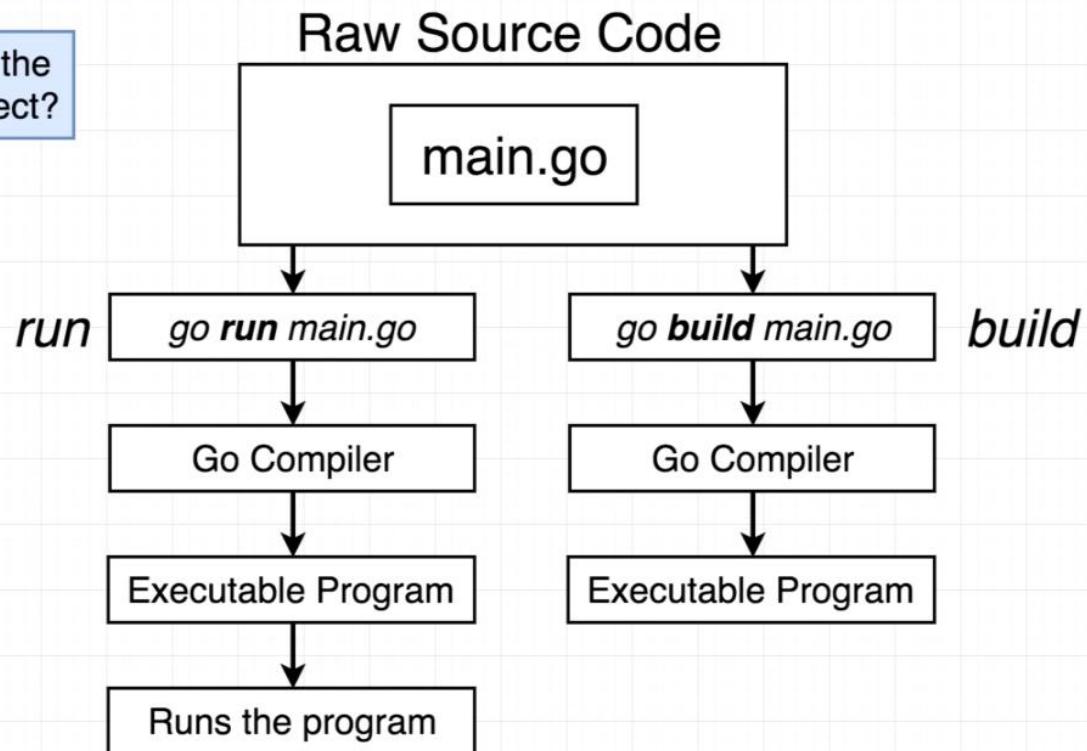
What does 'package main' mean?





Go

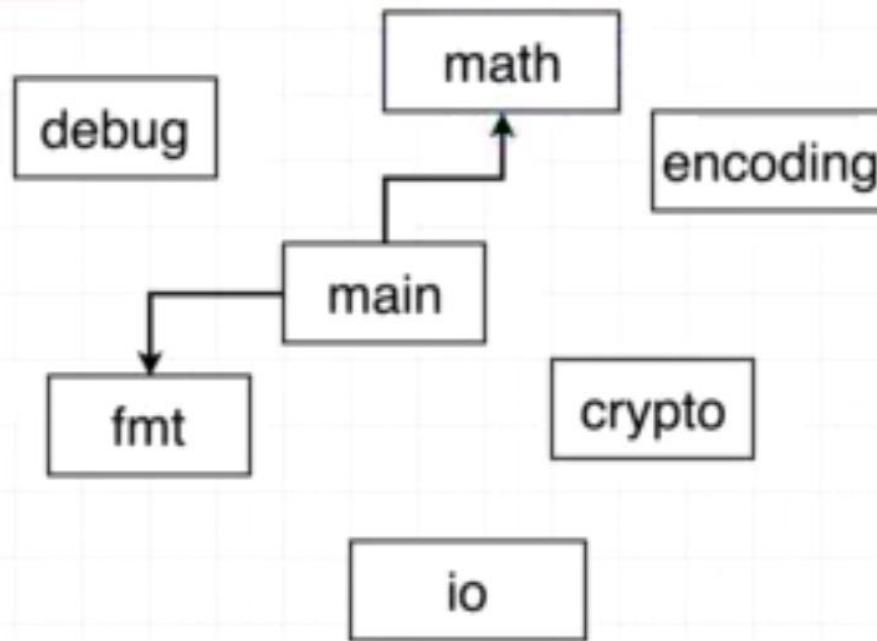
How do we run the code in our project?





Go

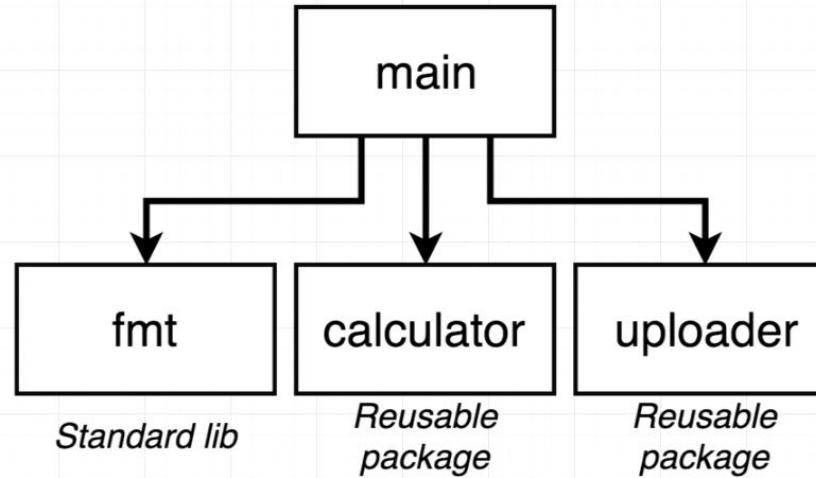
What does "import "fmt"" mean?





Go

What does 'import "fmt"' mean?

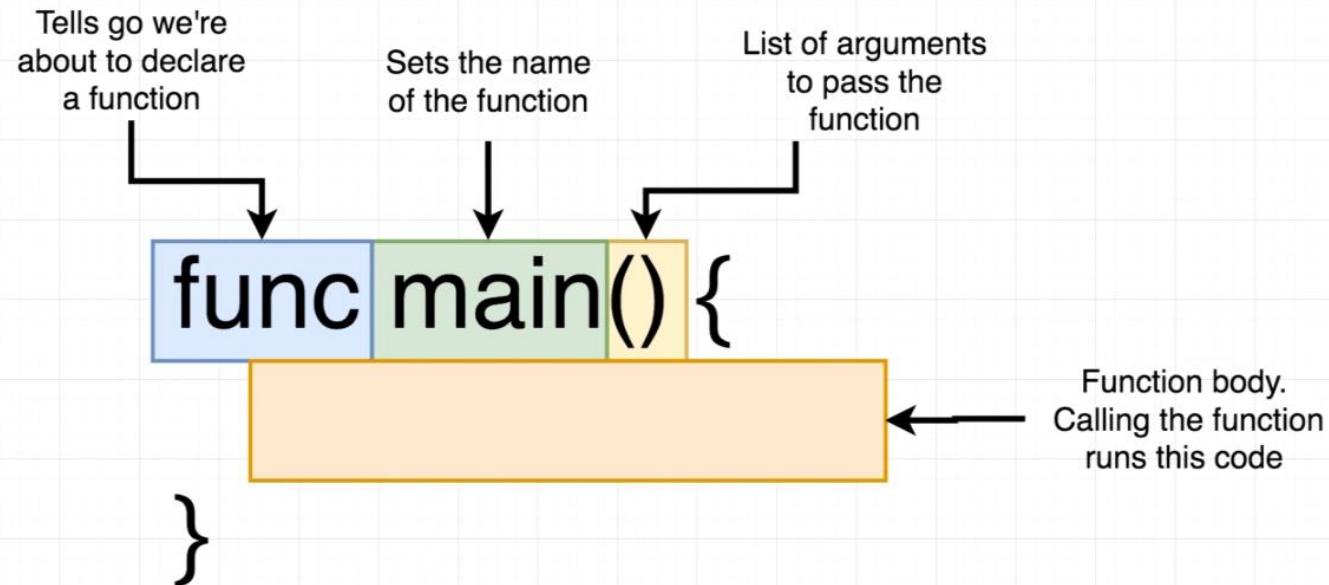


Go



golang.org/pkg

What's that 'func' thing?





How is the
main.go file
organized?

package main

Package declaration

import "fmt"

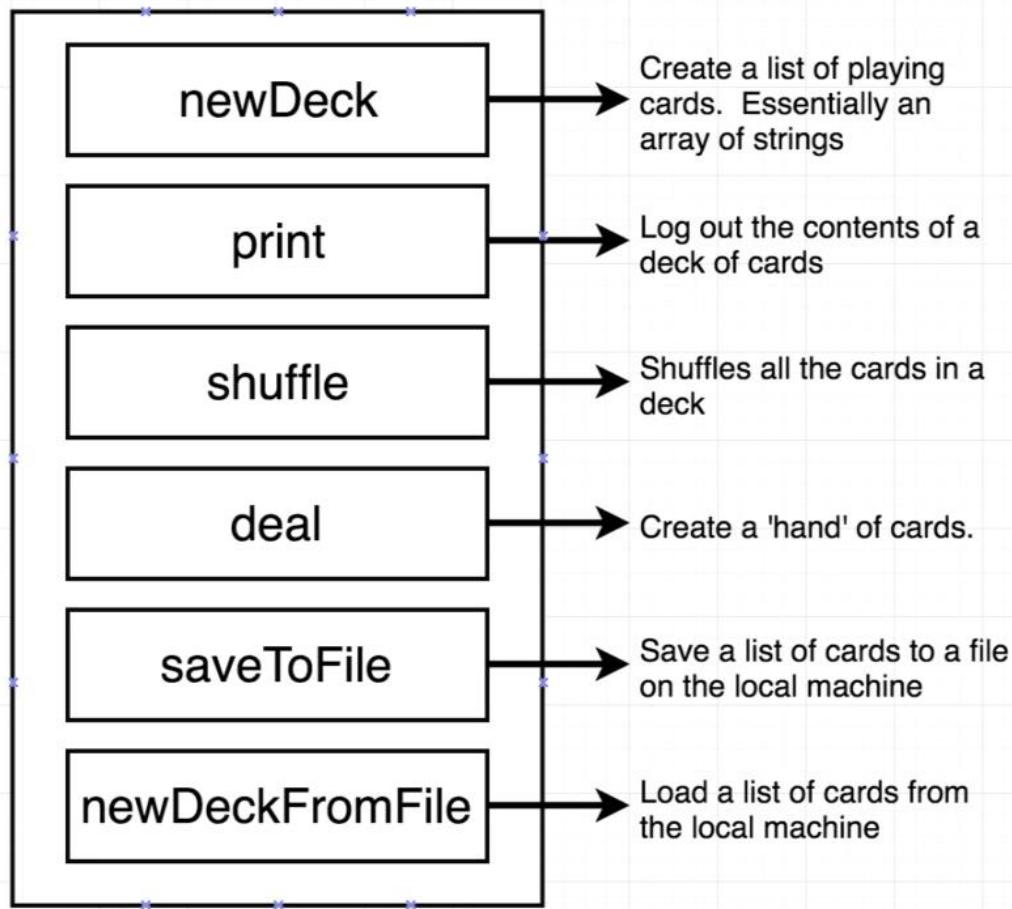
*Import other
packages that we
need*

func main() {
 fmt.Println("hi there")
}

*Declare functions, tell
Go to do things*



Cards





Golang Structure

- Every Go Program Contains the following Parts
 - Declaration of Packages
 - Package Importing
 - Functions
 - Variables
 - Expression and Statements
 - Comments

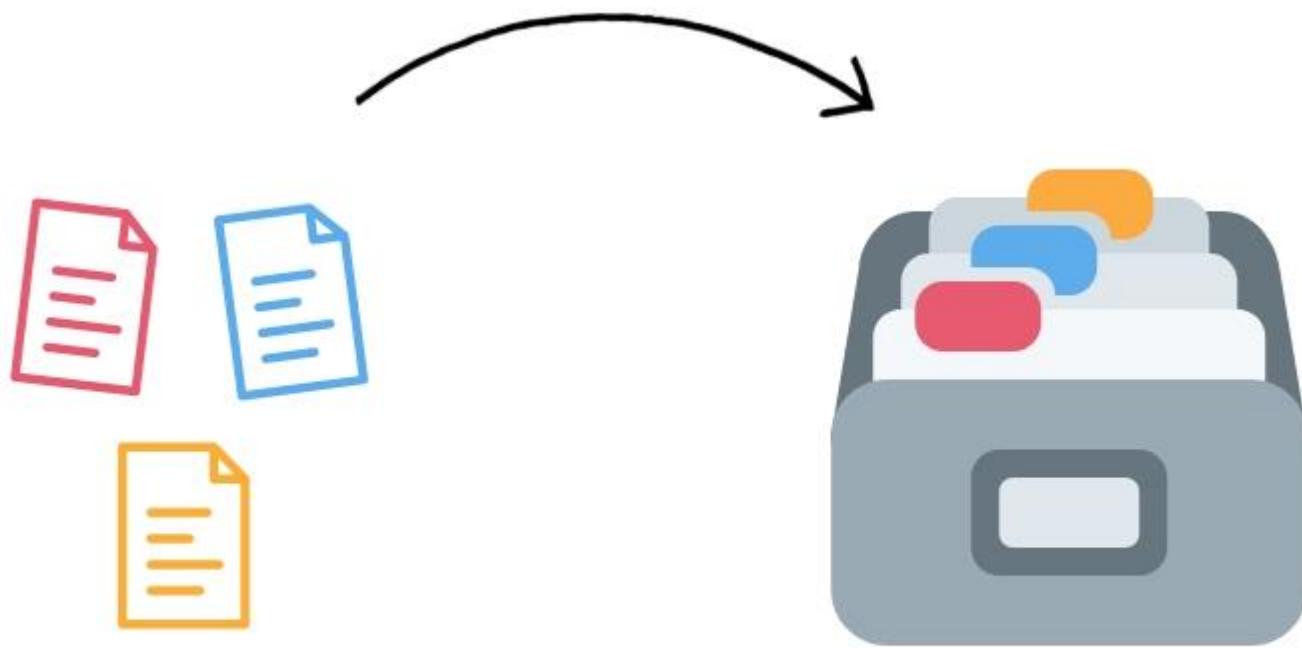


Golang Execution Steps

- Step1) Write the Go Program in a Text Editor.
- Step2) Save the program with “.go” as the program file extension.
- Step3) Go to command Prompt.
- Step4) In the command prompt, we have to open the directory where we have saved our Program.
- Step5) After opening the directory, we have to open our file and click enter to compile our code.
- Step6) If no errors are present in our code, then our program is executed, and the following output is displayed:



Packages



Go source files

Go Package



Packages

Standard library ▾

Name	Synopsis
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.
compress	
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzw	Package lzw implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.
container	
heap	Package heap provides heap operations for any type that implements heap.Interface.
list	Package list implements a doubly linked list.
ring	Package ring implements operations on circular lists.
context	Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.
crypto	
aes	Package aes implements AES encryption (formerly Rijndael), as defined in U.S. Federal Information Processing Standards Publication 197.
cipher	Package cipher implements standard block cipher modes that can be wrapped around low-level block cipher implementations.
des	Package des implements the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) as defined in U.S. Federal Information Processing Standards Publication 46-3.
dsa	Package dsa implements the Digital Signature Algorithm, as defined in FIPS 186-3.
ecdsa	Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-3.
ed25519	Package ed25519 implements the Ed25519 signature algorithm.





Packages

database	
sql	Package sql provides a generic interface around SQL (or SQL-like) databases.
driver	Package driver defines interfaces to be implemented by database drivers as used by package sql.
debug	
dwarf	Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at http://dwarfstd.org/doc/dwarf-2.0.0.pdf
elf	Package elf implements access to ELF object files.
gosym	Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.
macho	Package macho implements access to Mach-O object files.
pe	Package pe implements access to PE (Microsoft Windows Portable Executable) files.
plan9obj	Package plan9obj implements access to Plan 9 a.out object files.
encoding	
asci85	Package encoding defines interfaces shared by other packages that convert data to and from byte-level and textual representations.
asn1	Package asci85 implements the asci85 data encoding as used in the btoa tool and Adobe's PostScript and PDF document formats.
base32	Package asn1 implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.
base64	Package base32 implements base32 encoding as specified by RFC 4648.
binary	Package base64 implements base64 encoding as specified by RFC 4648.
csv	Package binary implements simple translation between numbers and byte sequences and encoding and decoding of varints.
gob	Package csv reads and writes comma-separated values (CSV) files.
hex	Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver).
json	Package hex implements hexadecimal encoding and decoding.
pem	Package json implements encoding and decoding of JSON as defined in RFC 7159.
xml	Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail.
errors	Package xml implements a simple XML 1.0 parser that understands XML name spaces.
expvar	Package errors implements functions to manipulate errors.
flag	Package expvar provides a standardized interface to public variables, such as operation counters in servers.
fmt	Package flag implements command-line flag parsing.
go	Package fmt implements formatted I/O with functions analogous to C's printf and scanf.
ast	Package go declares the types used to represent syntax trees for Go packages.
build	Package build gathers information about Go packages.
constant	Package constant implements Values representing untyped Go constants and their corresponding operations.
doc	Package doc extracts source code documentation from a Go AST.



Your First Program

```
package main

import "fmt"

// this is a comment

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



package main

- This is known as a “package declaration”.
- Every Go program must start with a package declaration.
- Packages are Go's way of organizing and reusing code.
- There are two types of Go programs: executables and libraries.
- **Executable** applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)
- **Libraries** are collections of code that we package together so that we can use them in other programs.



Packages and imports Every Go

```
package main

func main() {
    print("Hello, World!\n")
}
```

```
import "fmt"
import "math/rand"
```

```
import (
    "fmt"
    "math/rand"
)
```



Code Location

```
import "github.com/mattetti/goRailsYourself/crypto"
```

```
$ go get github.com/mattetti/goRailsYourself/crypto
```



Import fmt

- The import keyword is how we include code from other packages to use with our program.
- The fmt package (shorthand for format) implements formatting for input and output.



Fmt package

Printing

The verbs:

General:

```
%v      the value in a default format  
       when printing structs, the plus flag (%+v) adds field names  
%#v     a Go-syntax representation of the value  
%T      a Go-syntax representation of the type of the value  
%%      a literal percent sign; consumes no value
```

Boolean:

```
%t      the word true or false
```

Integer:

```
%b      base 2  
%c      the character represented by the corresponding Unicode code point  
%d      base 10  
%o      base 8  
%o      base 8 with 0o prefix  
%q      a single-quoted character literal safely escaped with Go syntax.  
%x      base 16, with lower-case letters for a-f  
%X      base 16, with upper-case letters for A-F  
%u      Unicode format: U+1234; same as "U+%04X"
```



Fmt package

Floating-point and complex constituents:

%b	decimalless scientific notation with exponent a power of two, in the manner of <code>strconv.FormatFloat</code> with the 'b' format, e.g. -123456p-78
%e	scientific notation, e.g. -1.234456e+78
%E	scientific notation, e.g. -1.234456E+78
%f	decimal point but no exponent, e.g. 123.456
%F	synonym for %f
%g	%e for large exponents, %f otherwise. Precision is discussed below.
%G	%E for large exponents, %F otherwise
%x	hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
%X	upper-case hexadecimal notation, e.g. -0X1.23ABCP+20

String and slice of bytes (treated equivalently with these verbs):

%s	the uninterpreted bytes of the string or slice
%q	a double-quoted string safely escaped with Go syntax
%x	base 16, lower-case, two characters per byte
%X	base 16, upper-case, two characters per byte

Slice:

%p	address of 0th element in base 16 notation, with leading 0x
----	---

Pointer:

%p	base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers, formatting the value exactly as if it were an integer.	



Fmt package

Index ▾

```
func Errorf(format string, a ...interface{}) error
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
func Print(a ...interface{}) (n int, err error)
func Printf(format string, a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
func Scan(a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
func Scanln(a ...interface{}) (n int, err error)
func Sprint(a ...interface{}) string
func Sprintf(format string, a ...interface{}) string
func Sprintln(a ...interface{}) string
func Sscan(str string, a ...interface{}) (n int, err error)
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
func Sscanln(str string, a ...interface{}) (n int, err error)
type Formatter
type GoStringer
type ScanState
type Scanner
type State
type Stringer
```



How To Write Comments in Go

```
// This is a comment
```

```
/*
```

```
Everything here  
will be considered  
a block comment
```

```
*/
```



Naming Variables: Rules and Style

- The naming of variables is quite flexible, but there are some rules to keep in mind:
- Variable names must only be one word (as in no spaces).
- Variable names must be made up of only letters, numbers, and underscores (_).
- Variable names cannot begin with a number.

Valid	Invalid	Why Invalid
userName	user-name	Hyphens are not permitted
name4	4name	Cannot begin with a number
user	\$user	Cannot use symbols
userName	user name	Cannot be more than one word



Basic Types

```
bool  
string
```

Numeric types:

```
uint      either 32 or 64 bits  
int       same size as uint  
uintptr   an unsigned integer large enough to store the uninterpreted bits of  
           a pointer value  
uint8     the set of all unsigned 8-bit integers (0 to 255)  
uint16    the set of all unsigned 16-bit integers (0 to 65535)  
uint32    the set of all unsigned 32-bit integers (0 to 4294967295)  
uint64    the set of all unsigned 64-bit integers (0 to 18446744073709551615)  
  
int8      the set of all signed 8-bit integers (-128 to 127)  
int16    the set of all signed 16-bit integers (-32768 to 32767)  
int32    the set of all signed 32-bit integers (-2147483648 to 2147483647)  
int64    the set of all signed 64-bit integers  
           (-9223372036854775808 to 9223372036854775807)  
  
float32   the set of all IEEE-754 32-bit floating-point numbers  
float64   the set of all IEEE-754 64-bit floating-point numbers  
  
complex64  the set of all complex numbers with float32 real and imaginary parts  
complex128 the set of all complex numbers with float64 real and imaginary parts  
  
byte      alias for uint8  
rune     alias for int32 (represents a Unicode code point)
```



Variables & inferred typing

The var statement declares a list of variables with the type declared last.

```
var (
    name      string
    age       int
    location string
)
```

Or even

```
var (
    name, location string
    age           int
)
```



Variables & inferred typing

Variables can also be declared one by one:

```
var name      string
var age       int
var location  string
```

A var declaration can include initializers, one per variable.

```
var (
    name      string = "Prince Oberyn"
    age       int    = 32
    location  string = "Dorne"
)
```

If an initializer is present, the type can be omitted, the variable will take the type of the initializer (inferred typing).

```
var (
    name      = "Prince Oberyn"
    age       = 32
    location = "Dorne"
)
```



Variables & inferred typing

You can also initialize variables on the same line:

```
var (
    name, location, age = "Prince Oberyn", "Dorne", 32
)
```

Inside a function, the `:=` short assignment statement can be used in place of a var declaration with implicit type.

```
func main() {
    name, location := "Prince Oberyn", "Dorne"
    age := 32
    fmt.Printf("%s (%d) of %s", name, age, location)
}
```

A variable can contain any type, including functions:

```
func main() {
    action := func() {
        //doing something
    }
    action()
}
```



Constants

```
const Pi = 3.14
const (
    StatusOK          = 200
    StatusCreated     = 201
    StatusAccepted    = 202
    StatusNonAuthoritativeInfo = 203
    StatusNoContent   = 204
    StatusResetContent = 205
    StatusPartialContent = 206
)
```

```
const (
    Pi      = 3.14
    Truth   = false
    Big     = 1 << 62
    Small   = Big >> 61
)

func main() {
    const Greeting = ""
    fmt.Println(Greeting)
    fmt.Println(Pi)
    fmt.Println(Truth)
    fmt.Println(Big)
}
```



Basic Types

```
package main

import (
    "fmt"
    "math/cmplx"
)

var (
    goIsFun bool        = true
    maxInt  uint64      = 1<<64 - 1
    complex complex128 = cmplx.Sqrt(-5 + 12i)
)

func main() {
    const f = "%T(%v)\n"
    fmt.Printf(f, goIsFun, goIsFun)
    fmt.Printf(f, maxInt, maxInt)
    fmt.Printf(f, complex, complex)
}
```

```
bool(true)
uint64(18446744073709551615)
complex128((2+3i))
```



Type conversion

The expression **T(v)** converts the value **v** to the type **T**. Some numeric conversions:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

```
package main

b := 125.0
c := 390.8
fmt.Println(int(b))
fmt.Println(int(c))

func main() {
    a := strconv.Itoa(12)
    fmt.Printf("%q\n", a)
}
```



Type conversion

```
a := "not a number"
b, err := strconv.Atoi(a)
fmt.Println(b)
fmt.Println(err)
```



Global and Local Variables

```
package main
```

```
import "fmt"
```

```
var g = "global"
```

```
func printLocal() {
```

```
    l := "local"
```

```
    fmt.Println(l)
```

```
}
```

```
func main() {
```

```
    printLocal()
```

```
    fmt.Println(g)
```

```
}
```

A large, light blue starburst shape with an arrow pointing towards the word "global".

global

A large, light blue starburst shape with an arrow pointing towards the word "local".

local



Data Types

DATA TYPES

Numeric

String

Boolean

Derived

int

float

true

false

Pointer

Array

Structure

Mp

Interface



Data Types

Data Type	Range
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 4294967295
uint64	0 to 18446744073709551615

Data Type	Range
int8	-128 to 127
int16	-32768 to 32767
int32	-2147483648 to 2147483647
int64	-9223372036854775808 to 9223372036854775808



Data Types

Go has the following architecture-independent integer types:

```
uint8      unsigned 8-bit integers (0 to 255)
uint16     unsigned 16-bit integers (0 to 65535)
uint32     unsigned 32-bit integers (0 to 4294967295)
uint64     unsigned 64-bit integers (0 to 18446744073709551615)
int8       signed 8-bit integers (-128 to 127)
int16      signed 16-bit integers (-32768 to 32767)
int32      signed 32-bit integers (-2147483648 to 2147483647)
int64      signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
```

Floats and complex numbers also come in varying sizes:

```
float32    IEEE-754 32-bit floating-point numbers
float64    IEEE-754 64-bit floating-point numbers
complex64  complex numbers with float32 real and imaginary parts
complex128 complex numbers with float64 real and imaginary parts
```

There are also a couple of alias number types, which assign useful names to specific data types:

```
byte       alias for uint8
rune      alias for int32
```



Data Types

`uint` unsigned, either 32 or 64 bits

`int` signed, either 32 or 64 bits

`uintptr` unsigned integer large enough to store the uninterpreted bits of a pointer value



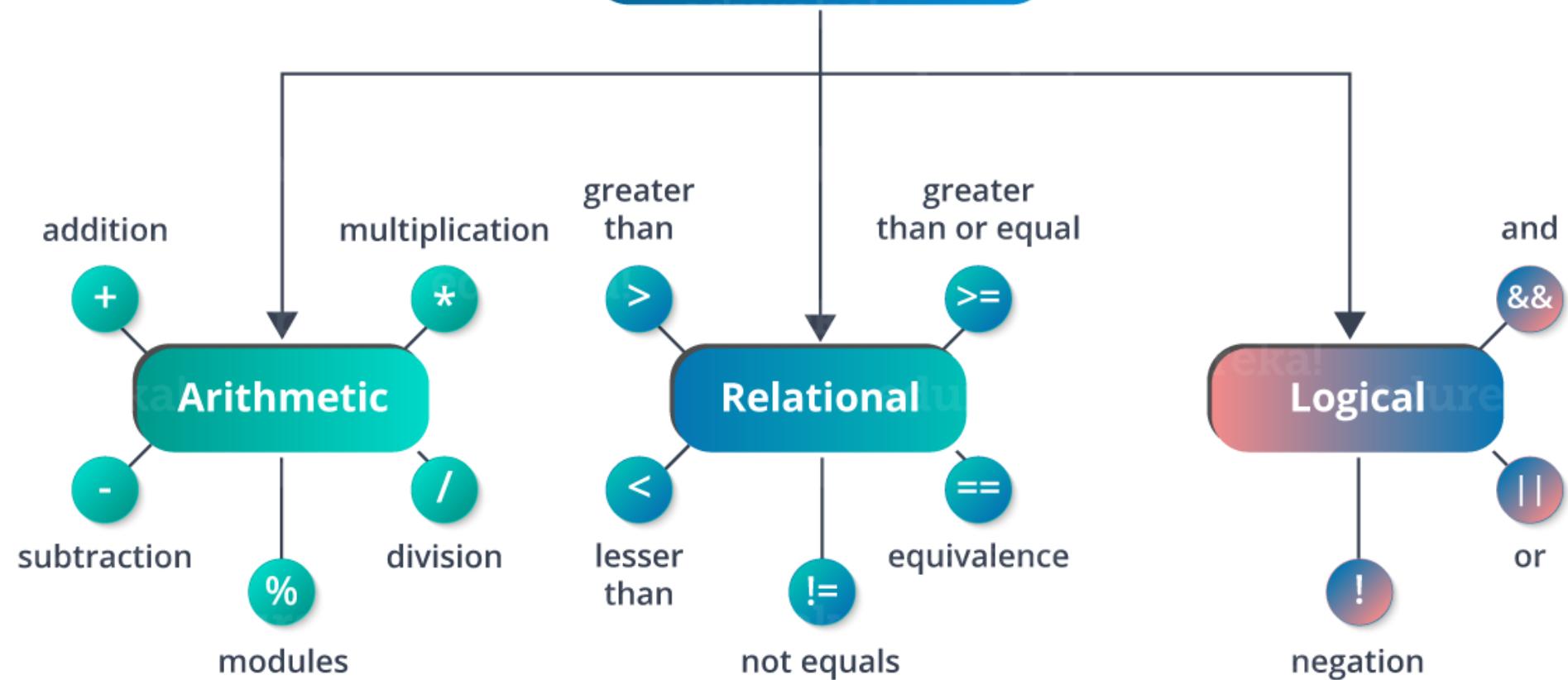
Keywords

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	If	range	type
continue	for	import	return	var



Operators

Types of Operators





Array

Array

*Fixed length list
of things*

Slice

*An array that can
grow or shrink*





Array

Slices

"Five of Spades" "Three of Diamonds" "Five of Diamonds"

*Every element in a slice must
be of same type*

"Five of Spades" 55525235 "Five of Diamonds"



Array

```
import "fmt"

func main() {
    cards := []string{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")

    for i, card := range cards {
        fmt.Println(i, card)
    }
}

func newCard() string {
    return "Five of Diamonds"
}
```



Array

```
index of this  
element in the  
array  
for index, card := range cards {  
    fmt.Println(card)  
}  
Current card  
we're iterating  
over  
Take the slice  
of 'cards' and  
loop over it  
Run this one time  
for each card in the  
slice
```





Array

Question 3:

How do we iterate through each element in a slice and print out its value?

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for value in colors {  
    fmt.Println(value)  
}
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ colors.forEach(func(value, index) {  
    fmt.Println(value)  
})
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for index, color := colors {  
    fmt.Println(index, color)  
}
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for index, color := range colors {  
    fmt.Println(index, color)  
}
```



Arrays

- Define Array
 - *[capacity]data_type{element_values}*
 - *[4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}*
- `coral := [4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}`
- `fmt.Println(coral)`



Arrays

- var myArray1 = [3]int // will be filled with so called zero values, for integers: 0
- var myArray2 = [5]int{1,2,3,4,5} // number of values between { } can not be larger than size (ofc)
- var myArray3 = [...]int{1,2,3,4} // the compiler will count the array elements for you



Arrays

- ar variable_name [SIZE1][SIZE2]...[SIZEN]
variable_type
- a := [3][4]int{
 - {0, 1, 2, 3} , // initializers for row indexed by 0
 - {4, 5, 6, 7} , // initializers for row indexed by 1 */
 - {8, 9, 10, 11} // initializers for row indexed by 2
- }



Creating slices

- From an existing array
- We already stated slices are an abstraction over the array, so let's grab a slice from an array.
- `myArray := [10]int{0,1,2,3,4,5,6,7,8,9} // firstly we need an array to make the slice out of`
- `mySlice1 := myArray[1:5] // slice from 1st up to 5th element, so [1 2 3 4]`
- `mySlice2 := mySlice1[0:2] // slicing a slice // [1 2]`



Creating slices

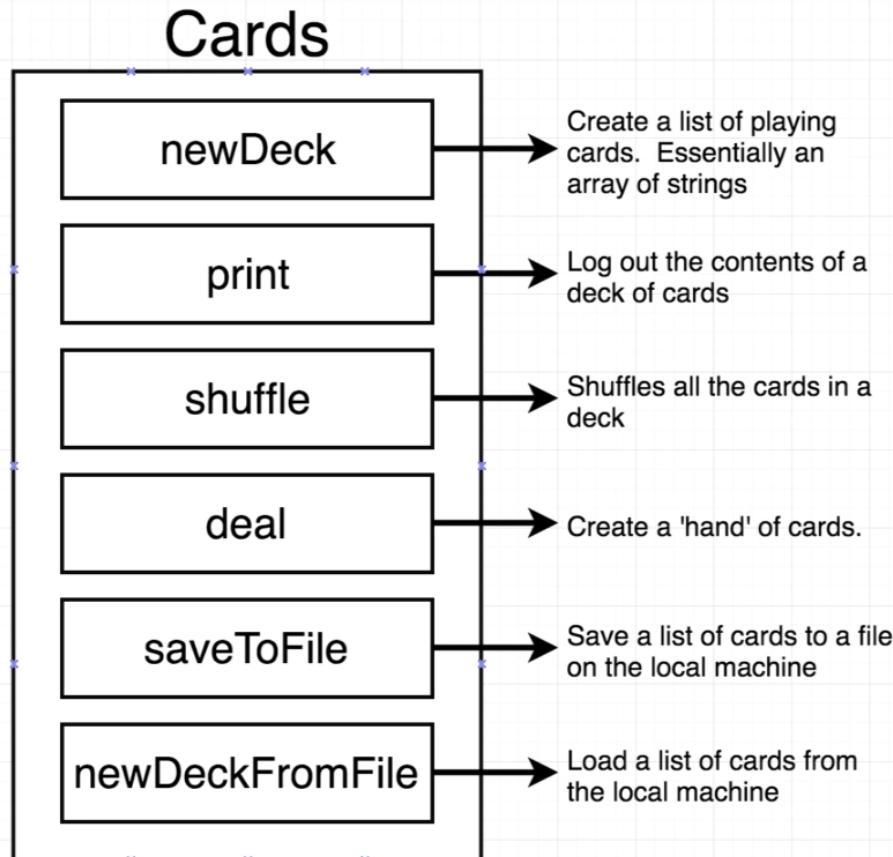
- `var mySlice3 := make([]int, 4, 4)` // mySlice has length and
- // capacity 5, and it's
- // filled with zero values
- // for int, so it's [0 0 0 0]
- `mySlice5 := make([]int, 2, 4)` // now it's [0 0 nil nil]
 - // but be aware that nil
 - // values don't print, so
 - // when printed it will be
 - // just [0 0]



Arrays

- `regNos := make([]int32, 100)`
- `for r:=range regNos{`
- `regNos[r] = rand.Int31n(1000)`
- `}`
- `for index, val := range regNos{`
- `fmt.Printf("%d = %d\n", index, val)`
- `}`

Use case





Use case

```
var card string = "Ace of Spades"
```

We're about to
create a new
variable

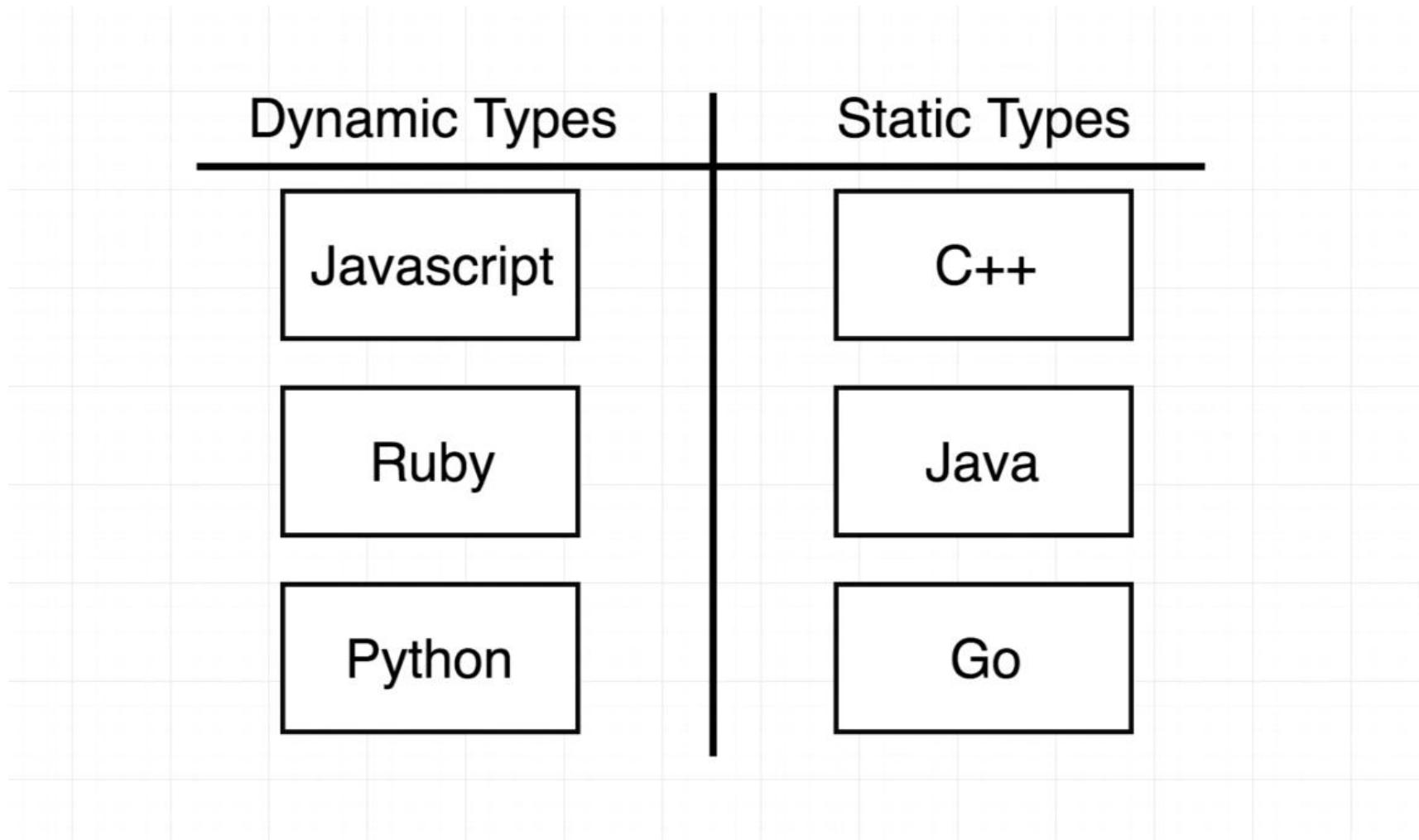
The name of the
variable will be
'greeting'

Only a "string" will
ever be assigned
to this variable

Assign the value
"Ace of Spades"
to this variable



Use case





Use case

Question 4:

Will the following code compile? Why or why not?

```
1 | paperColor := "Green"  
2 | paperColor := "Blue"
```

- Yes, because we are creating and assigning a value to the variable 'paperColor' twice.
- No, because a variable can only be initialized one time. In this case, the ':=' operator is being used to initialize 'paperColor' two times
- No, because ':=' is not a valid operator



Use case

Question 6:

This might require a bit of experimentation on your end :). Remember that you can use the Go Playgorund at <https://play.golang.org/> to quickly run a snippet of code.

Is the following code valid?

```
1 | package main
2 |
3 | import "fmt"
4 |
5 | deckSize := 20
6 |
7 | func main() {
8 |     fmt.Println(deckSize)
9 | }
```

Yes

No



Functions

Define a function
called 'newCard'

```
func newCard() string {  
}
```

When executed, this
function will return a
value of type 'string'



Functions

Question 5:

Here's a tough one! Imagine we have two files in the same folder with the same package name. Will the following code successfully compile? This might take a little experimentation on your side. If you do try this out yourself, run your code with the command `go run main.go state.go`.

In `main.go`:

```
1 | package main
2 |
3 | func main() {
4 |     printState()
5 | }
```

In a separate file called `state.go`:

```
1 | package main
2 |
3 | import "fmt"
4 |
5 | func printState() {
6 |     fmt.Println("California")
7 | }
```

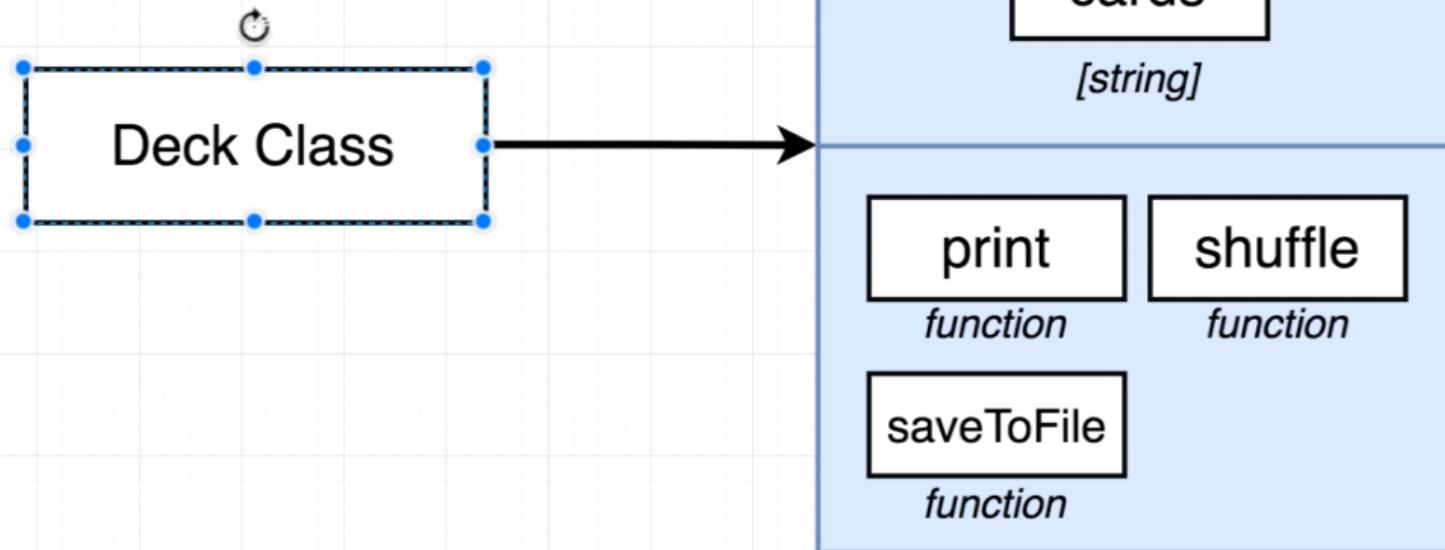
No, because the '`state.go`' file must be imported because it can be used.

Yes, because both files are apart of the same package.

OO vs Go Structure

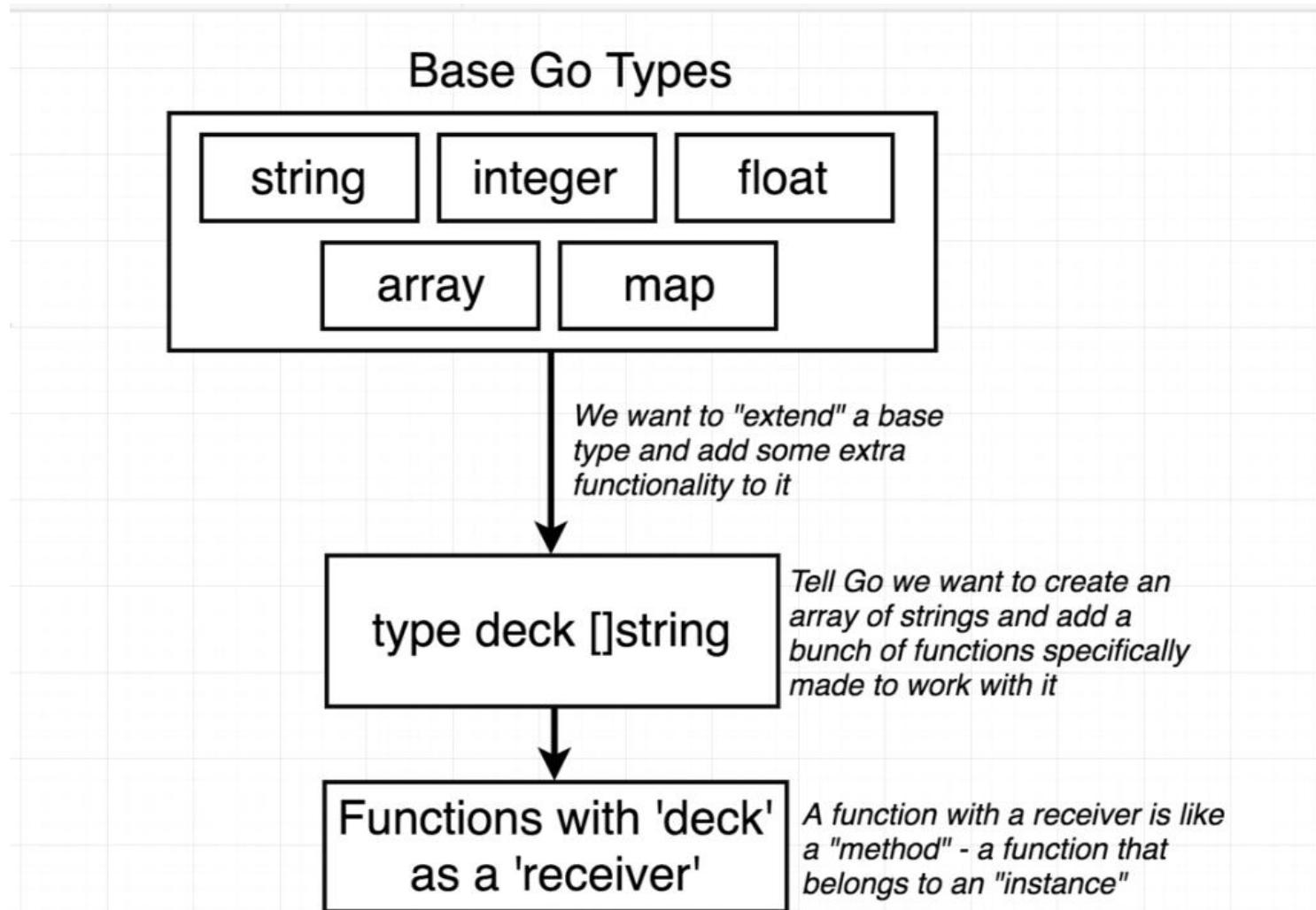
Cards - OO Approach

Deck Instance





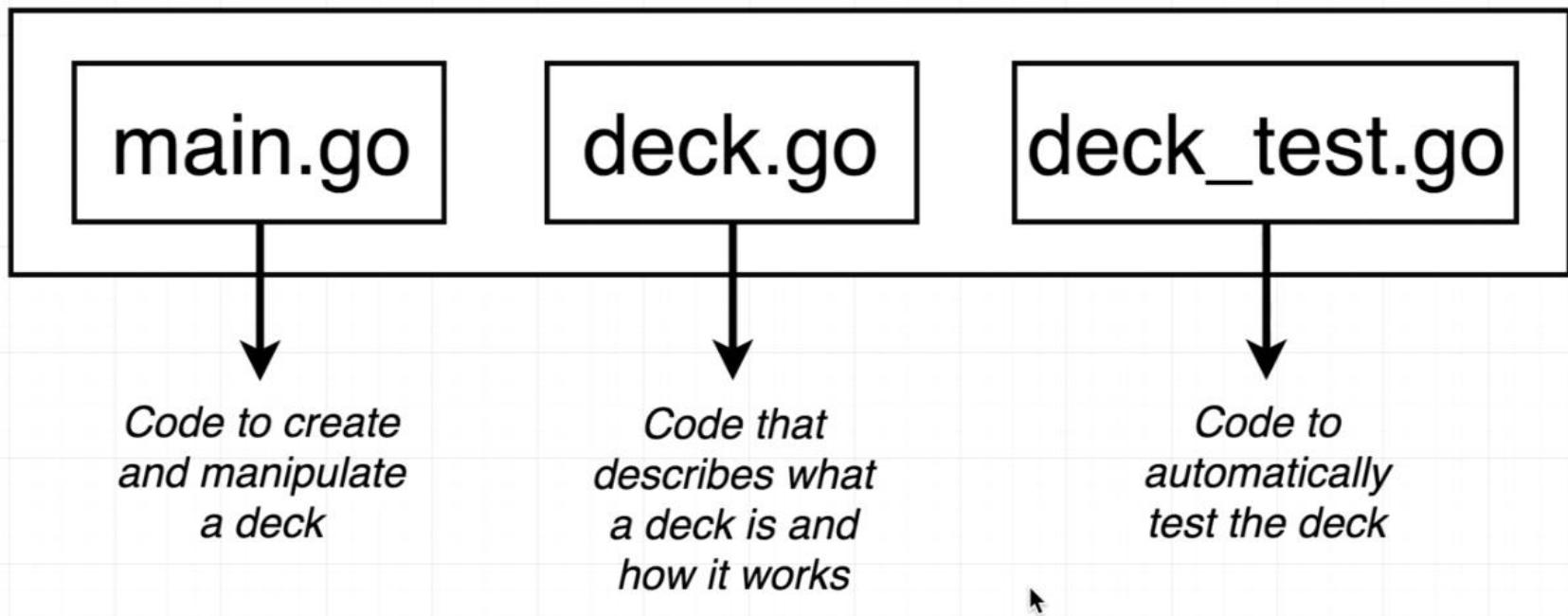
OO vs Go Structure





OO vs Go Structure

' cards' folder





Custom Type

```
package main

// Create a new type of 'deck'
// which is a slice of strings
type deck []string
```



Custom Type

```
func (d deck) print() {
```

Any variable of type "deck"
now gets access to the
"print" method



Custom Type

The actual copy of the deck we're working with is available in the function as a variable called 'd'

Every variable of type 'deck' can call this function on itself

```
func (d deck) print() {  
    for i, card := range d {  
        fmt.Println(i, card)  
    }  
}
```



Custom Type

Is the following code valid?

```
1 | type laptopSize float64
2 |
3 | func (this laptopSize) getSizeOfLaptop() laptopSize {
4 |     return this
5 | }
```

- No, it would fail to compile
- Yes, and there is nothing wrong with it.
- Yes, but it is breaking convention, Go avoids any mention of 'this' or 'self'



Case Study

```
cards = deck{}
```

```
cardSuits := []string{"Spades", "Hearts", "Diamonds"}
```

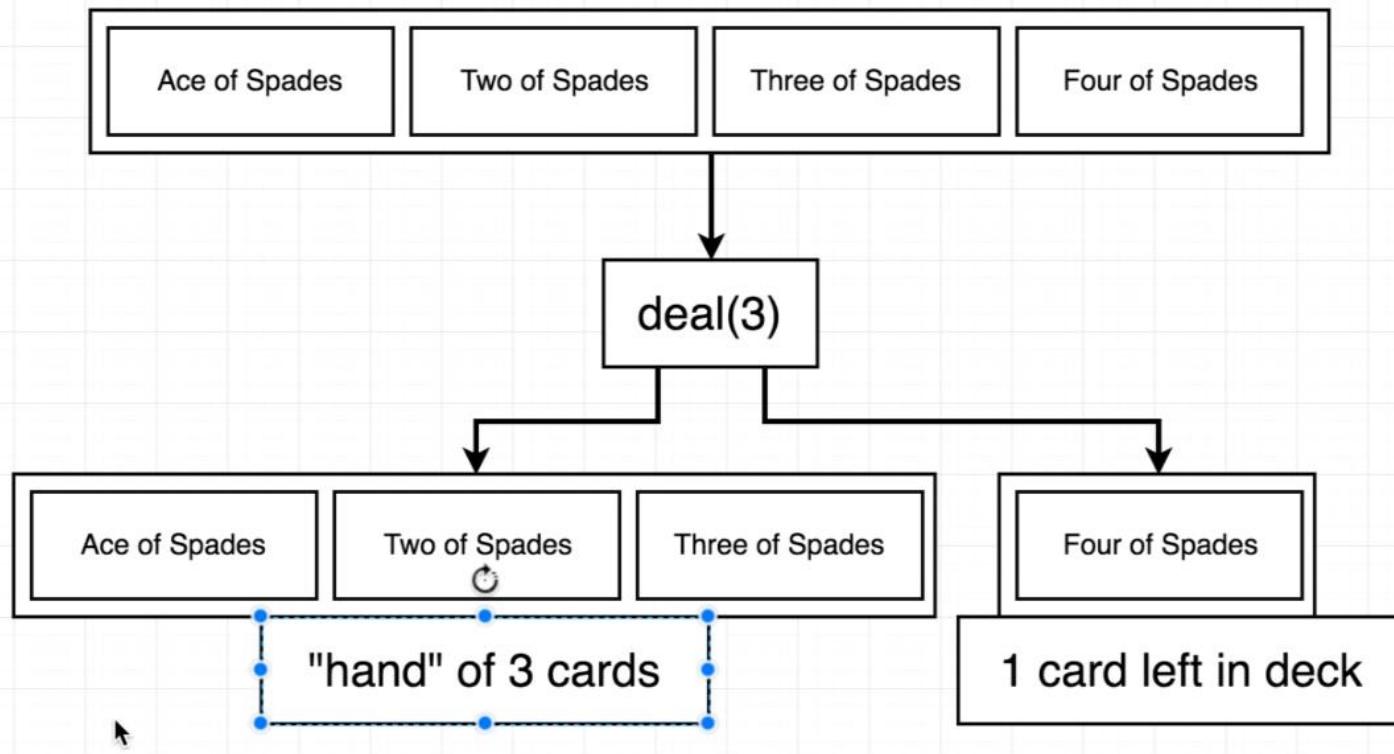
```
cardValues := []string{"Ace", "Two", "Three"}
```

```
for each suit in cardSuits
```

```
    for each value in cardValues
```

```
        Add a new card of 'value of suit' to the 'cards' deck
```

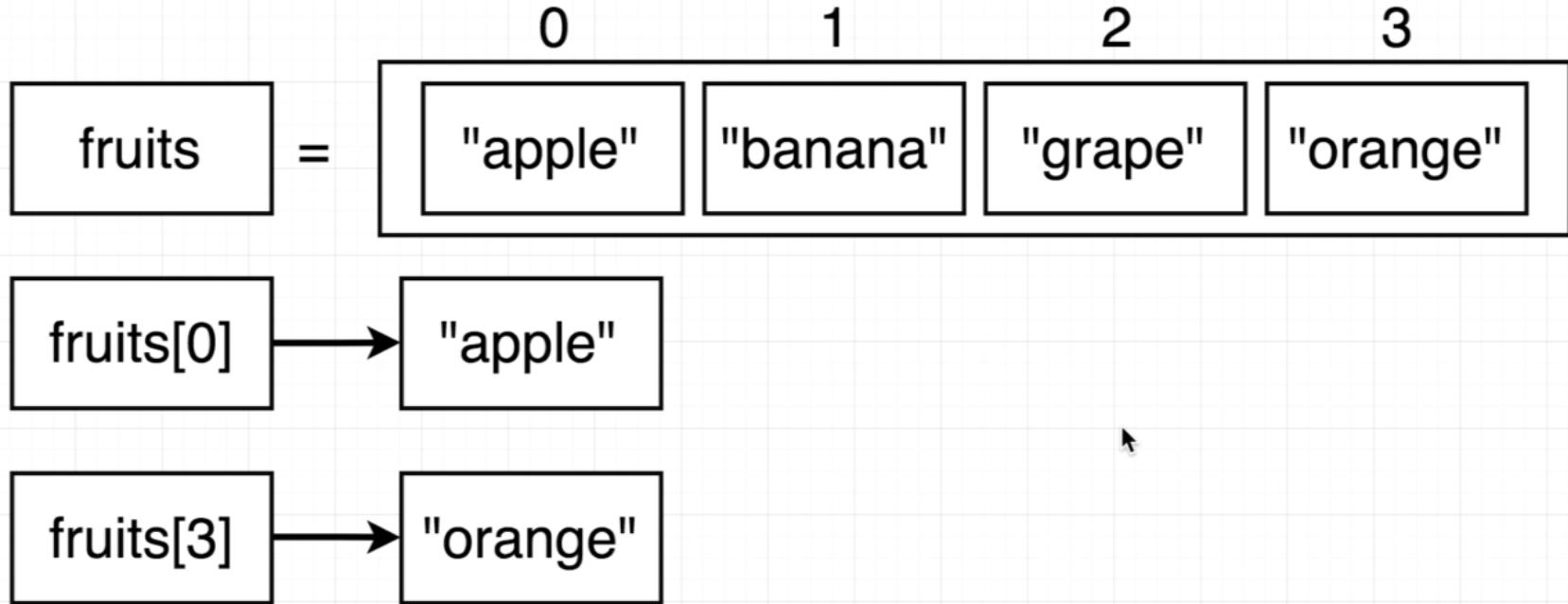
Case Study





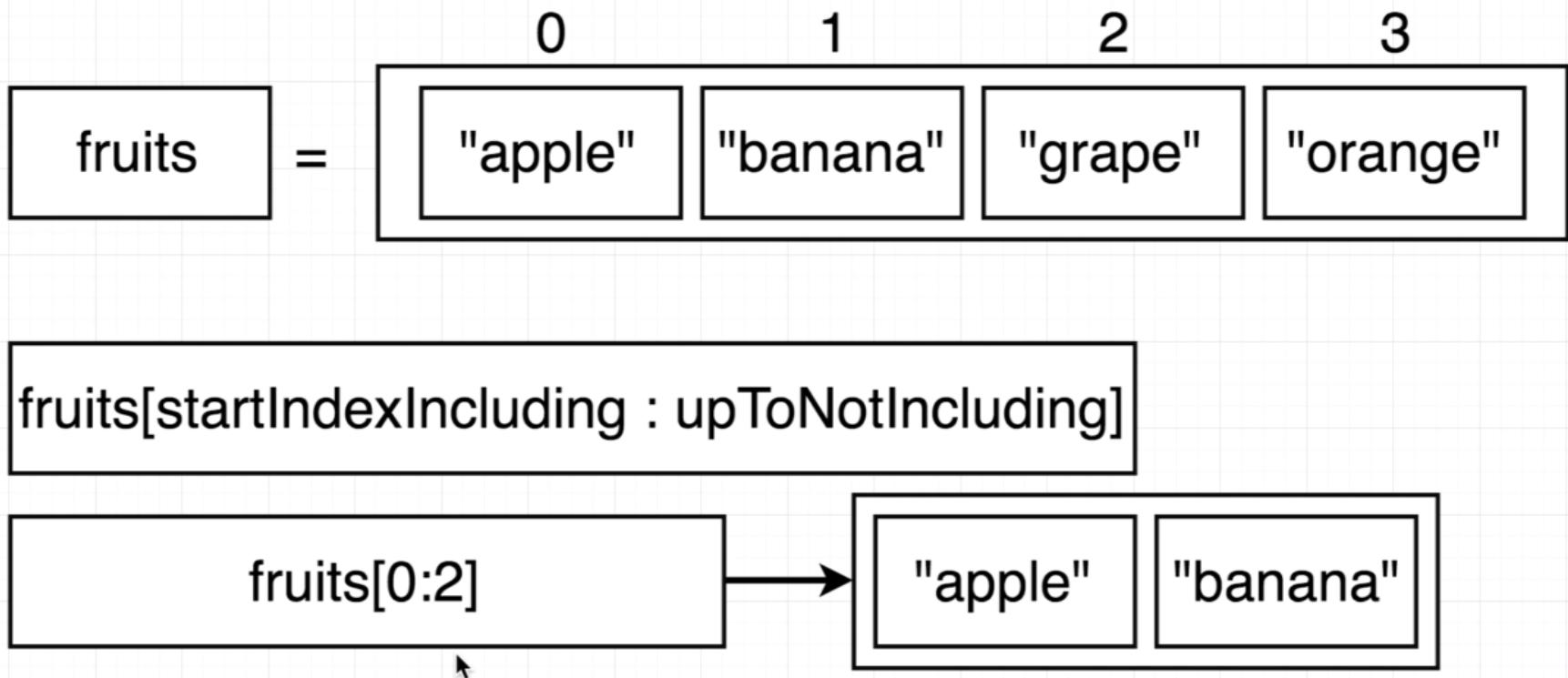
Case Study

Slices are zero-indexed



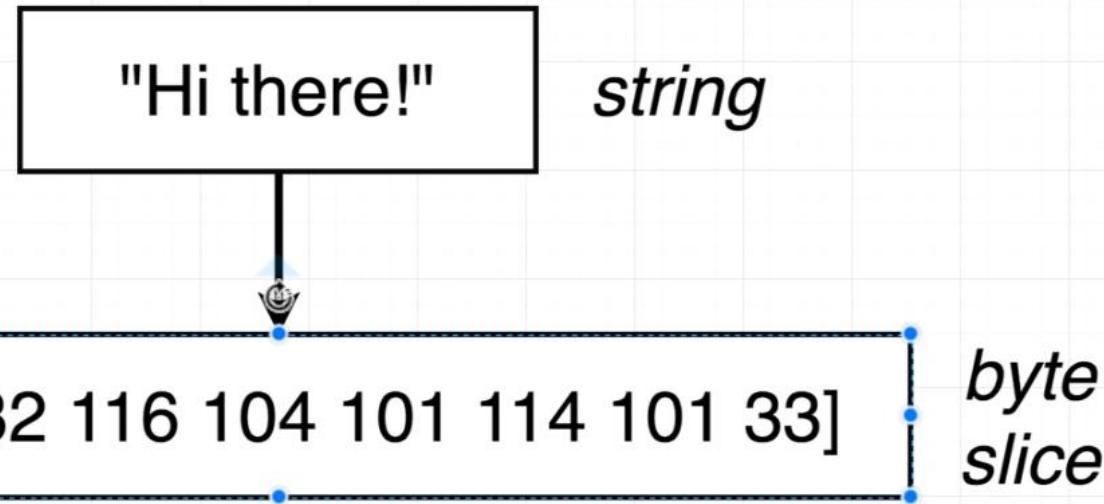


Case Study





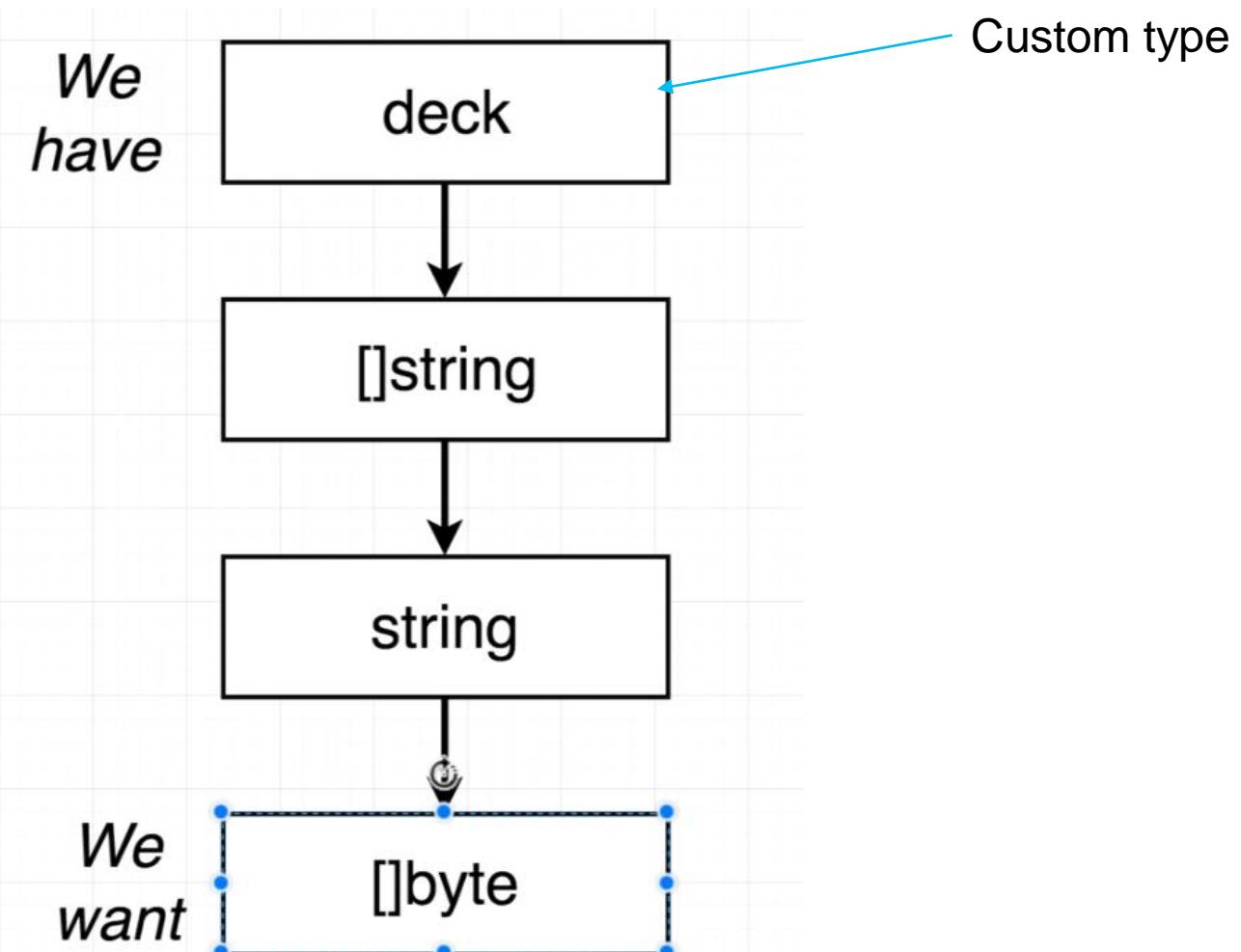
Byte Slice



asciitable.com



Byte Slice





Pointers

- A data type called a pointer holds the memory address of the data, but not the data itself.
- The memory address tells the function where to find the data, but not the value of the data.
- You can pass the pointer to the function instead of the data, and the function can then alter the original variable in place.
- This is called passing by reference, because the value of the variable isn't passed to the function, just its location.



Pointers

- An ampersand in front of a variable name is to get the address, or a pointer to that variable.
- Asterisk (*) or dereferencing operator prefixed with an * creates variable and initializes the pointer with the address.
- **var myPointer *int32 = &someint**

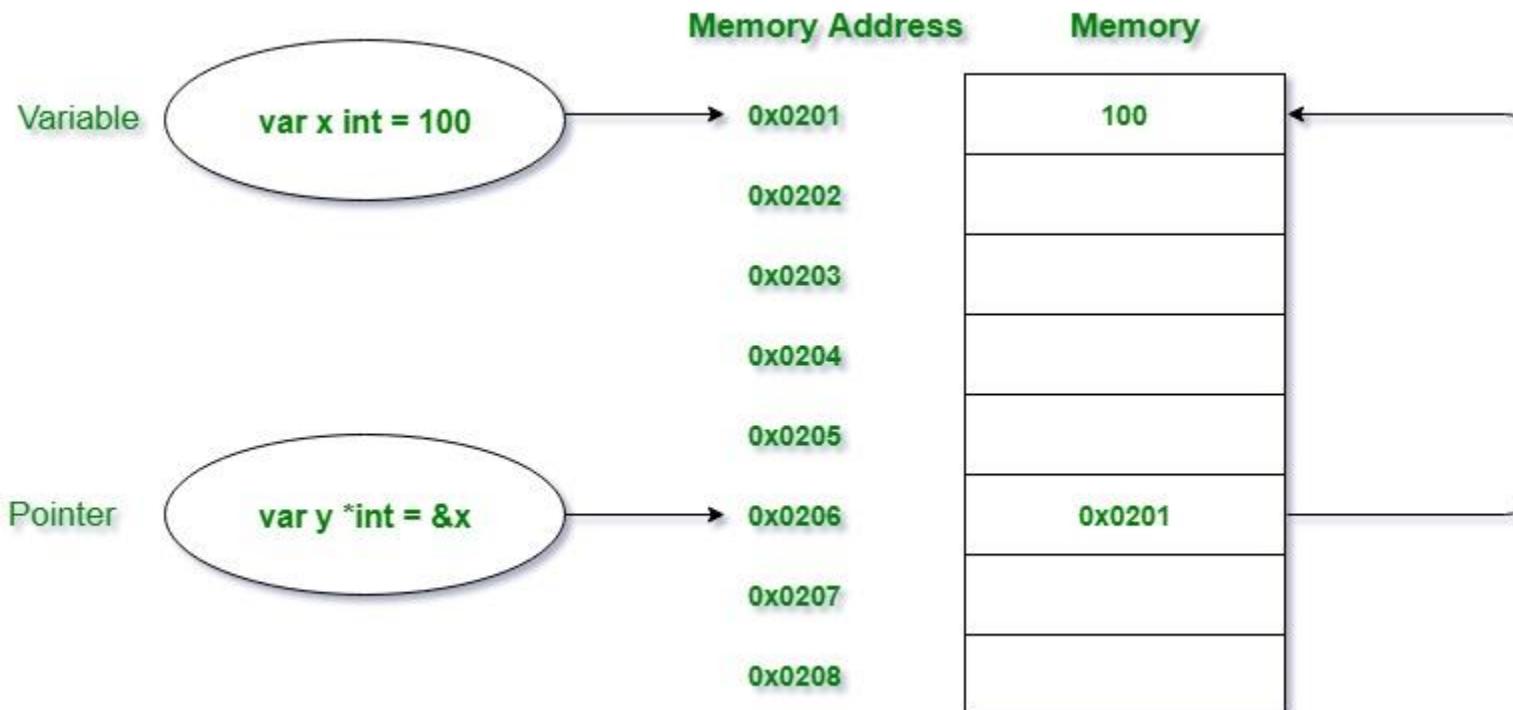


What is the need of the pointers?

- Variables are the names given to a memory location where the actual data is stored.
- To access the stored data we need the address of that particular memory location.
- To remember all the memory addresses(Hexadecimal Format) manually is an overhead that's why we use variables to store data and variables can be accessed just by using their name.
- Golang also allows saving a hexadecimal number into a variable using the literal expression i.e. number starting from 0x is a hexadecimal number.



What is the need of the pointers?





Pointers

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
}
```



Structures

- A structure or struct in Golang is a user-defined type that allows to group/combine items of possibly different types into a single type.
- Any real-world entity which has some set of properties/fields can be represented as a struct.
- This concept is generally compared with the classes in object-oriented programming.
- It can be termed as a lightweight class that does not support inheritance but supports composition.
- For Example, an address has a name, street, city, state, Pincode. It makes sense to group these three properties into a single structure address



Declaring Structure

Declaring a structure:

```
type Address struct {  
    name string  
    street string  
    city string  
    state string  
    Pincode int  
}
```

Refer [OOPsDemo](#) and [CompositionDemo](#)



Inline Structure

```
package main

import "fmt"

func main() {
    c := struct {
        Name string
        Type string
    } {
        Name: "Sammy",
        Type: "Shark",
    }
    fmt.Println(c.Name, "the", c.Type)
}
```



Methods in GO

```
func(receiver_name Type) method_name(parameter_list)(return_type){  
    // Code  
}
```

```
type Creature struct {  
    Name      string  
    Greeting string  
}
```

```
func (c Creature) Greet() {  
    fmt.Printf("%s says %s", c.Name, c.Greeting)  
}
```



Method vs Function

Method	Function
It contain receiver.	It does not contain receiver.
It can accept both pointer and value.	It cannot accept both pointer and value.
Methods of the same name but different types can be defined in the program.	Functions of the same name but different type are not allowed to define in the program.



Method with Pointer and Value

- Refer StructAuthorDemo



Map

- A map maps keys to values.
- The zero value of a map is nil. A nil map has no keys, nor can keys be added.
- The make function returns a map of the given type, initialized and ready for use.



Interface

- Go language interfaces are different from other languages.
- In Go language, the interface is a custom type that is used to specify a set of one or more method signatures and the interface is abstract, so you are not allowed to create an instance of the interface.
- But you are allowed to create a variable of an interface type and this variable can be assigned with a concrete type value that has the methods the interface requires.
- Interface is a collection of methods as well as it is a custom type.



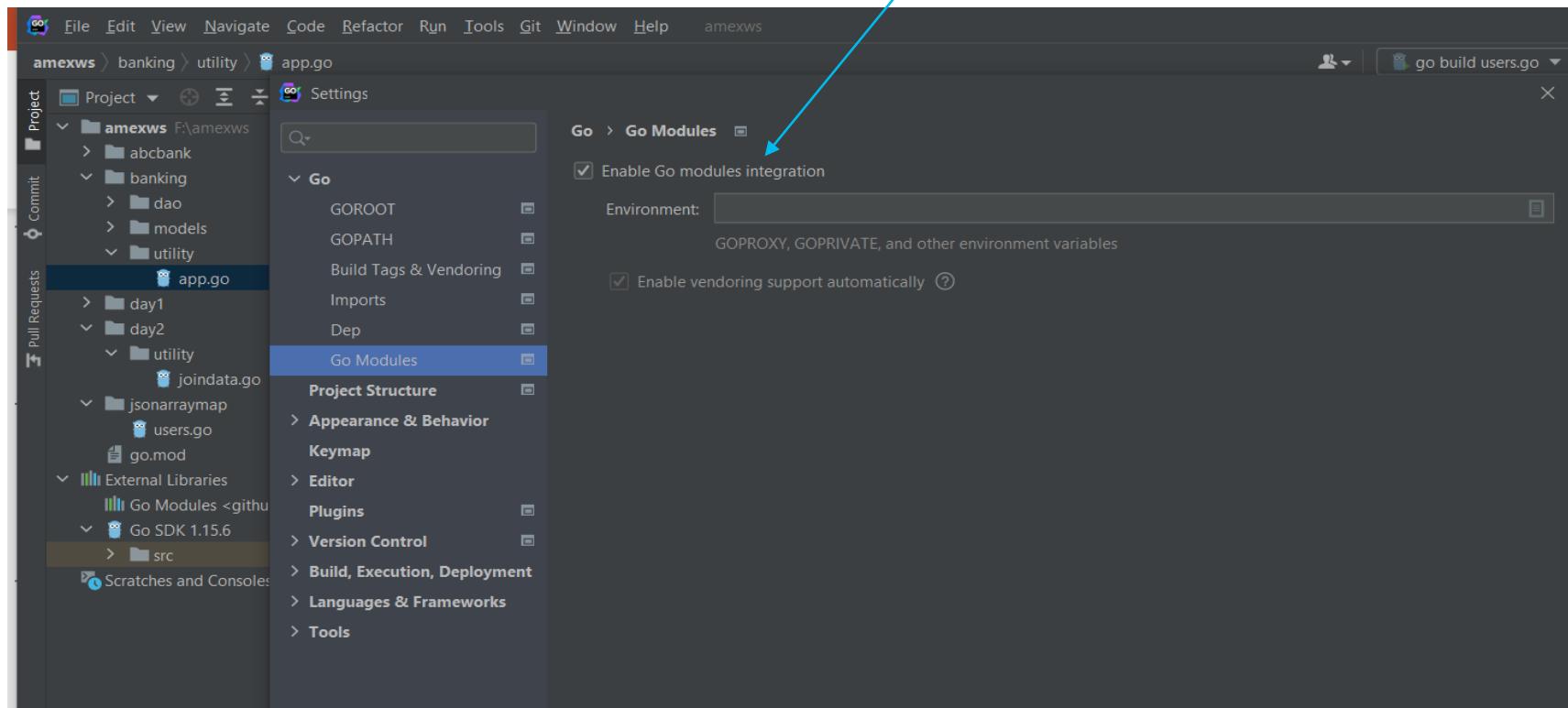
Interface

```
type Modify interface {  
    changeName(name *string)  
}  
  
func receiveInterface(m Modify){  
    var name string ="Parameswari"  
    var ptrname *string=&name  
    m.changeName(ptrname)
```



Go mod

- Go mod init github.com/amexws/abcbank
- Set File settings Go mod preferences enable integration





How To Define and Call Functions in Go

- A function is defined by using the `func` keyword. This is then followed by a name of your choosing and a set of parentheses that hold any parameters the function will take (they can be empty).
- The lines of function code are enclosed in curly brackets `{}`.
- In this case, we'll define a function named `hello()`:
- **`func hello() {}`**
- This sets up the initial statement for creating a function.
- From here, we'll add a second line to provide the instructions for what the function does. In this case, we'll be printing Hello, World! to the console:
- Our function is now fully defined, but if we run the program at this point, nothing will happen since we didn't call the function.
- So, inside of our `main()` function block, let's call the function with `hello()`:



How To Define and Call Functions in Go

```
import (
    "fmt"
    "strings"
)

func main() {
    names()
}

func names() {
    fmt.Println("Enter your name:")

    var name string
    fmt.Scanln(&name)
    // Check whether name has a vowel
    for _, v := range strings.ToLower(name) {
        if v == 'a' || v == 'e' || v == 'i' || v == 'o' || v == 'u' {
            fmt.Println("Your name contains a vowel.")
            return
        }
    }
    fmt.Println("Your name does not contain a vowel.")
}
```



How To Define and Call Functions in Go

repeat.go

```
package main

import "fmt"

func main() {
    repeat("Sammy", 5)
}

func repeat(word string, reps int) {
    for i := 0; i < reps; i++ {
        fmt.Println(word)
    }
}
```



How To Define and Call Functions in Go

double.go

```
package main

import "fmt"

func main() {
    result := double(3)
    fmt.Println(result)
}

func double(x int) int {
    y := x * 2
    return y
}
```

We can run the program and see the output:

```
go run double.go
```



How To Define and Call Functions in Go

```
repeat.go

package main

import "fmt"

func main() {
    val, err := repeat("Sammy", -1)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(val)
}

func repeat(word string, reps int) (string, error) {
    if reps <= 0 {
        return "", fmt.Errorf("invalid value of %d provided for reps. value must be greater than 0", reps)
    }
    var value string
    for i := 0; i < reps; i++ {
        value = value + word
    }
    return value, nil
}
```



Variadic Function

hello.go

```
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```



Variadic Function

menu.go

```
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    if len(names) == 0 {
        fmt.Println("nobody to greet")
        return
    }
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```



Variadic Function with ordered Arguments

join.go

```
package main

import "fmt"

func main() {
    var line string

    names := []string{"Sammy", "Jessica", "Drew", "Jamie"}

    line = join(",", names)
    fmt.Println(line)
}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```



Recursive Function

- Recursion is the process of repeating items in a self-similar way.
- The same concept applies in programming languages as well.
- If a program allows to call a function inside the same function, then it is called a recursive function call.



Anonymous Function

- Go language provides a special feature known as an anonymous function.
- An anonymous function is a function which doesn't contain any name.
- It is useful when you want to create an inline function.
- In Go language, an anonymous function can form a closure.



Understanding init in Go

- In Go, the predefined `init()` function sets off a piece of code to run before any other part of your package.
- This code will execute as soon as the package is imported.
- It can be used when you need your application to initialize in a specific state, such as when you have a specific configuration or set of resources with which your application needs to start.
- It is also used when importing a side effect, a technique used to set the state of a program by importing a specific package.
- This is often used to register one package with another to make sure that the program is considering the correct code for the task.
- Although `init()` is a useful tool, it can sometimes make code difficult to read, since a hard-to-find `init()` instance will greatly affect the order in which the code is run.
- Because of this, it is important for developers who are new to Go to understand the facets of this function, so that they can make sure to use `init()` in a legible manner when writing code.



Understanding init in Go

main.go

```
package main

import (
    "fmt"
    "time"
)

var weekday string

func init() {
    weekday = time.Now().Weekday().String()
}

func main() {
    fmt.Printf("Today is %s", weekday)
}
```



Handling Errors

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    err := errors.New("barnacles")
    fmt.Println("Sammy says:", err)
}
```

A blue arrow originates from the text "Create Error" and points directly at the line of code where an error is created: `err := errors.New("barnacles")`.



Dynamic Error Message

```
package main

import (
    "fmt"
    "time"
)

func main() {
    err := fmt.Errorf("error occurred at: %v", time.Now())
    fmt.Println("An error happened:", err)
}
```

Output

An error happened: Error occurred at: 2019-07-11
16:52:42.532621 -0400 EDT m=+0.000137103



Error Nil

```
package main

import (
    "errors"
    "fmt"
)

func boom() error {
    return errors.New("barnacles")
}

func main() {
    err := boom()

    if err != nil {
        fmt.Println("An error occurred:", err)
        return
    }

    fmt.Println("Anchors away!")
}
```



Error Along Value

```
package main

import (
    "errors"
    "fmt"
    "strings"
)

func capitalize(name string) (string, error) {
    if name == "" {
        return "", errors.New("no name provided")
    }
    return strings.ToTitle(name), nil
}
```



Error Along Value

```
func main() {
    name, err := capitalize("sammy")
    if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
    }
    fmt.Println("Capitalized name:", name)
}
```



Handling Panics in Go

- Errors that a program encounters fall into two broad categories: those the programmer has anticipated and those the programmer has not.
- The error largely deal with errors that we expect as we are writing Go programs.
- The error interface even allows us to acknowledge the rare possibility of an error occurring from function calls, so we can respond appropriately in those situations.



Handling Panics in Go

- Panics fall into the second category of errors, which are unanticipated by the programmer.
- These unforeseen errors lead a program to spontaneously terminate and exit the running Go program.
- Common mistakes are often responsible for creating panics.



Handling Panics in Go

- 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, and 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.
- Since panics include detail that is useful for resolving an issue, developers commonly use panics as an indication that they have made a mistake during a program's development.



Handling Panics in Go

- **Out of Bounds Panics**
- When you attempt to access an index beyond the length of a slice or the capacity of an array, the Go runtime will generate a panic.

```
func main() {  
    names := []string{  
        "lobster",  
        "sea urchin",  
        "sea cucumber",  
    }  
    fmt.Println("My favorite sea creature is:", names[len(names)])  
}
```





Nil Receivers

- The Go programming language has pointers to refer to a specific instance of some type existing in the computer's memory at runtime.
- Pointers can assume the value nil indicating that they are not pointing at anything.
- When we attempt to call methods on a pointer that is nil, the Go runtime will generate a panic.

```
func main() {
    s := &Shark{"Sammy"}
    s = nil
    s.SayHello()
}
```



Deferred Functions

- Your program may have resources that it must clean up properly, even while a panic is being processed by the runtime.
- Go allows you to defer the execution of a function call until its calling function has completed execution.
- Deferred functions run even in the presence of a panic, and are used as a safety mechanism to guard against the chaotic nature of panics.
- Functions are deferred by calling them as usual, then prefixing the entire statement with the `defer` keyword, as in `defer sayHello()`.



Deferred Function

Syntax:

```
// Function
defer func func_name(parameter_list Type) return_type{
// Code
}
```

```
// Method
defer func (receiver Type) method_name(parameter_list){
// Code
}
```

```
defer func (parameter_list)(return_type){
// code
}()
```



File Handling

- Intro
 - Everything is a File
- Basic Operations
 - Create Empty File
 - Truncate a File
 - Get File Info
 - Rename and Move a File
 - Delete Files
 - Open and Close Files
 - Check if File Exists
 - Check Read and Write Permissions
 - Change Permissions, Ownership, and Timestamps
 - Create Hard Links and Symlinks



File Handling

- Reading and Writing
 - Copy a File
 - Seek Positions in File
 - Write Bytes to a File
 - Quick Write to File
 - Use Buffered Writer
 - Read up to n Bytes from File
 - Read Exactly n Bytes
 - Read At Least n Bytes
 - Read All Bytes of File



File Handling

- Quick Read Whole File to Memory
 - Use Buffered Reader
 - Read with a Scanner
 - Archiving(Zipping)
 - Archive(Zip) Files
 - Extract(Unzip) Archived Files
 - Compressing
 - Compress a File
 - Uncompress a File
- Misc
 - Temporary Files and Directories
 - Downloading a File Over HTTP
 - Hashing and Checksums



Embedding interfaces in Go

- In Go language, the interface is a collection of method signatures and it is also a type means you can create a variable of an interface type.
- As Go language does not support inheritance, but the Go interface fully supports embedding.
- In embedding, an interface can embed other interfaces or an interface can embed other interface's method signatures in it.
- Any number of interfaces can be embedded in a single interface.
- Day3/Embeddingdemo.go



Bit Vector Type

- A bit vector is an array data structure that compactly stores bits.
- This library is based on 5 static different data structures:
- 8-bit vector: relies on an internal uint8
- 16-bit vector: relies on an internal uint16
- 32-bit vector: relies on an internal uint32
- 64-bit vector: relies on an internal uint64
- 128-bit vector: relies on two internal uint64 (for ASCII problems)
- The rationale of using a static integer compared to a dynamic []byte is first of all to save memory. There is no structure and/or slice overhead.



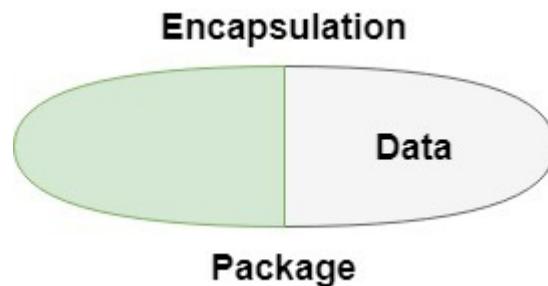
Bit Vector Type

- import "github.com/dropbox/godropbox/container/bitvector"
- Package bitvector provides the implementation of a variable sized compact vector of bits which supports lookups, sets, appends, insertions, and deletions.
- type BitVector
 - func NewBitVector(data []byte, length int) *BitVector
 - func (vector *BitVector) Append(bit byte)
 - func (vector *BitVector) Bytes() []byte
 - func (vector *BitVector) Delete(index int)
 - func (vector *BitVector) Element(i int) byte
 - func (vector *BitVector) Insert(bit byte, index int)
 - func (vector *BitVector) Length() int
 - func (vector *BitVector) Set(bit byte, index int)



Encapsulation in Golang

- Encapsulation is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.





Encapsulation in Golang

- Go language, encapsulation is achieved by using packages.
- Go provides two different types of identifiers, i.e. exported and unexported identifiers.
- Encapsulation is achieved by exported elements(variables, functions, methods, fields, structures) from the packages, it helps to control the visibility of the elements(variables, functions, methods, fields, structures).
- The elements are visible if the package in which they are defined is available in your program.



Exported Identifiers

- Exported identifiers are those identifiers which are exported from the package in which they are defined.
- The first letter of these identifiers is always in capital letter.
- This capital letter indicates that the given identifier is exported identifier.
- Exported identifiers are always limited to the package in which they are defined.
- When you export the specified identifier from the package you simple just export the name not the implementation of that identifier.
- This mechanism is also applicable for function, fields, methods, and structures.
- **// Exported Method**
- **res := strings.ToUpper(slc[x])**





Unexported Identifiers

- Unexported identifiers are those identifiers which are not exported from any package.
- They are always in lowercase.
- **// Unexported function**

```
func addition(val ...int) int {  
    s := 0  
    for x := range val {  
        s += val[x]  
    }  
    fmt.Println("Total Sum: ", s)  
    return s  
}
```



What Are Interface Types?

- An interface type specifies a collection of method prototypes.
- In other words, each interface type defines a method set.
- In fact, we can view an interface type as a method set. For any of the method prototype specified in an interface type, its name can't be the blank identifier `_`.
- We also often say that each interface type specifies a behavior set (represented by the method set specified by that interface type).



Interface Satisfaction

- A type satisfies an interface if it possesses all the methods the interface requires.
- For example, an `*os.File` satisfies `io.Reader`, `Writer`, `Closer`, and `ReadWriter`.
- A `*bytes.Buffer` satisfies `Reader`, `Writer`, and `ReadWriter`, but does not satisfy `Closer` because it does not have a `Close` method.
- As a shorthand, Go programmers often say that a concrete type “is a” particular interface type, meaning that it satisfies the interface.
- For example, a `*bytes.Buffer` is an `io.Writer`; an `*os.File` is an `io.ReadWriter`.



Parsing flags by flag value

- Using the flag package involves three steps:
- Define variables to capture flag values.
- Define the flags your Go application will use
- Finally, parse the flags provided to the application upon execution.
- Most of the functions within the flag package are concerned with defining flags and binding them to variables that you have defined.
- The parsing phase is handled by the Parse() function.
- `go run colortext.go -color`

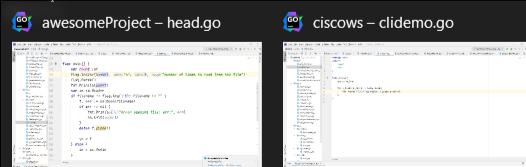
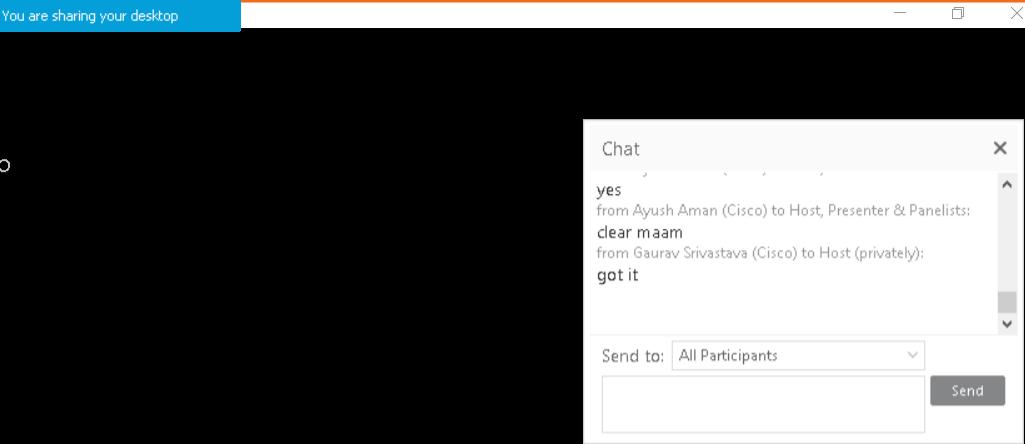


Parsing flags by flag value

```
c:\ Administrator: Command Prompt
import (
    "bufio"
    "flag"
)
F:\go\src\awesomeProject\Day3>go run head.go --n 25 -- head.go
25
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
//var fileName *string
//var count *int

func main() {
    var count int
    flag.IntVar(&count, "n", 5, "number of lines to read from the file")
    flag.Parse()
    fmt.Println(count)
    var in io.Reader
    if filename := flag.Arg(0); filename != "" {
        f, err := os.Open(filename)
        if err != nil {
            fmt.Println("error opening file: err:", err)
            os.Exit(1)
        }
        defer f.Close()
    }
}
F:\go\src\awesomeProject\Day3>
```





Parsing flags by flag value

```
F:\go\src\awesomeProject\Day3>go run head.go -- head.go
package main

import (
    "bufio"
    "flag"

F:\go\src\awesomeProject\Day3>go run flagstringint.go
env: development
port: 3000

F:\go\src\awesomeProject\Day3>go run flagstringint.go --port 8000
env: development
port: 8000

F:\go\src\awesomeProject\Day3>
```



The 3 ways to sort in Go

- Slice of ints, float64s or strings
- Custom comparator
- Custom data structures
- Bonus: Sort a map by key or value
- Performance and implementation



Assertions

- Package assert provides a set of comprehensive testing tools for use with the normal Go testing system.
- Assertions allow you to easily write test code, and are global funcs in the `assert` package.
- All assertion functions take, as the first argument, the `*testing.T` object provided by the testing framework.
- This allows the assertion funcs to write the failings and other details to the correct place.
- Every assertion function also takes an optional string message as the final argument, allowing custom error messages to be appended to the message the assertion method outputs.



Type Assertions

- Type assertions in Golang provide access to the exact type of variable of an interface.
- If already the data type is present in the interface, then it will retrieve the actual data type value held by the interface.
- A type assertion takes an interface value and extracts from it a value of the specified explicit type.
- Basically, it is used to remove the ambiguity from the interface variables.



Type Assertions

- Syntax:
- `t := value.(typeName)`
- where `value` is a variable whose type must be an interface.
- `typeName` is the concrete type we want to check and underlying `typeName` value is assigned to variable.



Type Switch

- type switch which makes use of type assertion to determine the type of variable, and do the operations accordingly.

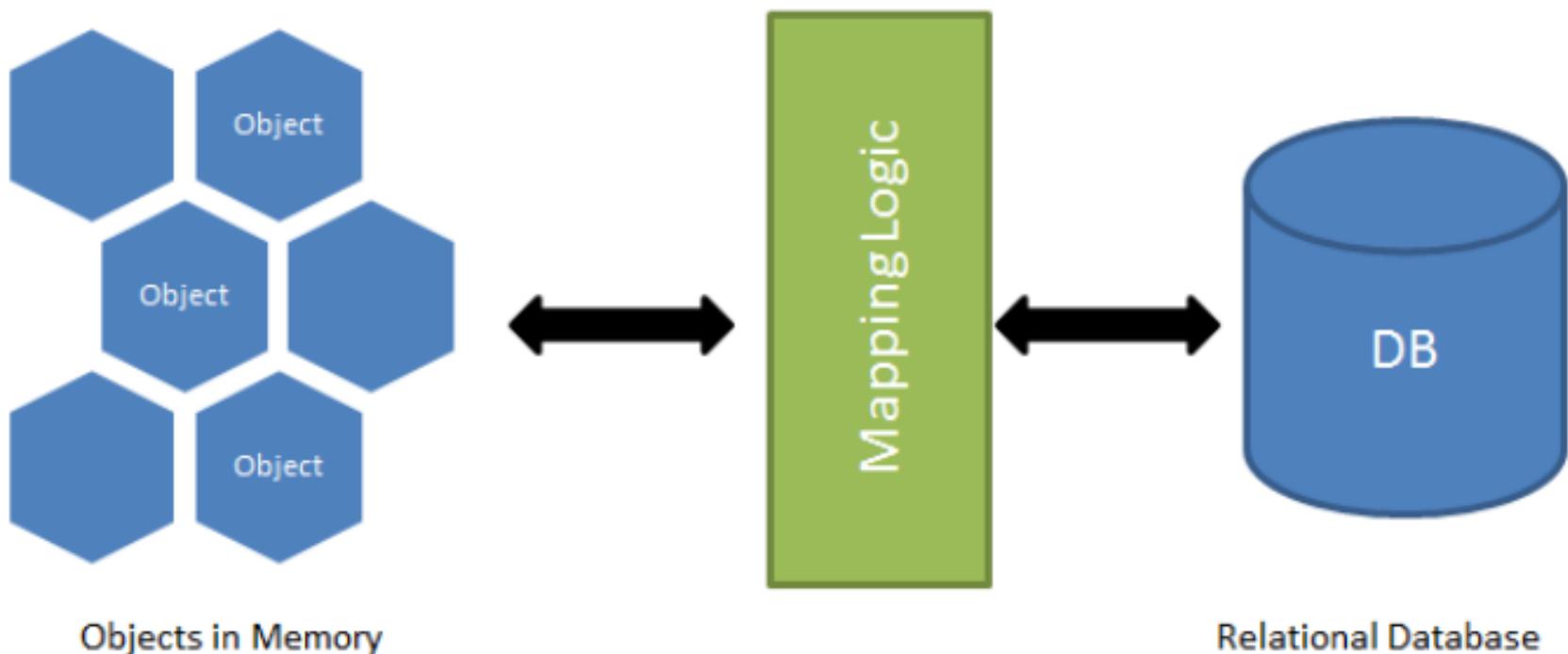
The background of the slide is a soft-focus photograph of a computer setup. It shows the edge of a white monitor on the right and a dark computer keyboard in the lower right foreground. The overall color palette is cool and muted.

Go with back end mysql



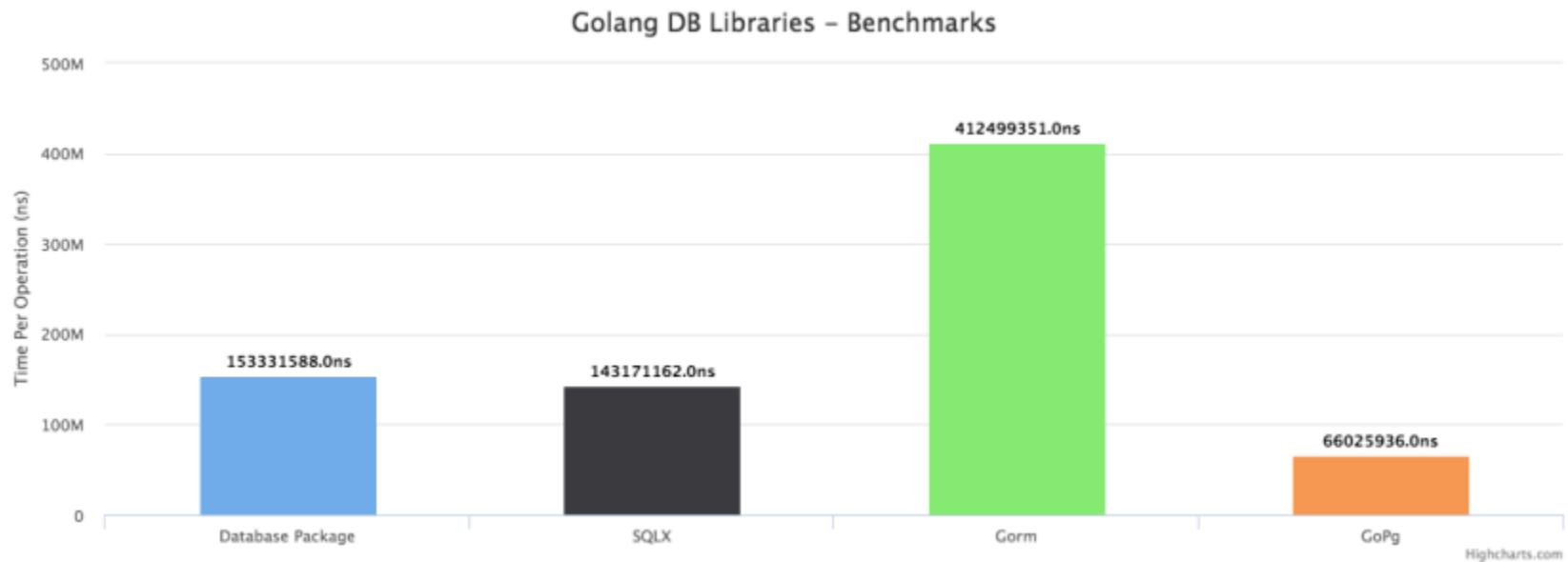
Go ORM

O/R Mapping





Go ORM





Regular Expressions

Choice and grouping

Regexp	Meaning
xy	x followed by y
$x y$	x or y, prefer x
$xy z$	same as $(xy) z$
xy^*	same as $x(y^*)$

Repetition (greedy and non-greedy)

Regexp	Meaning
x^*	zero or more x, prefer more
$x^?$	prefer fewer (non-greedy)
x^+	one or more x, prefer more
$x^+?$	prefer fewer (non-greedy)
$x^?$	zero or one x, prefer one
$x^{??}$	prefer zero
$x^{\{n\}}$	exactly n x



Regular Expressions

Character classes

Expression	Meaning
.	any character
[ab]	the character a or b
[^ab]	any character except a or b
[a-z]	any character from a to z
[a-zA-Z0-9]	any character from a to z or 0 to 9
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\f\r]
\S	a non-whitespace character: [^\t\n\f\r]
\w	a word character: [0-9A-Za-z_]
\W	a non-word character: [^0-9A-Za-z_]
\p{Greek}	Unicode character class*
\pN	one-letter name
\P{Greek}	negated Unicode character class*
\PN	one-letter name



Regular Expressions

Special characters

To match a **special character** `\^$. | ?*+-[] {} ()` literally, escape it with a backslash. For example `\{` matches an opening brace symbol.

Other escape sequences are:

Symbol	Meaning
<code>\t</code>	horizontal tab = <code>\011</code>
<code>\n</code>	newline = <code>\012</code>
<code>\f</code>	form feed = <code>\014</code>
<code>\r</code>	carriage return = <code>\015</code>
<code>\v</code>	vertical tab = <code>\013</code>
<code>\123</code>	octal character code (up to three digits)
<code>\x7F</code>	hex character code (exactly two digits)



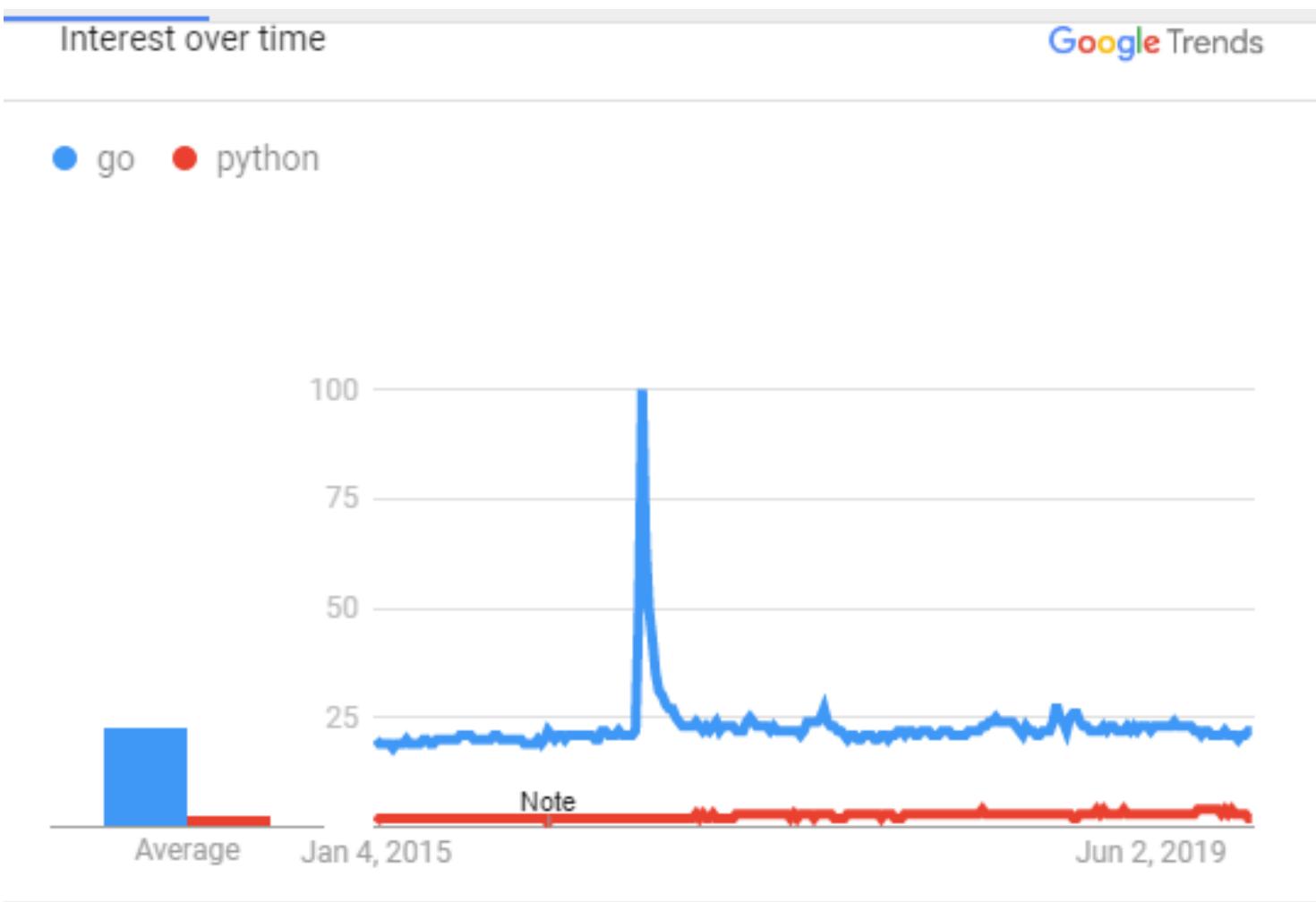
Regular Expressions

Text boundary anchors

Symbol	Matches
\A	at beginning of text
^	at beginning of text or line
\$	at end of text
\z	
\b	at ASCII word boundary
\B	not at ASCII word boundary

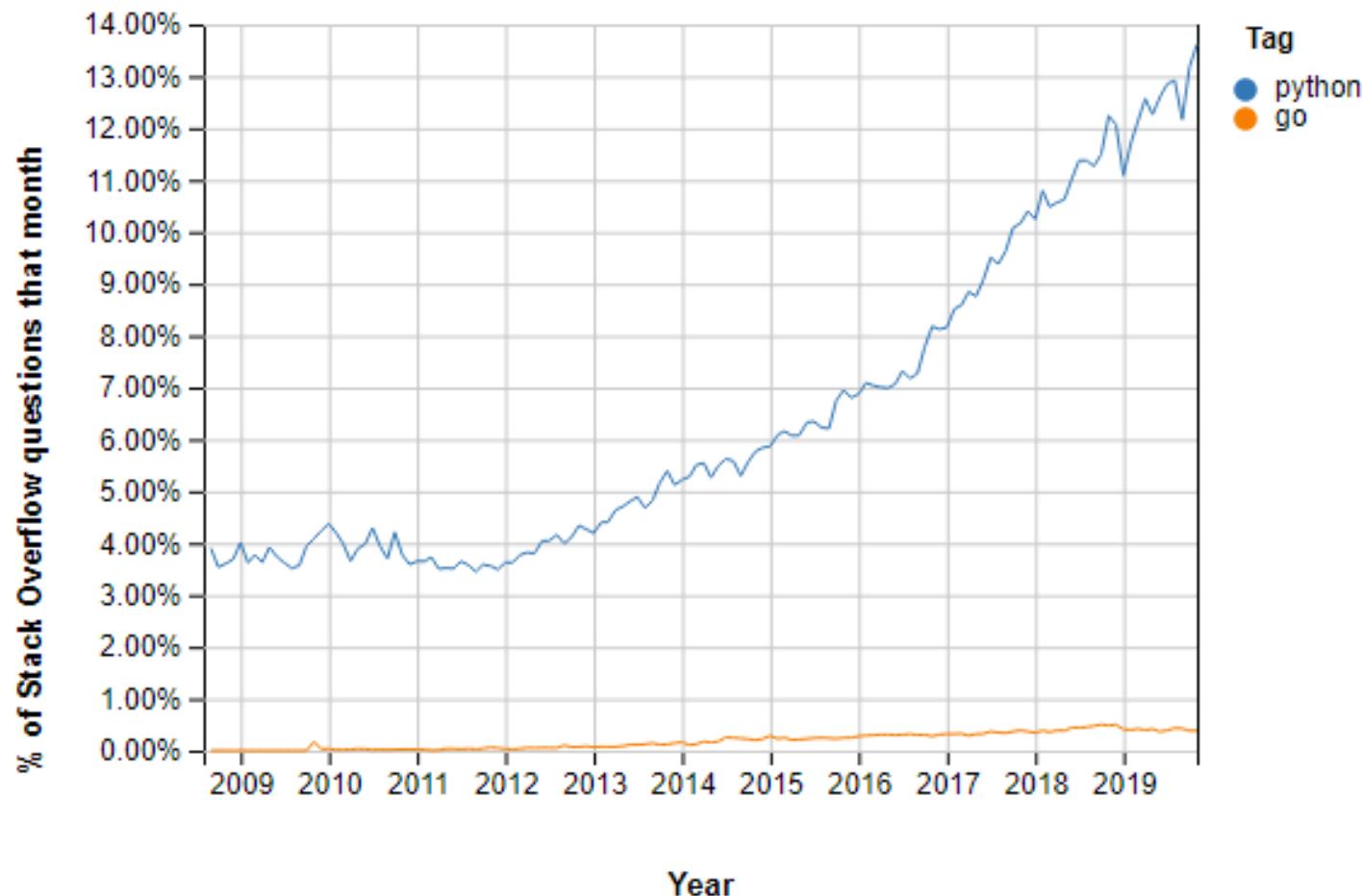


GO Community vs Python Community



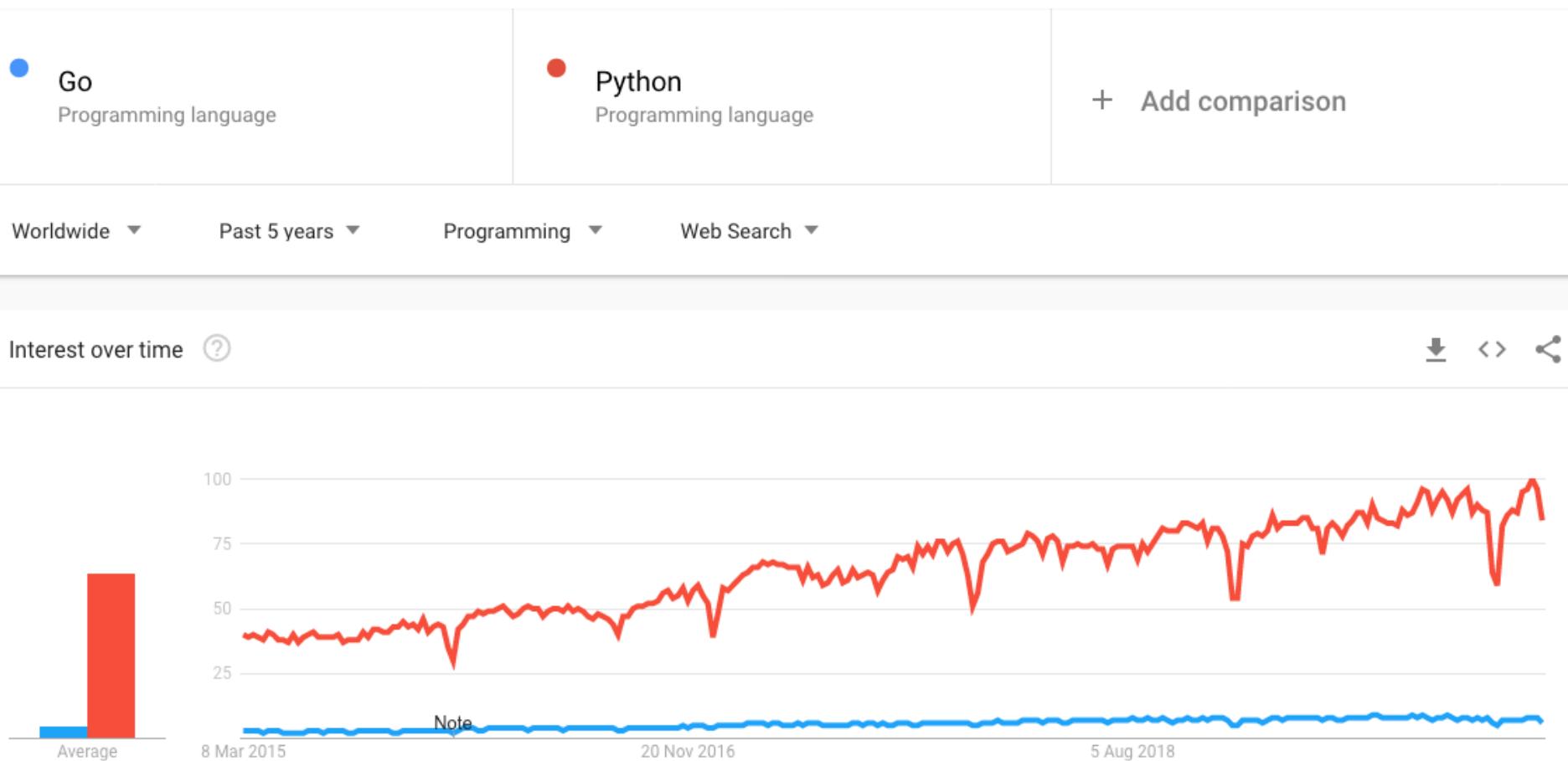


GO Community vs Python Community





GO Community vs Python Community





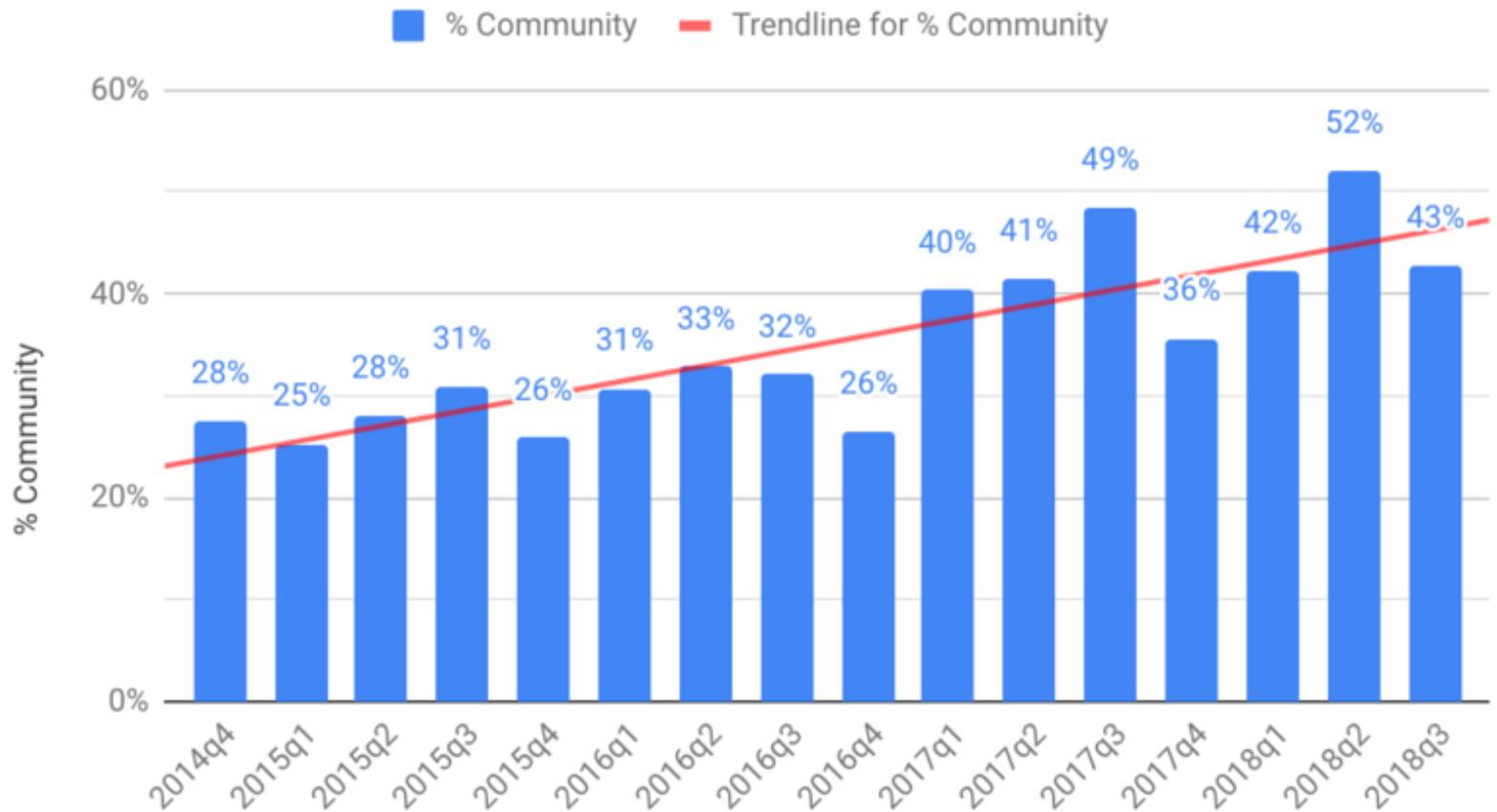
GO Community vs Python Community





GO Community vs Python Community

% of commits from the community



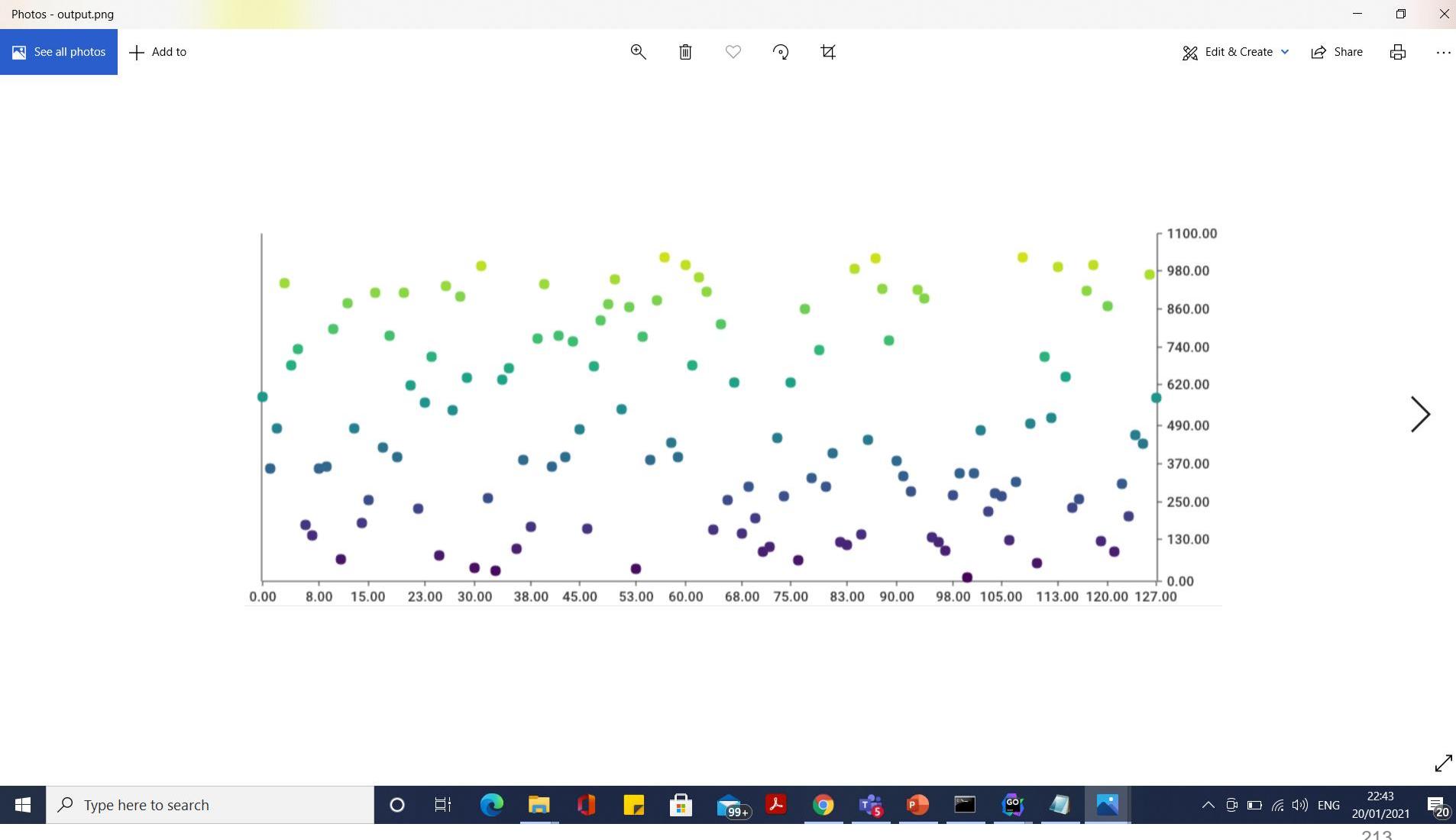


Developers can easily support Go apps

- Automatic documentation GoDoc
- Static code analysis GoMetaLinter is a meta tool
- Embedded testing environment
 - Go provides developers with a simple API that you can use for testing, profiling, and even adding your own code samples. You can easily start testing, run parallel tests, skip tests, and do much more.
- Race condition detection

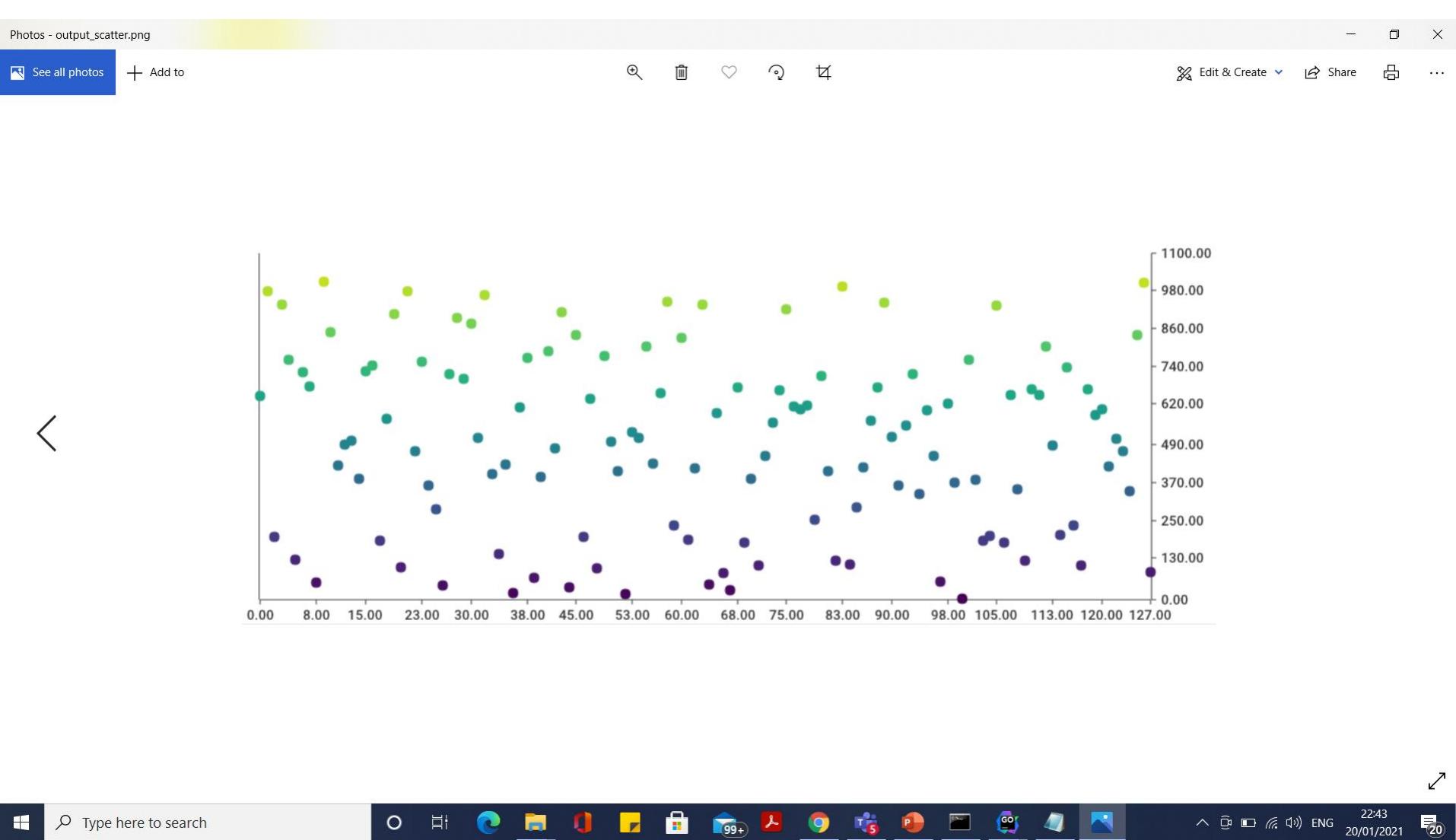


Go charts





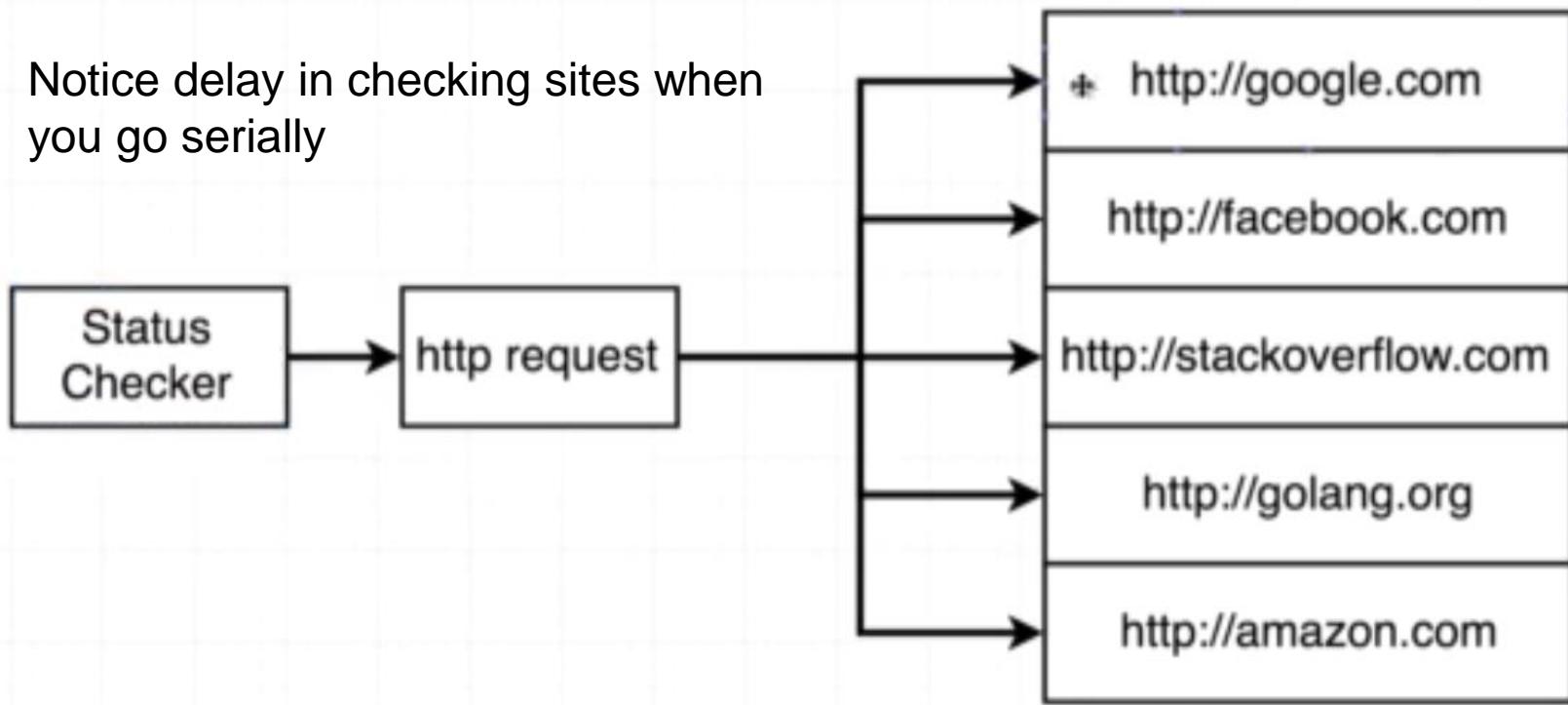
Go charts



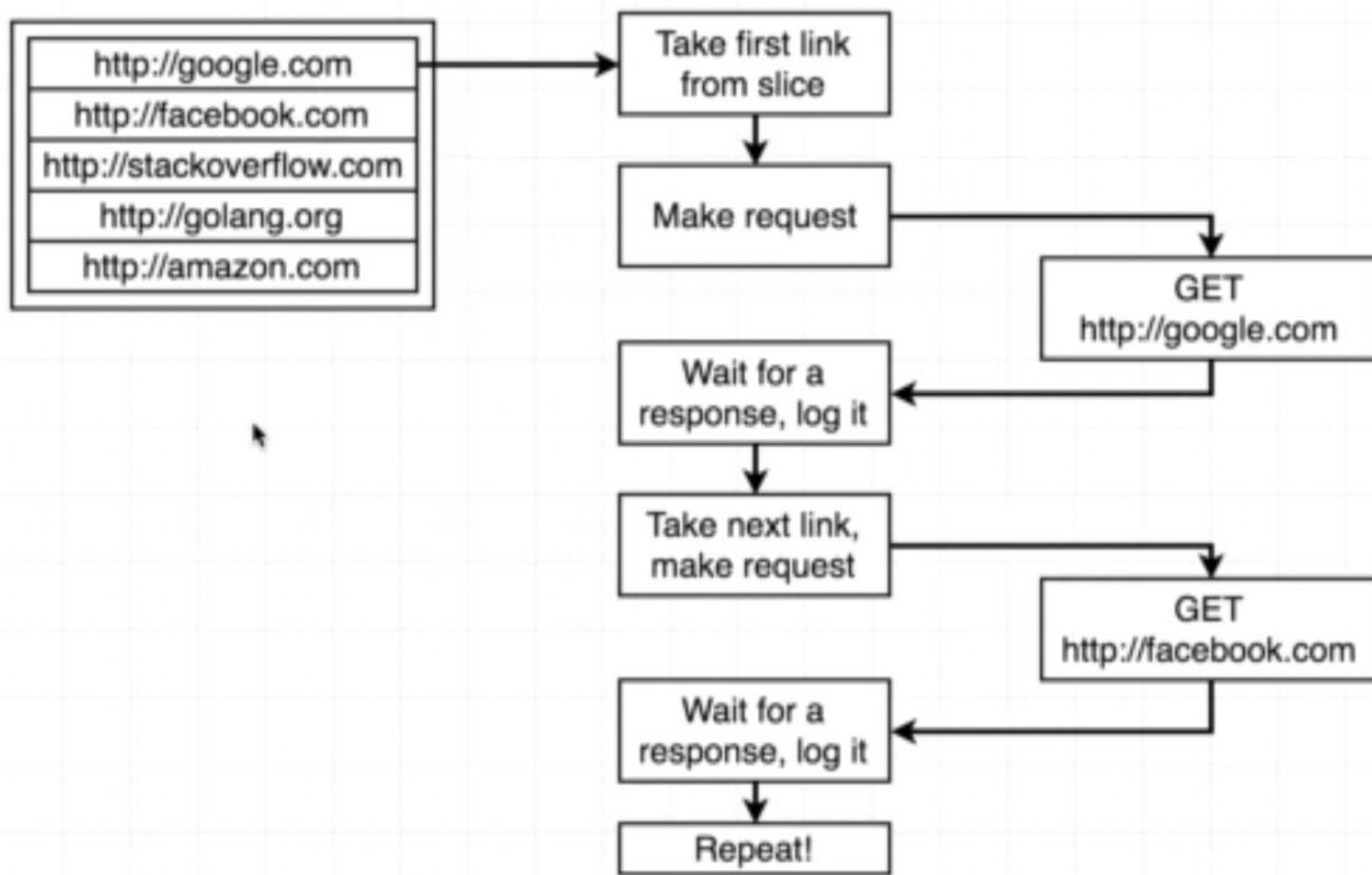
Website Status Checker

Serialchecker Ref

Notice delay in checking sites when you go serially

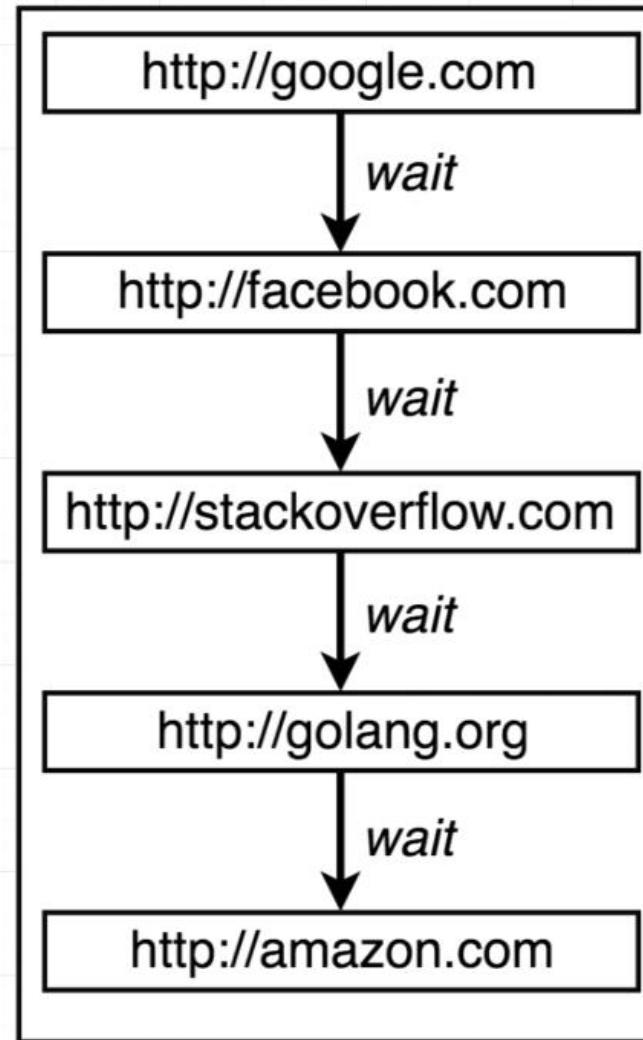


Website Status Checker



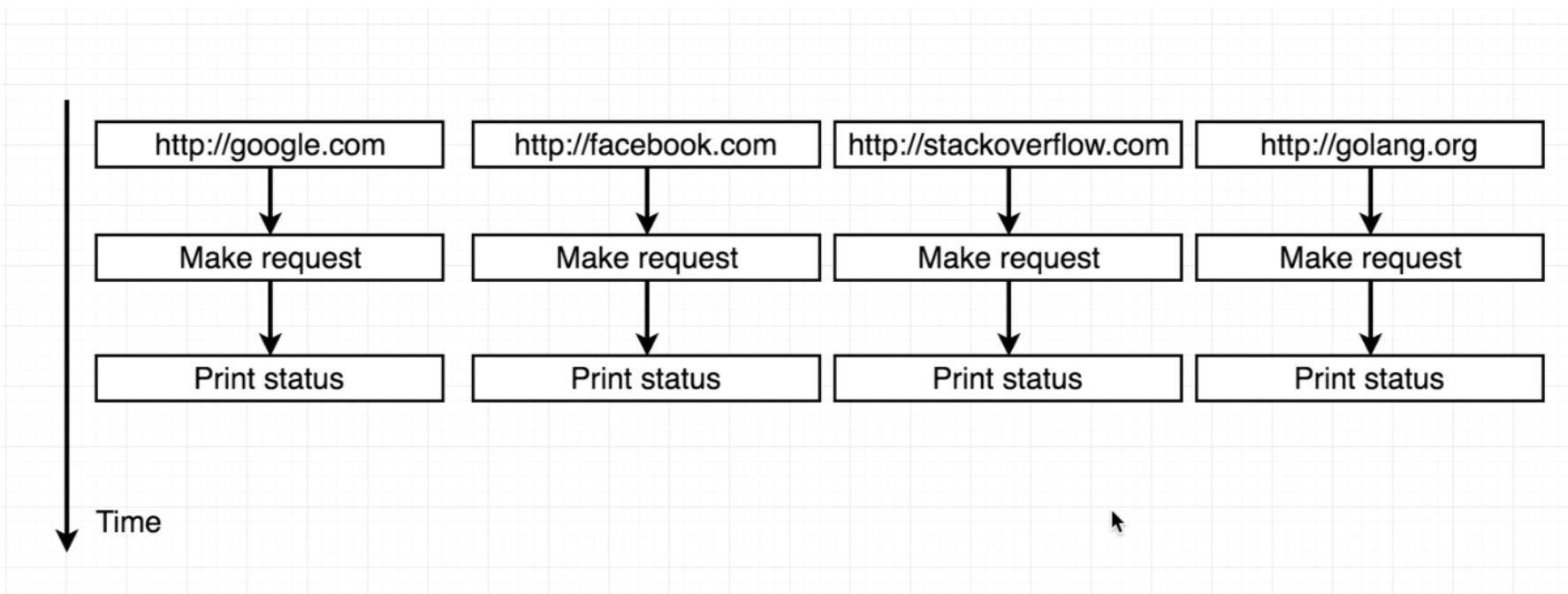


Website Status Checker



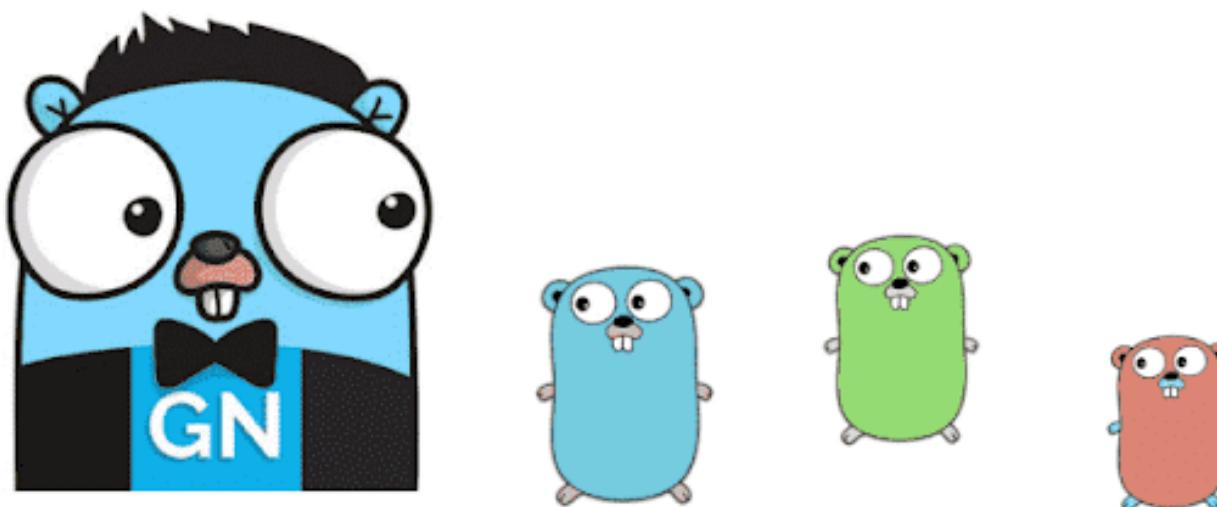


Go Routine Solution





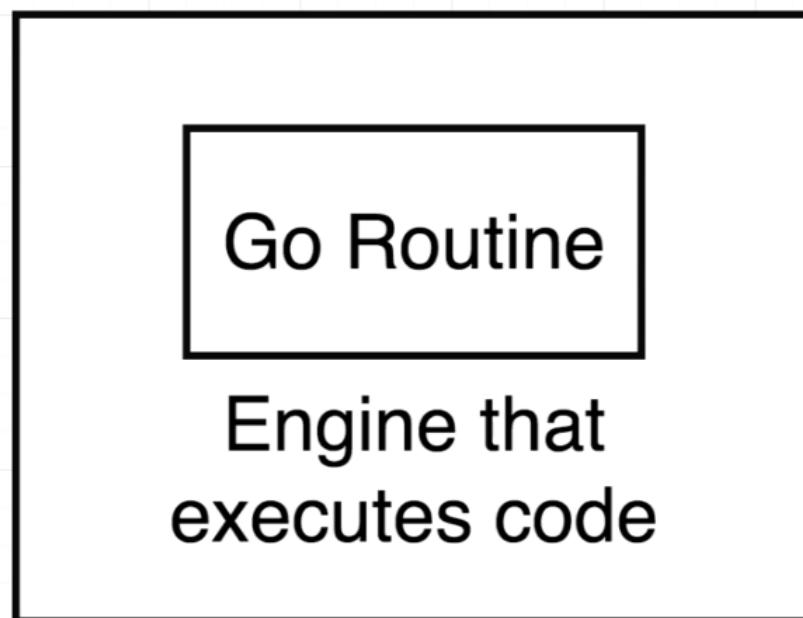
Go Routines





Go Routines

Our Running Program
(a process)





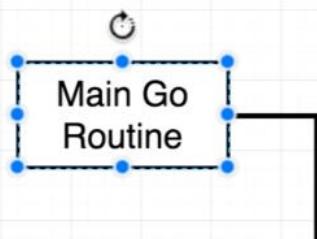
Go Routines

```
func main() {
    links:= []string{
        "http://google.com",
        "http://amazon.com",
    }

    for _, link:=range links {
        checkLink(link)
    }
}

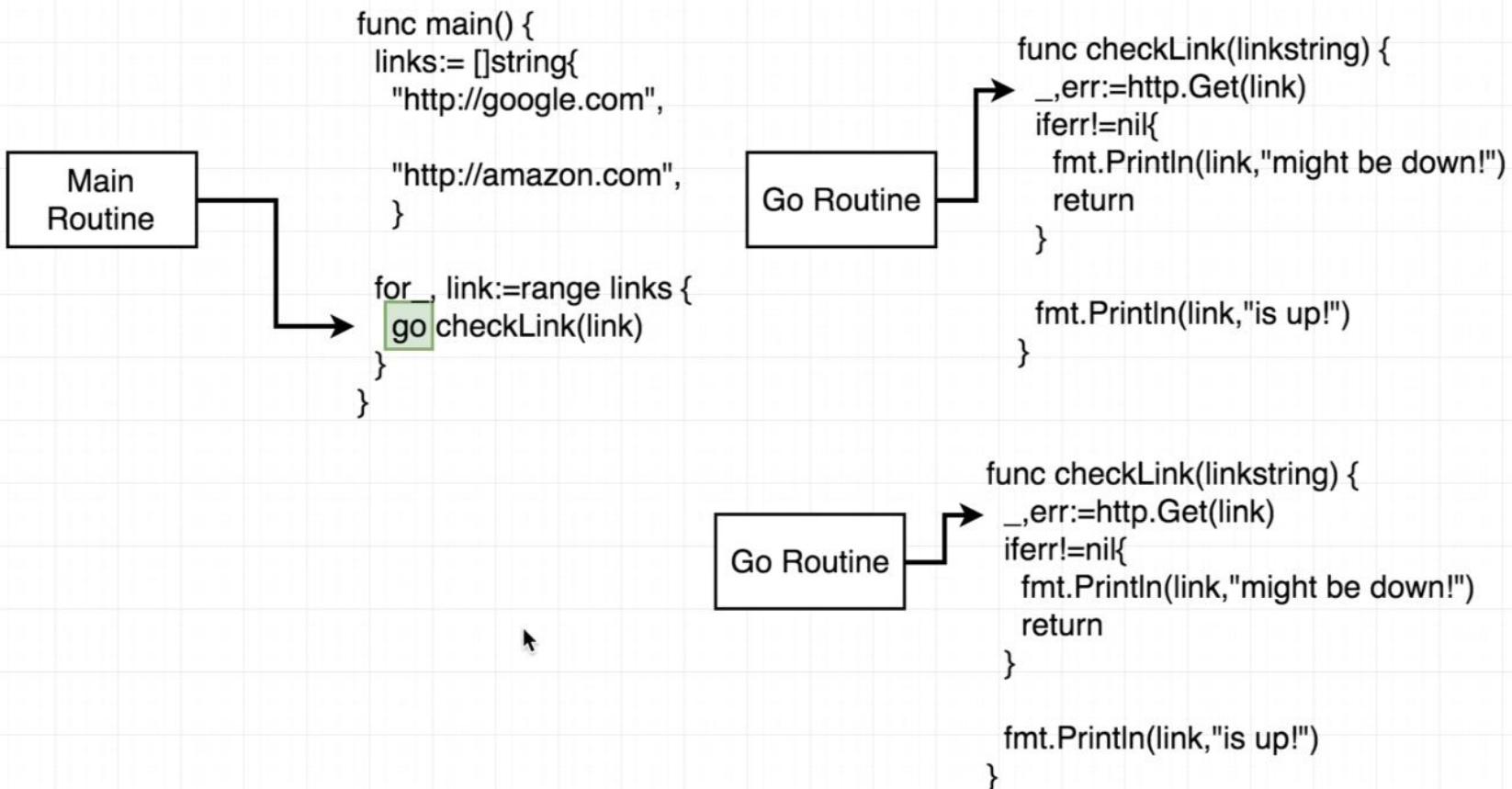
func checkLink(link string) {
    _, err:= http.Get(link) ← Blocking call!
    if err !=nil {
        fmt.Println(link, "might be down!")
        return
    }

    fmt.Println(link, "is up!")
}
```





Go Routines





Go Routines

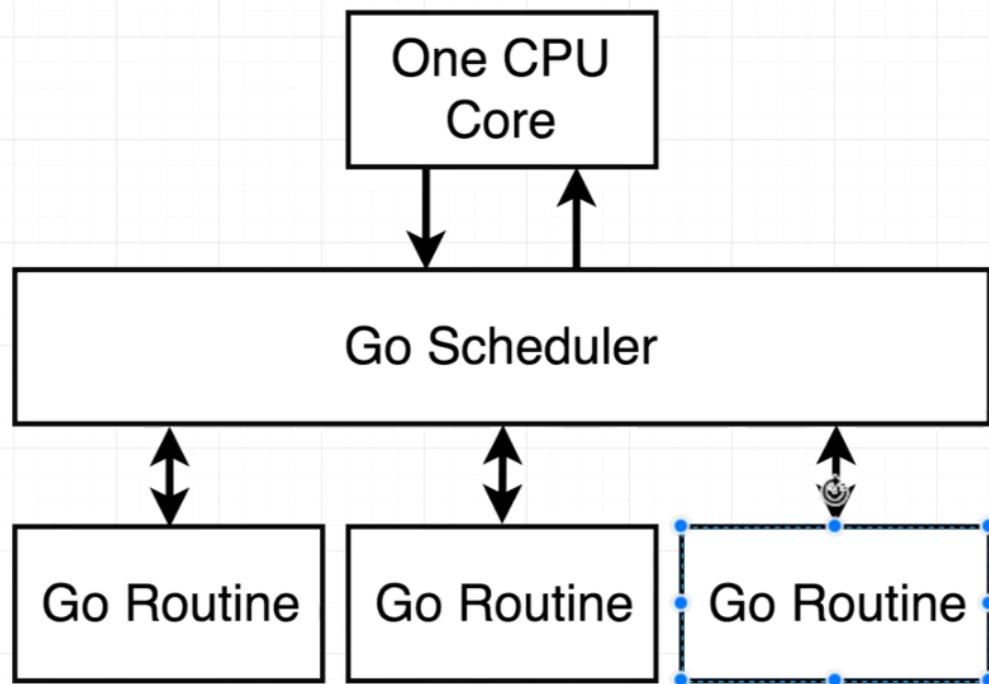


Create a new
thread go routine...

...And run this
function with it



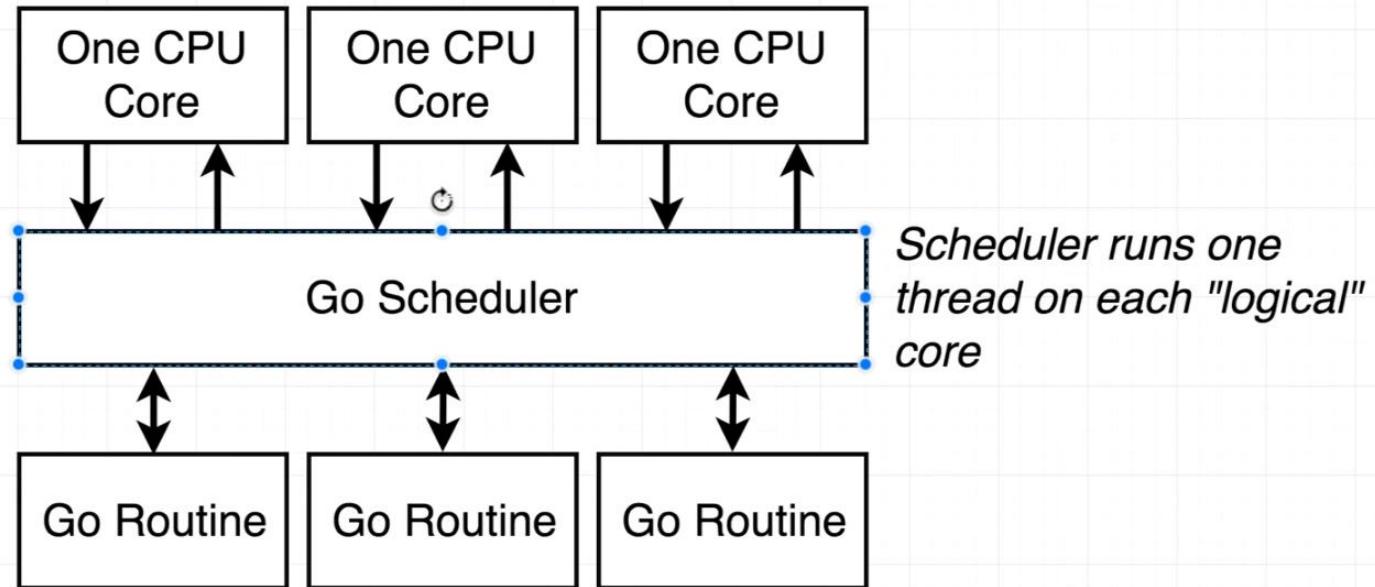
Go Routines



*Scheduler runs **one** routine until it finishes or makes a blocking call (like an HTTP request)*



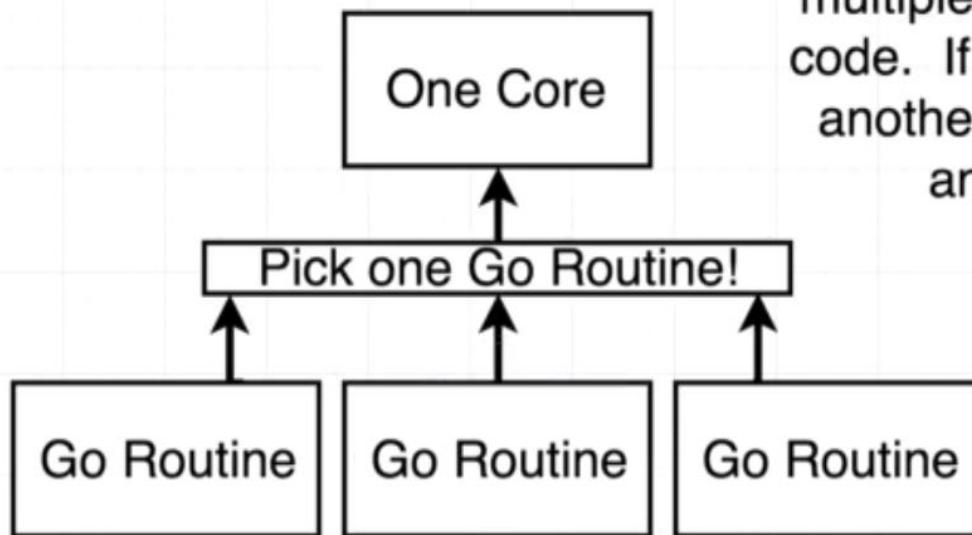
Go Routines



*By default Go tries
to use one core!*



Go Routines

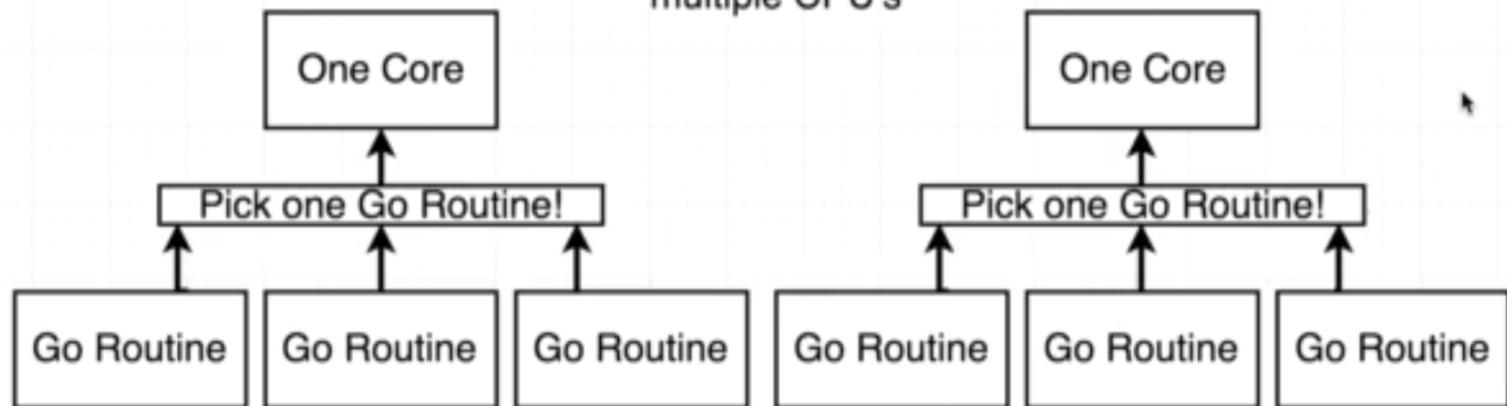


Concurrency - We can have multiple threads executing code. If one thread blocks, another one is picked up and worked on



Go Routines

Parallelism - Multiple threads executed at the exact same time. Requires multiple CPU's





Go Routines

Concurrency

Single core processor

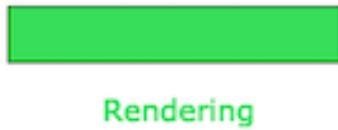


Parallelism

Core 1

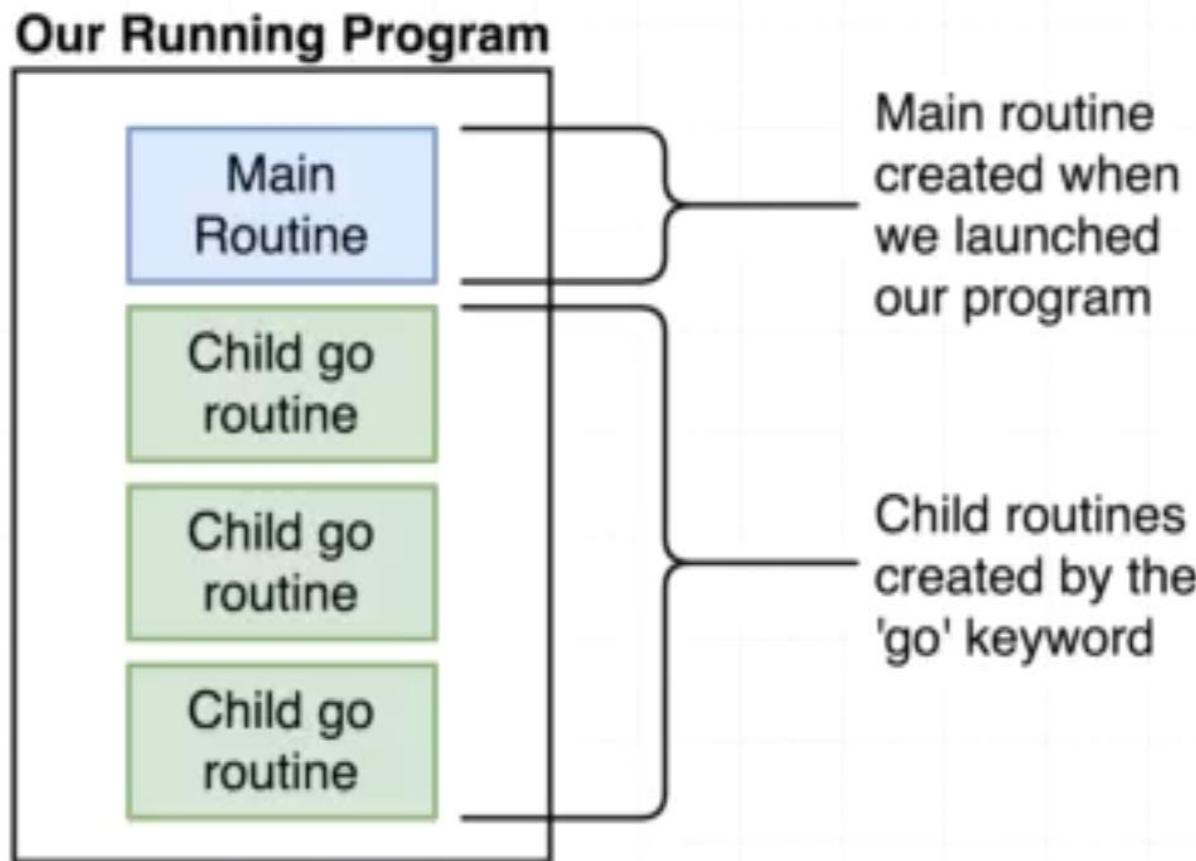


Core 2





Go Routines





Go Routines

- Goroutines are incredibly lightweight “threads” managed by the go runtime.
- They enable us to create asynchronous parallel programs that can execute some tasks far quicker than if they were written in a sequential manner.
- Goroutines are far smaller than threads, they typically take around 2kB of stack space to initialize compared to a thread which takes 1Mb
- Goroutines are typically multiplexed onto a very small number of OS threads which typically mean concurrent go programs require far less resources in order to provide the same level of performance as languages such as Java.



Go Routines

- Creating a thousand goroutines would typically require one or two OS threads at most, whereas if we were to do the same thing in java it would require 1,000 full threads each taking a minimum of 1Mb of Heap space.
- By mapping hundreds or thousands of goroutines onto a single thread we don't have to worry about the performance hit when creating and destroying threads in our application.
- It's incredibly in-expensive to create and destroy new goroutines due to their size and the efficient way that go handles them.



Goroutines and Threads - the differences

- Go uses goroutines while a language like Java uses threads.
- 3 factors - memory consumption, setup and teardown and switching time differentiates it from other languages.

Memory consumption



- The creation of a goroutine does not require much memory
 - only 2kB of stack space.
- They grow by allocating and freeing heap storage as required.
- Threads on the other hand start out at 1Mb (500 times more), along with a region of memory called a guard page that acts as a guard between one thread's memory and another.
- A server handling incoming requests can therefore create one goroutine per request without a problem, but one thread per request will eventually lead to the dreaded OutOfMemoryError.
- This isn't limited to Java - any language that uses OS threads as the primary means of concurrency will face this issue.

Setup and teardown costs



- Threads have significant setup and teardown costs because it has to request resources from the OS and return it once its done.
- The workaround to this problem is to maintain a pool of threads.
- In contrast, goroutines are created and destroyed by the runtime and those operations are pretty cheap.
- The language doesn't support manual management of goroutines.



Switching Costs

- When a thread blocks, another has to be scheduled in its place.
- Threads are scheduled preemptively, and during a thread switch, the scheduler needs to save/restore ALL registers, that is, 16 general purpose registers, PC (Program Counter), SP (Stack Pointer), segment registers, 16 XMM registers, FP coprocessor state, 16 AVX registers, all MSRs etc.
- This is quite significant when there is rapid switching between threads.
- Goroutines are scheduled cooperatively and when a switch occurs, only 3 registers need to be saved/restored - Program Counter, Stack Pointer and DX.
- The cost is much lower.
- The number of goroutines is generally much higher, but that doesn't make a difference to switching time for two reasons.
 - Only runnable goroutines are considered, blocked ones aren't.
 - Also, modern schedulers are O(1) complexity, meaning switching time is not affected by the number of choices (threads or goroutines).



How goroutines are executed

- The runtime manages the goroutines throughout from creation to scheduling to teardown.
- The runtime is allocated a few threads on which all the goroutines are multiplexed.
- At any point of time, each thread will be executing one goroutine.
- If that goroutine is blocked, then it will be swapped out for another goroutine that will execute on that thread instead



How goroutines are executed

- As the goroutines are scheduled cooperatively, a goroutine that loops continuously can starve other goroutines on the same thread.
- In Go 1.2, this problem is somewhat alleviated by occasionally invoking the Go scheduler when entering a function, so a loop that includes a non-inlined function call can be preempted.



Anonymous Goroutine Functions

- package main

```
import "fmt"
```

```
func main() {  
    // we make our anonymous function concurrent using `go`  
    go func() {  
        fmt.Println("Executing my Concurrent anonymous function")  
    }()  
    // we have to once again block until our anonymous goroutine  
    // has finished or our main() function will complete without  
    // printing anything  
    fmt.Scanln()  
}
```



Bug in Go Routines

No output after putting go

The screenshot shows a code editor and a terminal window within an IDE. The code editor displays `main.go` from the `goroutinebug` package. The terminal window below shows the output of the build command.

```
ciscogolangws > day3 > goroutinebug > main.go
Project  Proj.  channels\main.go  serialchecker\main.go  goroutinebug\main.go
ciscogolangws [ciscows] F:\cisc
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

for _, link := range links {
    go checkLink(link)
}

func checkLink(link string) {
    _, err := http.Get(link)
}

main()
```

Run: go build main.go (2) >

<4 go setup calls>

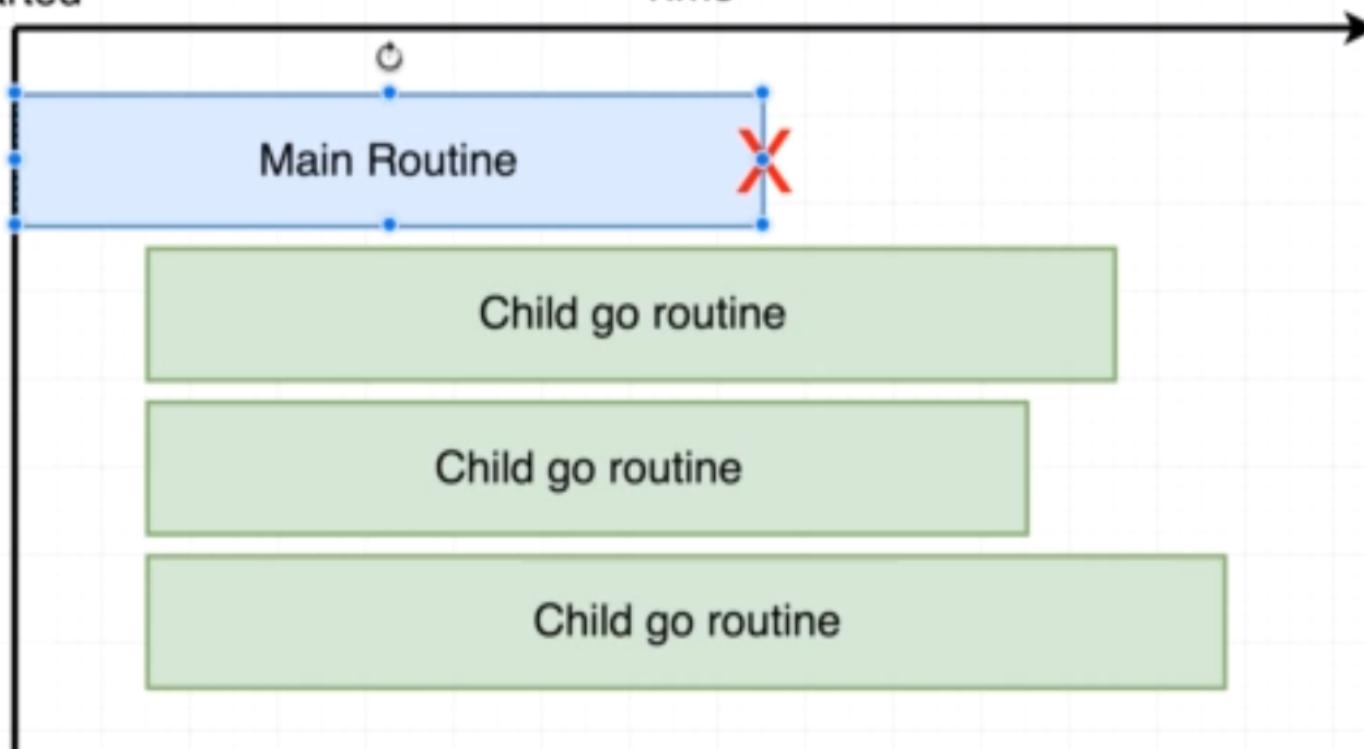
Process finished with the exit code 0



Bug in Go Routines

Program
started

Time





Waiting for Goroutines to Finish Execution

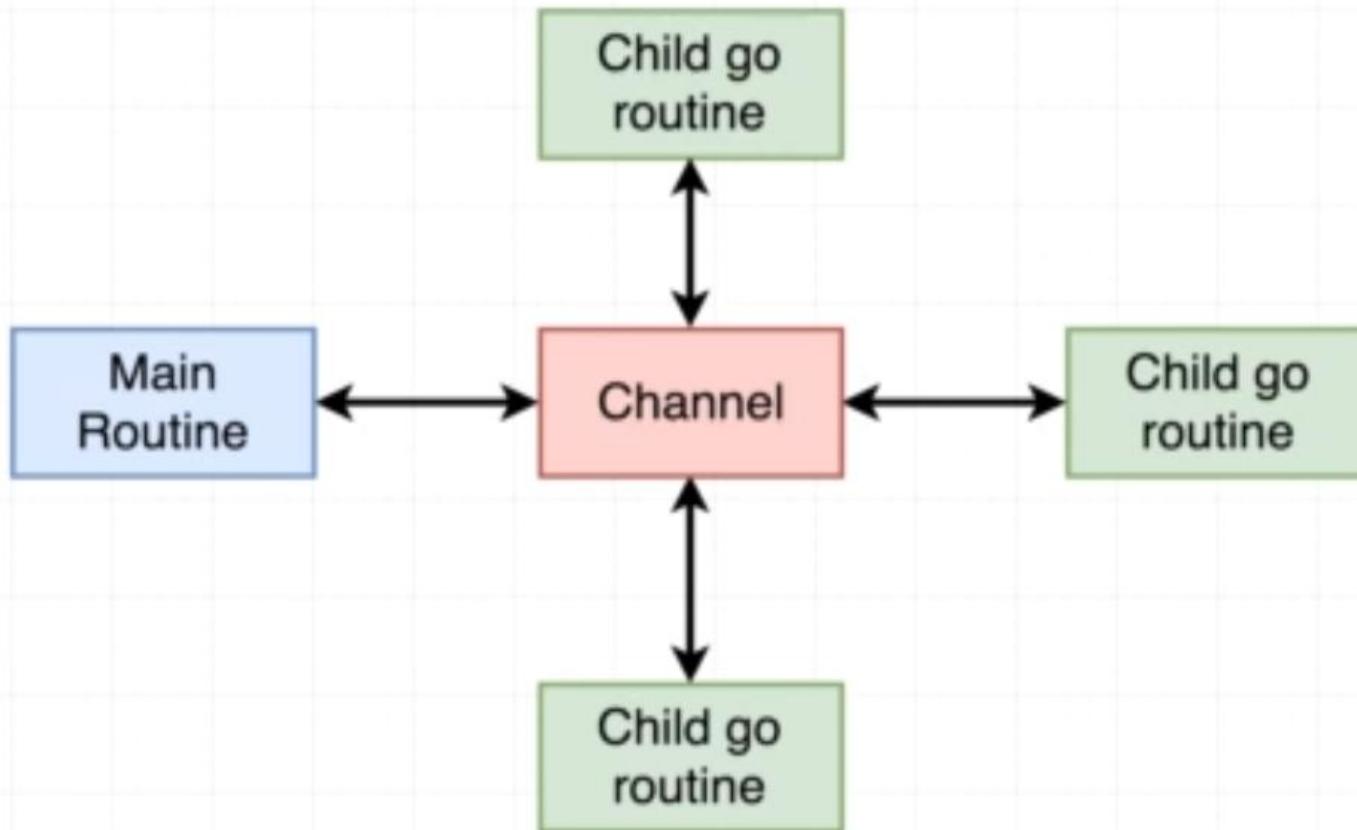
- The WaitGroup type of sync package, is used to wait for the program to finish all goroutines launched from the main function.
- It uses a counter that specifies the number of goroutines, and Wait blocks the execution of the program until the WaitGroup counter is zero.
- The Add method is used to add a counter to the WaitGroup.
- The Done method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter.
- The Wait method of the WaitGroup type waits for the program to finish all goroutines.
- The Wait method is called inside the main function, which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed.



Channels

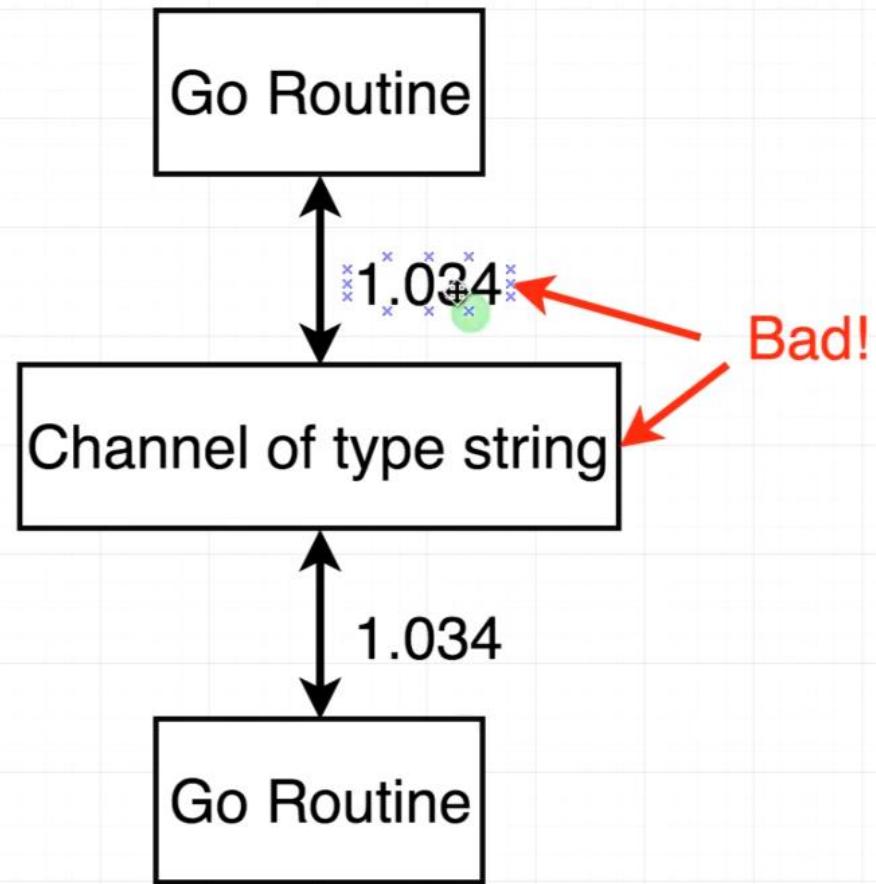
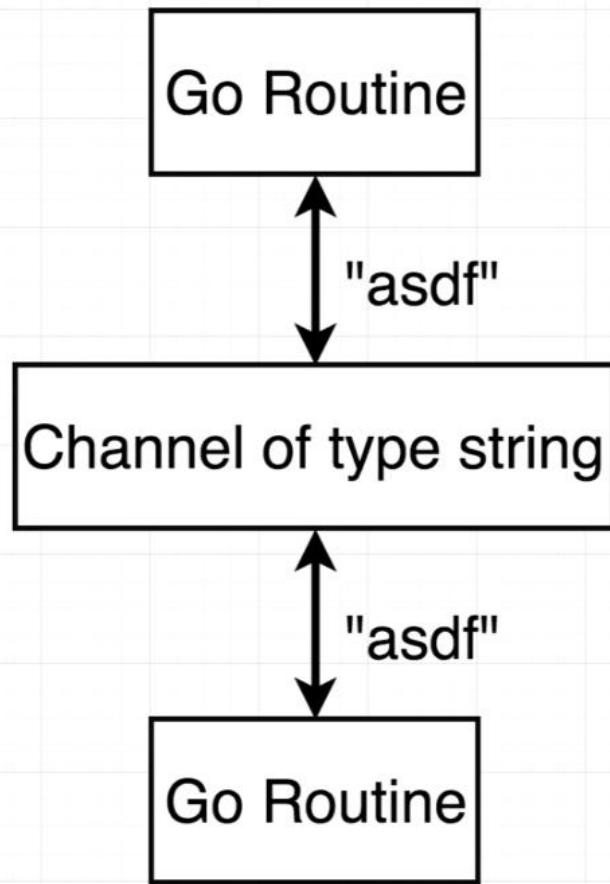
- Channels provide a way for two goroutines to communicate with one another and synchronize their execution.
- A channel allows one goroutine to signal another goroutine about a particular event.
- Signaling is at the core of everything you should be doing with channels.

Channels





Channels





Channels

```
main.go • func main() {
    9     links := []string{
10         "http://google.com",
11         "http://facebook.com",
12         "http://stackoverflow.com",
13         "http://golang.org",
14         "http://amazon.com",
15     }
16
17     c := make(chan string) | Create Channel
18
19     for _, link := range links {
20         go checkLink(link)
21     }
22 }
23
24 func checkLink(link string) {
```



Channels

- Channels are type safe message queues that have the intelligence to control the behavior of any goroutine attempting to receive or send on it.
- A channel acts as a conduit between two goroutines and will synchronize the exchange of any resource that is passed through it.
- It is the channel's ability to control the goroutines interaction that creates the synchronization mechanism.
- When a channel is created with no capacity, it is called an unbuffered channel.
- In turn, a channel created with capacity is called a buffered channel.



Access to Channel

```
10
11
12
13
14
15
16
17     c := make(chan string)
18
19     for _, link := range links {
20         go checkLink(link, c)
21     }
22 }
23
24 func checkLink(link string, c chan string) {
25     _, err := http.Get(link)
26     if err != nil {
27         fmt.Println(link, "might be down!")
28         return
29     }
30
31     fmt.Println(link, "is up!")
32 }
```



Channel Messaging

Sending Data with Channels

```
channel <- 5
```

Send the value '5' into this channel

```
myNumber <- channel
```

Wait for a value to be sent into the channel. When we get one, assign the value to 'myNumber'

```
fmt.Println(<- channel)
```

Wait for a value to be sent into the channel. When we get one, log it out immediately



Channel Messaging

```
main.go
19    for _, link := range links {
20        go checkLink(link, c)
21    }
22 }
23
24 func checkLink(link string, c chan string) {
25     _, err := http.Get(link)
26     if err != nil {
27         fmt.Println(link, "might be down!")
28         c <- "Might be down I think"
29     }
30 }
31
32     fmt.Println(link, "is up!")
33     c <- "Yep its up"
34 }
```



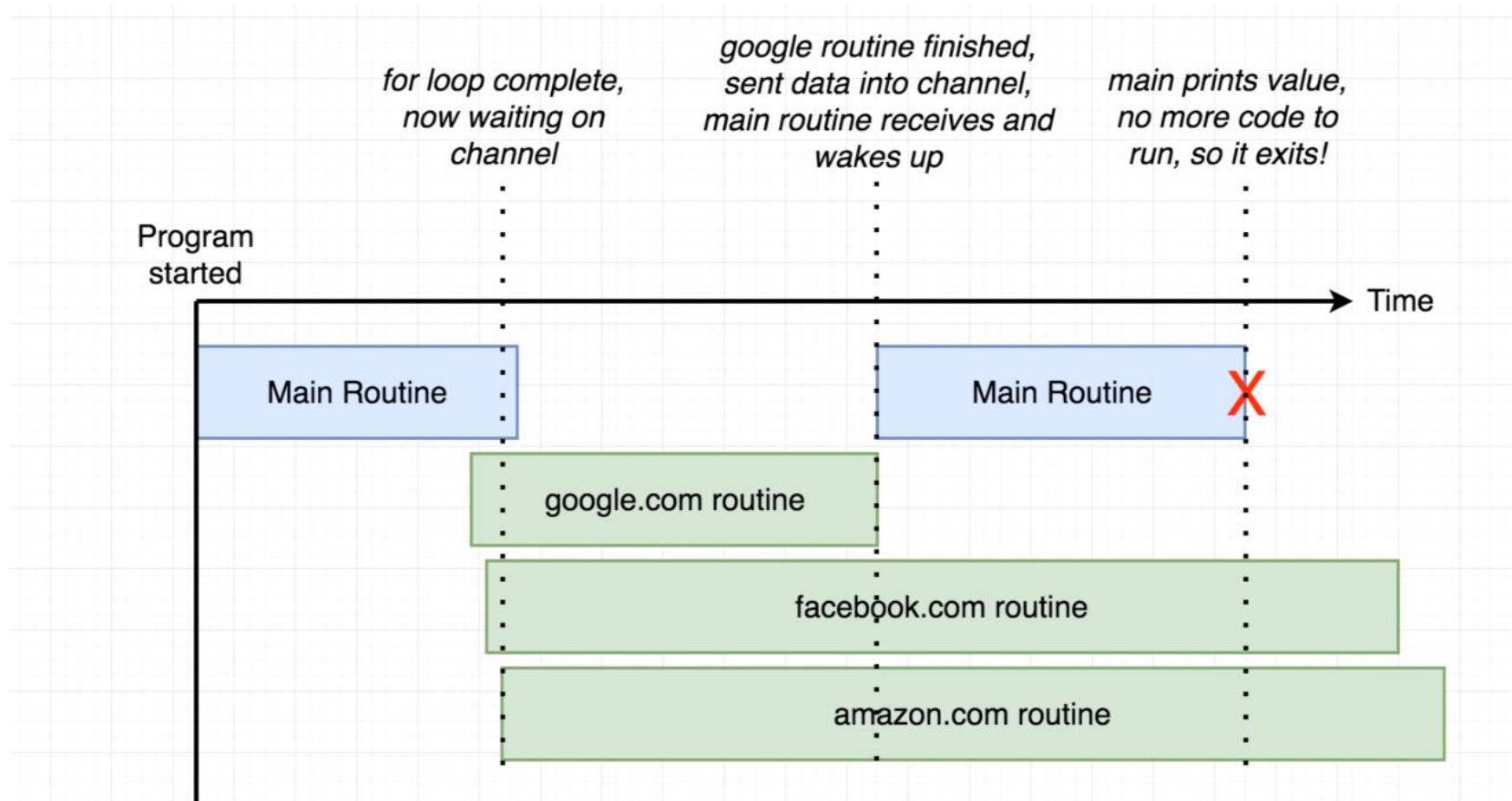
Channel Messaging

Receiver

```
main.go x
19     for _, link := range links {
20         go checkLink(link, c)
21     }
22
23     fmt.Println(<- c)
24 }
25
26 func checkLink(link string, c chan string) {
27     _, err := http.Get(link)
28     if err != nil {
29         fmt.Println(link, "might be down!")
30         c <- "Might be down I think"
31     }
32 }
33
34     fmt.Println(link, "is up!")
35     c <- "Yea its up"
```



Channel Messaging





Channel Messaging

Just one message

```
ws > day3 > blockingchannel > 🐈 main.go
  + channels\main.go x  serialchecker\main.go x  goroutinebug\main.go x  blockingchannel\main.go x
cogolangws [ciscows] F:\csc... 13     "http://stackoverflow.com",
| day1                                14     "http://golang.org",
| day2                                15     "http://amazon.com",
| day3                                16 }
| banking
| blockingchannel
|   🐈 main.go
| builder
| channels
|   🐈 main.go
| exportdemo
| goroutinebug
|   🐈 main.go
| packer
| serialchecker
|   🐈 main.go
| embeddedinterface.go
| go.mod
day4
| chatbot.go
day5
go build main.go (3) x
<4 go setup calls>
http://google.com is up!
Yes, It's up

Process finished with the exit code 0
```



Channel Messaging

Two messages

The screenshot shows an IDE interface with the following details:

- Project:** ciscogolangws
- File:** main.go (selected in the Project tree)
- Code Content:**

```
13     "http://stackoverflow.com",
14     "http://golang.org",
15     "http://amazon.com",
16 }
17
18 c:=make(chan string);
19
20 for _, link := range links {
21     go checkLink(link,c)
22 }
23
24 fmt.Println(<-c);
25
26 }
27 }
```

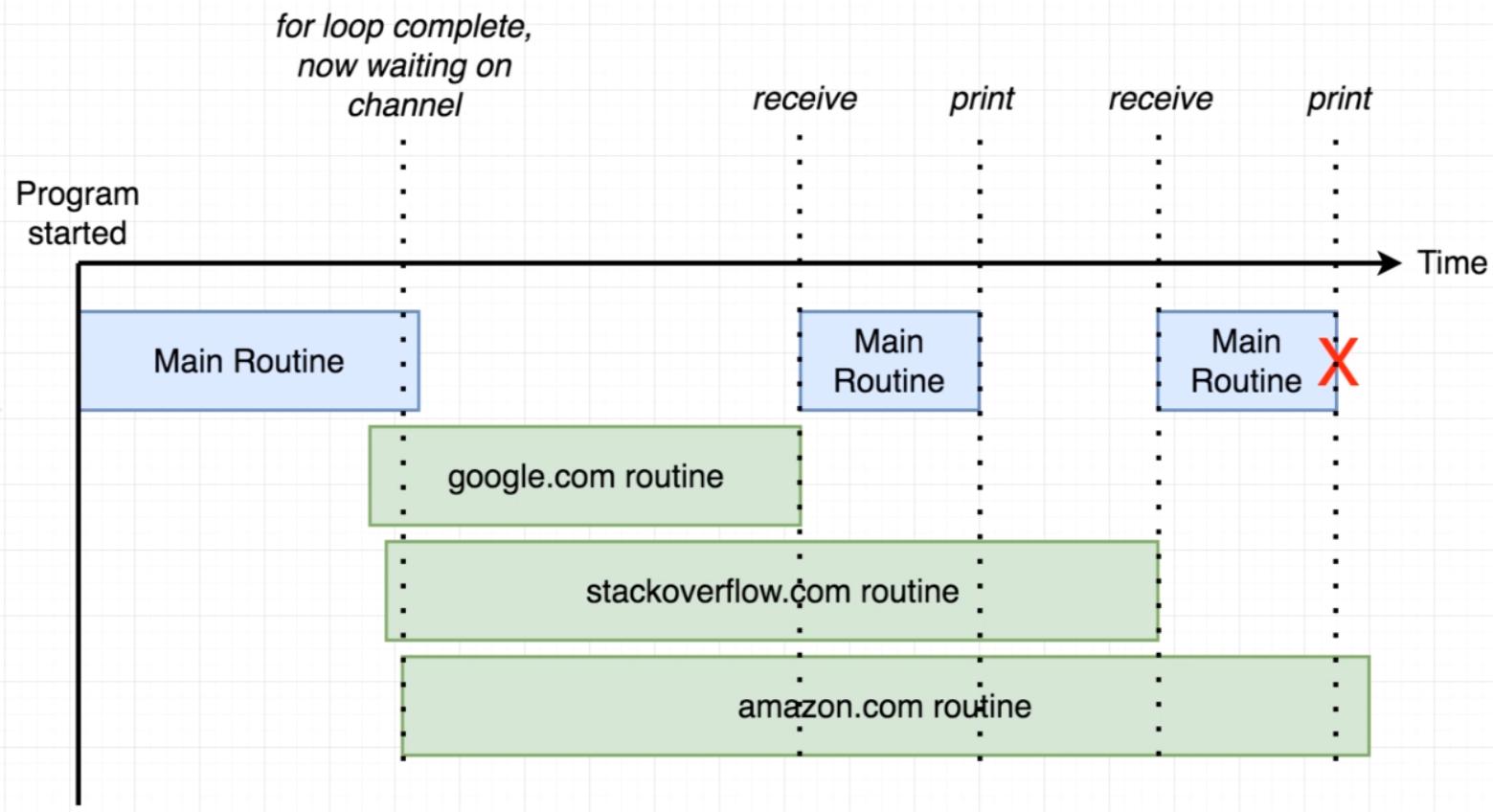
At the bottom of the code editor, there is a function definition:

```
func checkLink(link string, c chan string) {
```
- Run Output:**

```
<4 go setup calls>
http://google.com is up!
Yes, It's up
http://golang.org is up!
Yes, It's up
Process finished with the exit code 0
```



Channel Messaging





Channel Messaging

```
main.go • C ./main Chan String  
18  
19     for _, link := range links {  
20         go checkLink(link, c)  
21     }  
22  
23     fmt.Println(<-c)  
24     fmt.Println(<-c)  
25     fmt.Println(<-c)  
26     fmt.Println(<-c)  
27     fmt.Println(<-c)  
28 }  
29  
30 func checkLink(link string, c chan string) {  
31     _, err := http.Get(link)  
32     if err != nil {  
33         fmt.Println(link, "might be down!")  
        main.go
```



Channel Messaging

```
http://stackoverflow.com is up!
Yep its up
→ channels git:(master) ✘ go run main.go
http://stackoverflow.com is up!
Yep its up
http://google.com is up!
Yep its up
http://golang.org is up!
Yep its up
http://facebook.com is up!
Yep its up
http://amazon.com is up!
Yep its up
→ channels git:(master) ✘ █
```



Channel Messaging

```
main.go └── main.go — channels
17   c := make(chan string)
18
19   for _, link := range links {
20     go checkLink(link, c)
21   }
22
23   fmt.Println(<-c)
24   fmt.Println(<-c)
25   fmt.Println(<-c)
26   fmt.Println(<-c)
27   fmt.Println(<-c)
28
29   fmt.Println(<-c) | ↴
30 }
31
32 func checkLink(link string, c chan string) {
33   _, err := http.Get(link)
```



Channel with additional channel return

```
→ channels git:(master) ✘ go run main.go
http://google.com is up!
Yep its up
http://golang.org is up!
Yep its up
http://facebook.com is up!
Yep its up
http://stackoverflow.com is up!
Yep its up
http://amazon.com is up!
Yep its up
```

⋮



Channel with additional channel return

```
main.go x
14     "http://amazon.com",
15 }
16
17 c := make(chan string)
18
19 for _, link := range links {
20     go checkLink(link, c)
21 }
22
23 for i := 0; i < len(links); i++ {
24     fmt.Println(<-c)
25 }
26 }
27
28 func checkLink(link string, c chan string) {
29     _, err := http.Get(link)
re. 30     if err != nil {
```



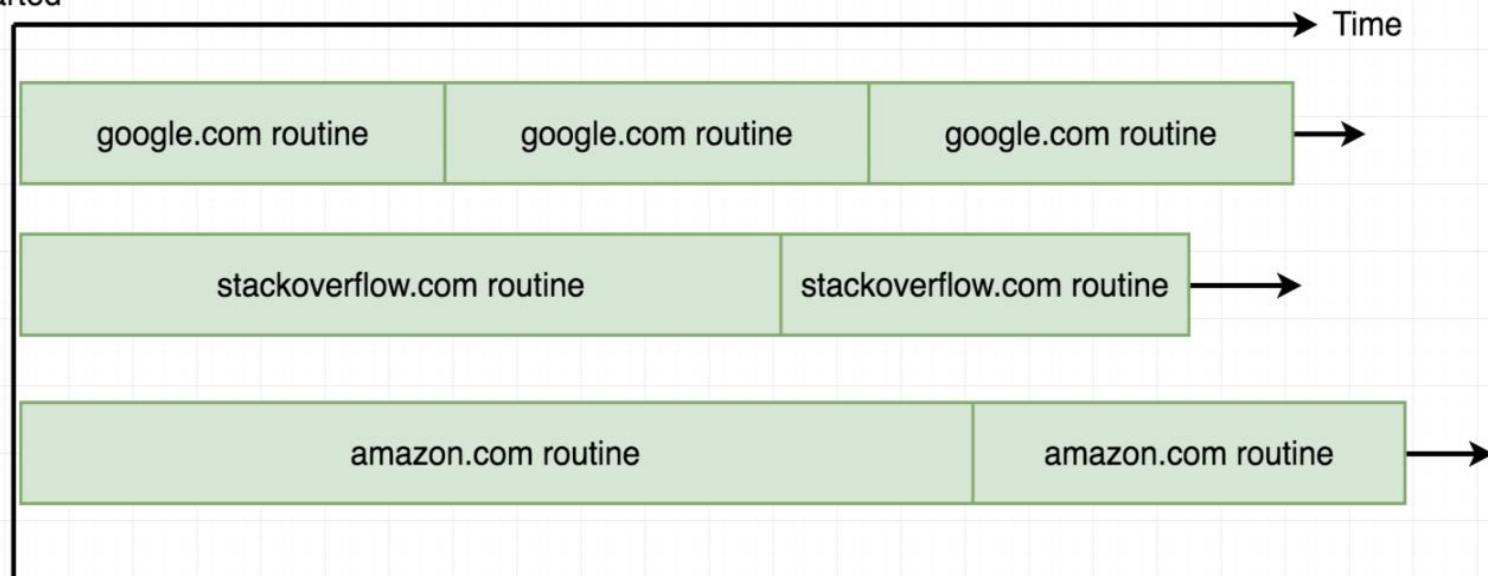
Channel Direction

- We can specify a direction on a channel type, thus restricting it to either sending or receiving. For example, pinger's function signature can be changed to this:
- **func pinger(c chan<- string)**
- Now pinger is only allowed to send to c. Attempting to receive from c will result in a compile-time error. Similarly, we can change printer to this:
- **func printer(c <-chan string)**
- A channel that doesn't have these restrictions is known as bidirectional.
- A bidirectional channel can be passed to a function that takes send-only or receive-only channels, but the reverse is not true.



Repeating Routines

Program
started





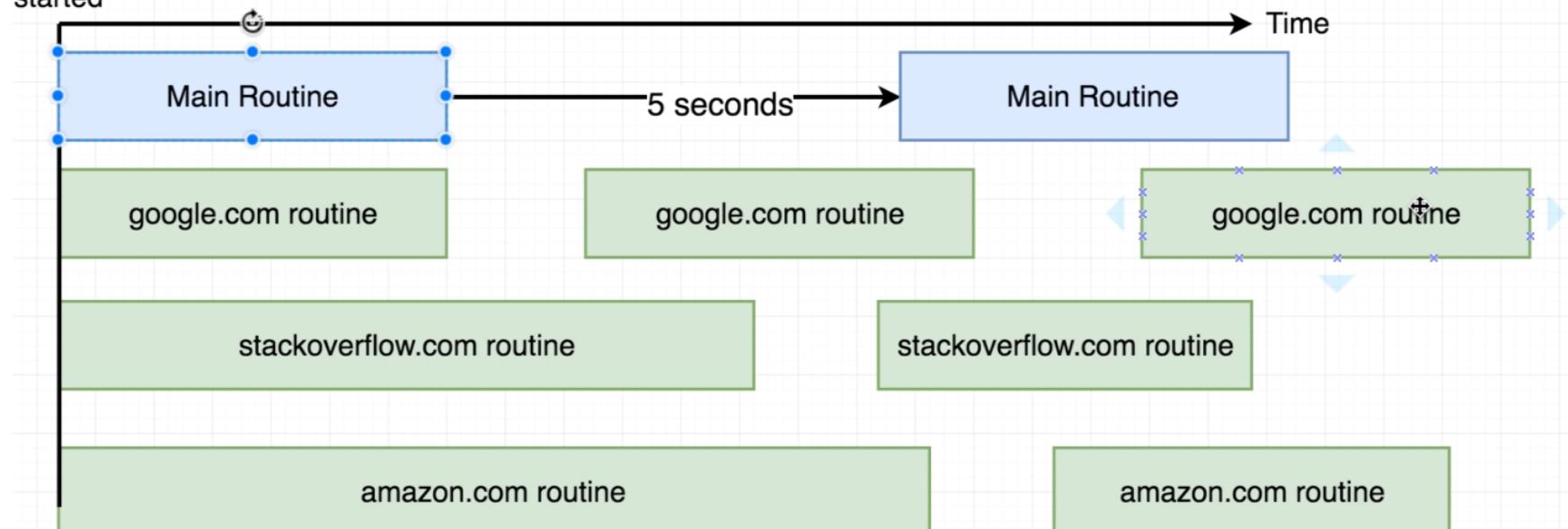
Repeating Routines

```
main.go x
26 }
27
28 func checkLink(link string, c chan string) {
29     _, err := http.Get(link)
30     if err != nil {
31         fmt.Println(link, "might be down!")
32         c <- link
33     }
34 }
35
36     fmt.Println(link, "is up!")
37     c <- link
38 }
39
```



Go Routines Sleep

?program
started





Go Routines Sleep

```
main.go x
19
20     for _, link := range links {
21         go checkLink(link, c)
22     }
23
24     for l := range c {
25         time.Sleep(5 * time.Second)
26         go checkLink(l, c)
27     }
28 }
29
30 func checkLink(link string, c chan string) {
31     _, err := http.Get(link)
32     if err != nil {
33         fmt.Println(link, "might be down!")
34         c <- link
35     }
36 }
```



Channel Select

- Go has a special statement called select that works like a switch but for channels.

```
go func() {
    for {
        select {
        case msg1 := <- c1:
            fmt.Println(msg1)
        case msg2 := <- c2:
            fmt.Println(msg2)
        }
    }
}()
```



Buffered Channel

- Buffered channels have capacity and therefore can behave a bit differently.
- When a goroutine attempts to send a resource to a buffered channel and the channel is full, the channel will lock the goroutine and make it wait until a buffer becomes available.
- If there is room in the channel, the send can take place immediately and the goroutine can move on.
- When a goroutine attempts to receive from a buffered channel and the buffered channel is empty, the channel will lock the goroutine and make it wait until a resource has been sent.

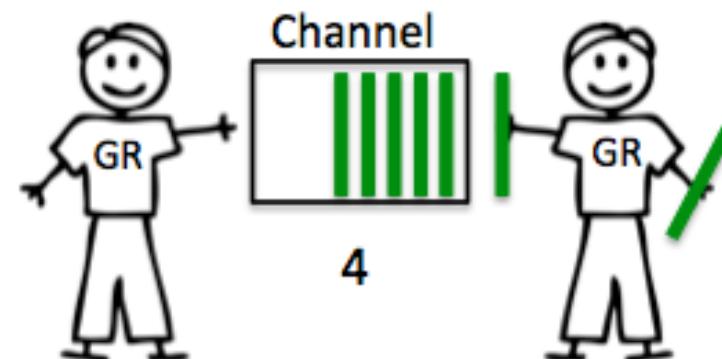
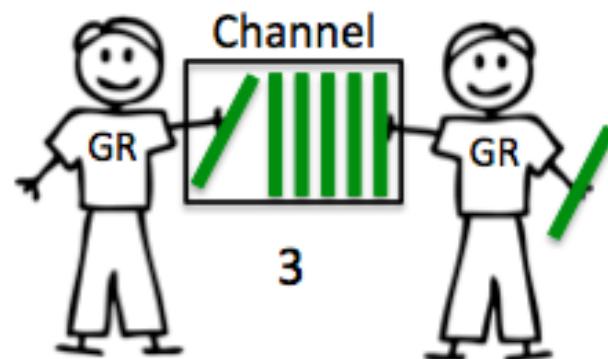
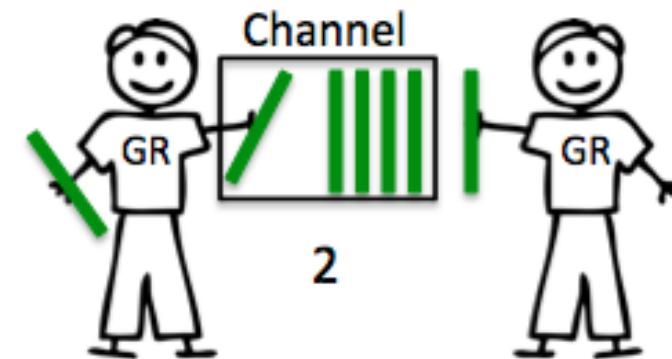
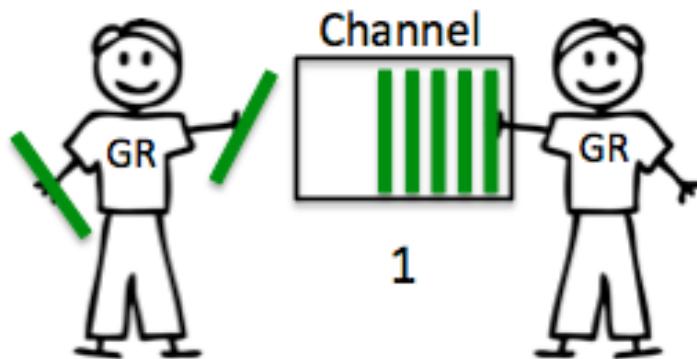


Buffered Channels

- It's also possible to pass a second parameter to the make function when creating a channel:
- `c := make(chan int, 10)`
- This creates a buffered channel with a capacity of 10.
- Normally, channels are synchronous; both sides of the channel will wait until the other side is ready.
- A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full.
- If the channel is full, then sending will wait until there is room for at least one more int.



Buffered Channel





Buffered Channels

- **goroutine1 := make(chan string, 5) // Buffered channel of strings.**
- **goroutine1 <- "Australia" // Send a string through the channel.**
- A **goroutine1** channel of type string that contains a buffer of 5 values.
- Then we send the string "Australia" through the channel.

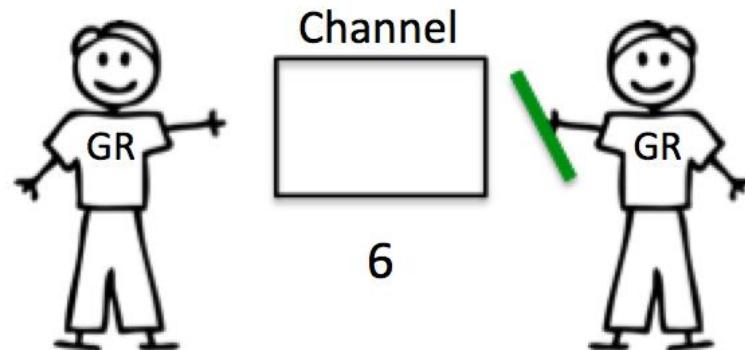
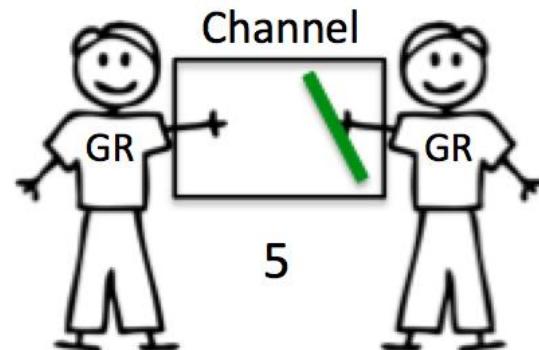
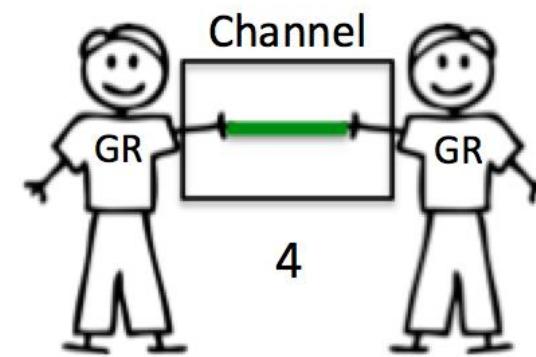
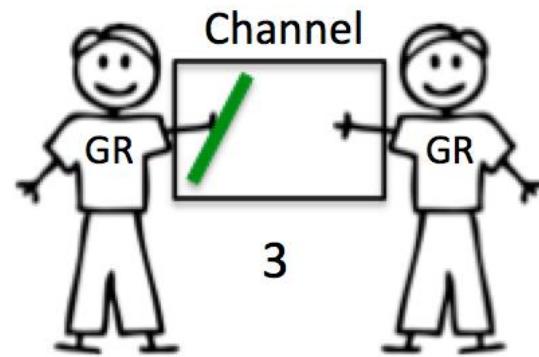
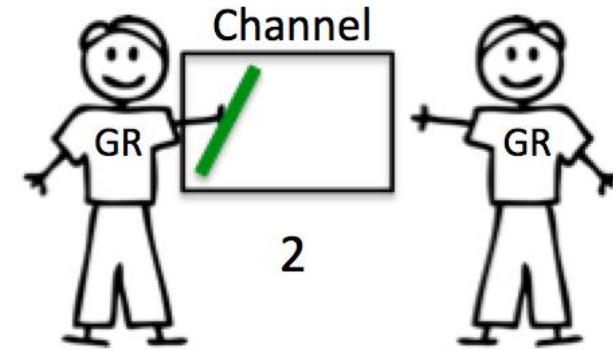
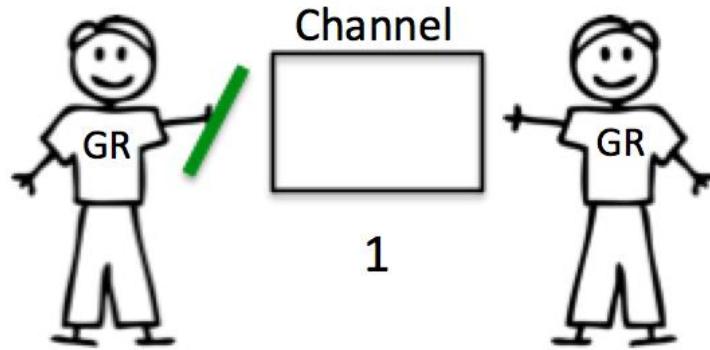


UnBuffered Channel

- An unbuffered channel is a channel that needs a receiver as soon as a message is emitted to the channel.
- To declare an unbuffered channel, you just don't declare a capacity.
- Unbuffered channels have no capacity and therefore require both goroutines to be ready to make any exchange.
- When a goroutine attempts to send a resource to an unbuffered channel and there is no goroutine waiting to receive the resource, the channel will lock the sending goroutine and make it wait.
- When a goroutine attempts to receive from an unbuffered channel, and there is no goroutine waiting to send a resource, the channel will lock the receiving goroutine and make it wait.



UnBuffered Channel





Concurrent clock server

- Networking is a natural domain to use concurrency since servers typically handle many connections from their clients at once, each client being essentially independent of the others.
- Net package, which provides the components for building networked client and server programs that communicate over TCP, UDP, or Unix domain sockets.



Concurrent clock server

- The Listen function creates a net.Listener, an object that listens for incoming connections on a network port, in this case TCP port localhost:8000.
- The listener's Accept method blocks until an incoming connection request is made, then returns a net.Conn object representing the connection.
- The handleConn function handles one complete client connection.
- In a loop, it writes the current time, time.Now(), to the client.
- Since net.Conn satisfies the io.Writer interface, we can write directly to it.
- The loop ends when the write fails, most likely because the client has disconnected, at which point handleConn closes its side of the connection using a deferred call to Close and goes back to waiting for another connection request.



Concurrent clock server

- The `time.Time.Format` method provides a way to format date and time information by example.
- Its argument is a template indicating how to format a reference time, specifically `Mon Jan 2 03:04:05PM 2006 UTC-0700`.
- The reference time has eight components.
- Any collection of them can appear in the `Format` string in any order and in a number of formats; the selected components of the date and time will be displayed in the selected formats.
- This example uses the hour, minute, and second of the time.
- The `time` package defines templates for many standard time formats, such as `time.RFC1123`. The same mechanism is used in reverse when parsing a time using `time.Parse`.



Concurrent clock server

```
C:\ Administrator: Command Prompt - clockserver
F:\go\src\awesomeProject\Day4>cd clockserver
F:\go\src\awesomeProject\Day4\clockserver>go build
F:\go\src\awesomeProject\Day4\clockserver>dir
 Volume in drive F is New Volume
 Volume Serial Number is 5641-E892

Directory of F:\go\src\awesomeProject\Day4\clockserver

21/01/2021  08:31 AM    <DIR>          .
21/01/2021  08:31 AM    <DIR>          ..
21/01/2021  08:31 AM           2,674,688 clockserver.exe
21/01/2021  08:28 AM                612 concurrentclockserver.
              2 File(s)       2,675,300 bytes
              2 Dir(s)   43,784,691,712 bytes free

F:\go\src\awesomeProject\Day4\clockserver>clockserver
F:\go\src\awesomeProject\Day4\clockserver>clockserver
```

```
C:\ Telnet localhost
08:32:45
08:32:46
```



Concurrent Echo Server

```
F:\go\src\awesomeProject\Day4\clockserver>cd..  
F:\go\src\awesomeProject\Day4>cd echoserver  
F:\go\src\awesomeProject\Day4\echoserver>go build  
F:\go\src\awesomeProject\Day4\echoserver>dir  
Volume in drive F is New Volume  
Volume Serial Number is 5641-E892  
  
Directory of F:\go\src\awesomeProject\Day4\echoserver  
  
21/01/2021 08:44 AM <DIR> .  
21/01/2021 08:44 AM <DIR> ..  
21/01/2021 08:44 AM 832 concurrentechoserver.go  
21/01/2021 08:44 AM 2,711,552 echoserver.exe  
2 File(s) 2,712,384 bytes  
2 Dir(s) 43,774,881,792 bytes free  
  
F:\go\src\awesomeProject\Day4\echoserver>echoserver
```

```
F:\ Telnet localhost  
  
HELLO Hello hello  
HI Hi ho hi  
HOW ARE YOU w are you  
DONE done done  
how are you  
done
```



Channels

- Three channel attributes
 - Guarantee Of Delivery
 - State
 - With or Without Data



Guarantee of Delivery

	Unbuffered	Buffered
Delivery	Guaranteed	Not Guaranteed



State

	NIL	Open	Closed
Send	Blocked	Allowed	Panic
Receive	Blocked	Allowed	Allowed



With and Without Data

Signaling With Data

	Guarantee	No Guarantee	Delayed Guarantee
Channel	Unbuffered	Buffered >1	Buffered =1



With and Without Data

Signaling Without Data

	First Choice	Second Choice	Smell
Channel	context.Context	Unbuffered	Buffered

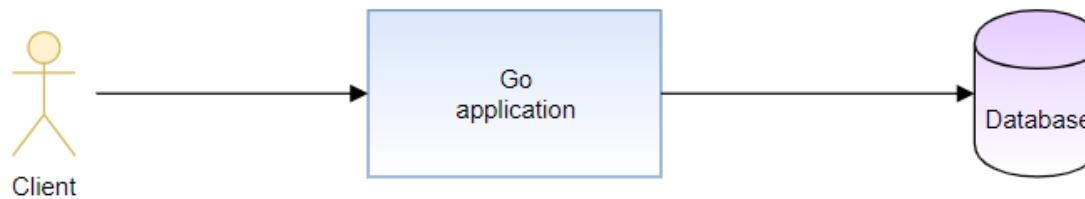


Cancellation

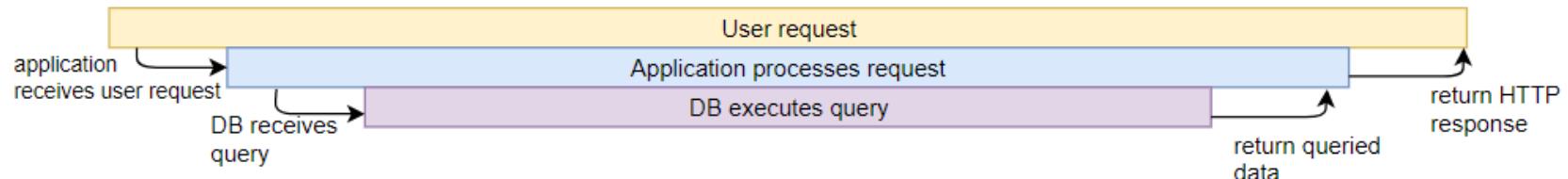
- Most use context with downstream operations, like making an HTTP call, or fetching data from a database, or while performing async operations with go-routines.
- It's most common use is to pass down common data which can be used by all downstream operations.
- However, a lesser known, but highly useful feature of context is it's ability to cancel, or halt an operation mid-way.



Cancellation



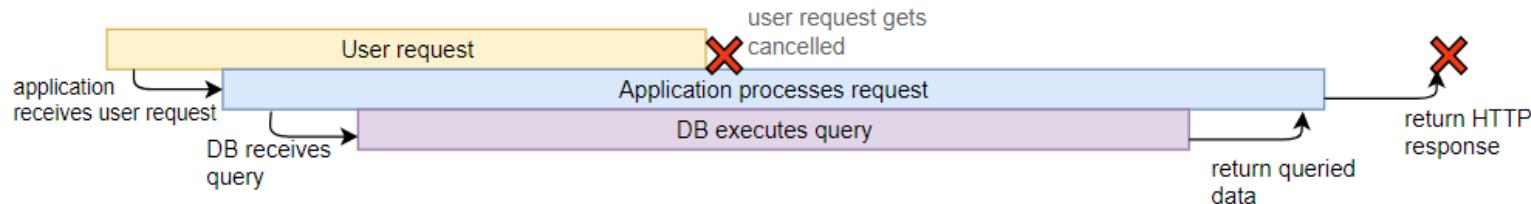
The timing diagram, if everything worked perfectly, would look like this:



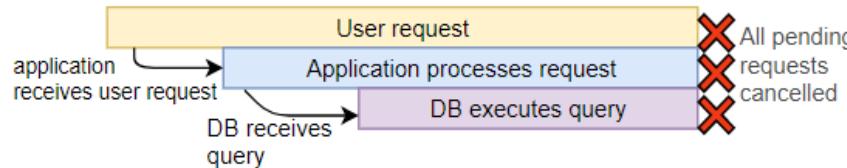


Cancellation

- if the client cancelled the request in the middle? This could happen if, for example, the client closed their browser mid-request.
- Without cancellation, the application server and database would continue to do their work, even though the result of that work would be wasted:



Ideally, we would want all downstream components of a process to halt, if we know that the process (in this example, the HTTP request) halted:





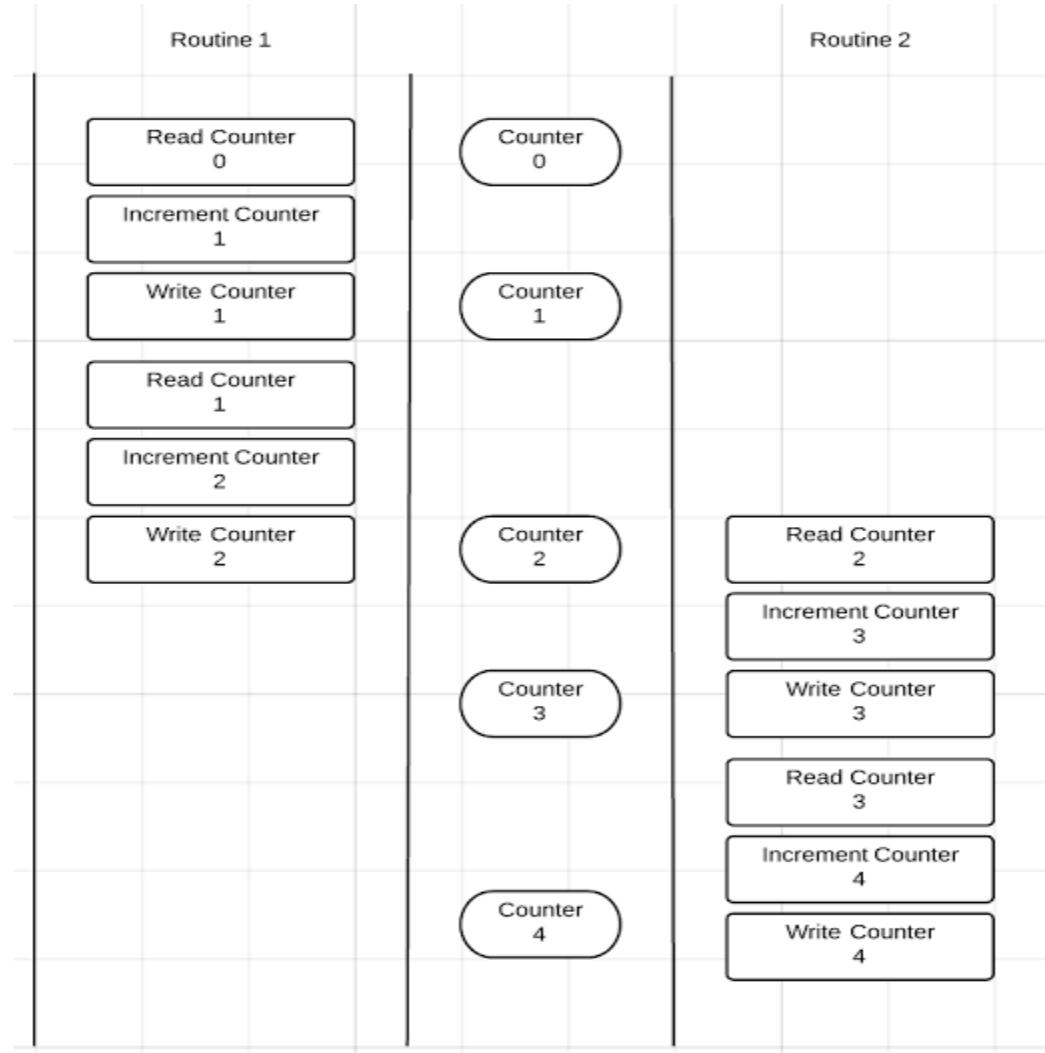
Cancellation

```
F:\go\src\awesomeProject\day5\cancellation>go run cancel_listen.go
processing request
request cancelled
exit status 2
```

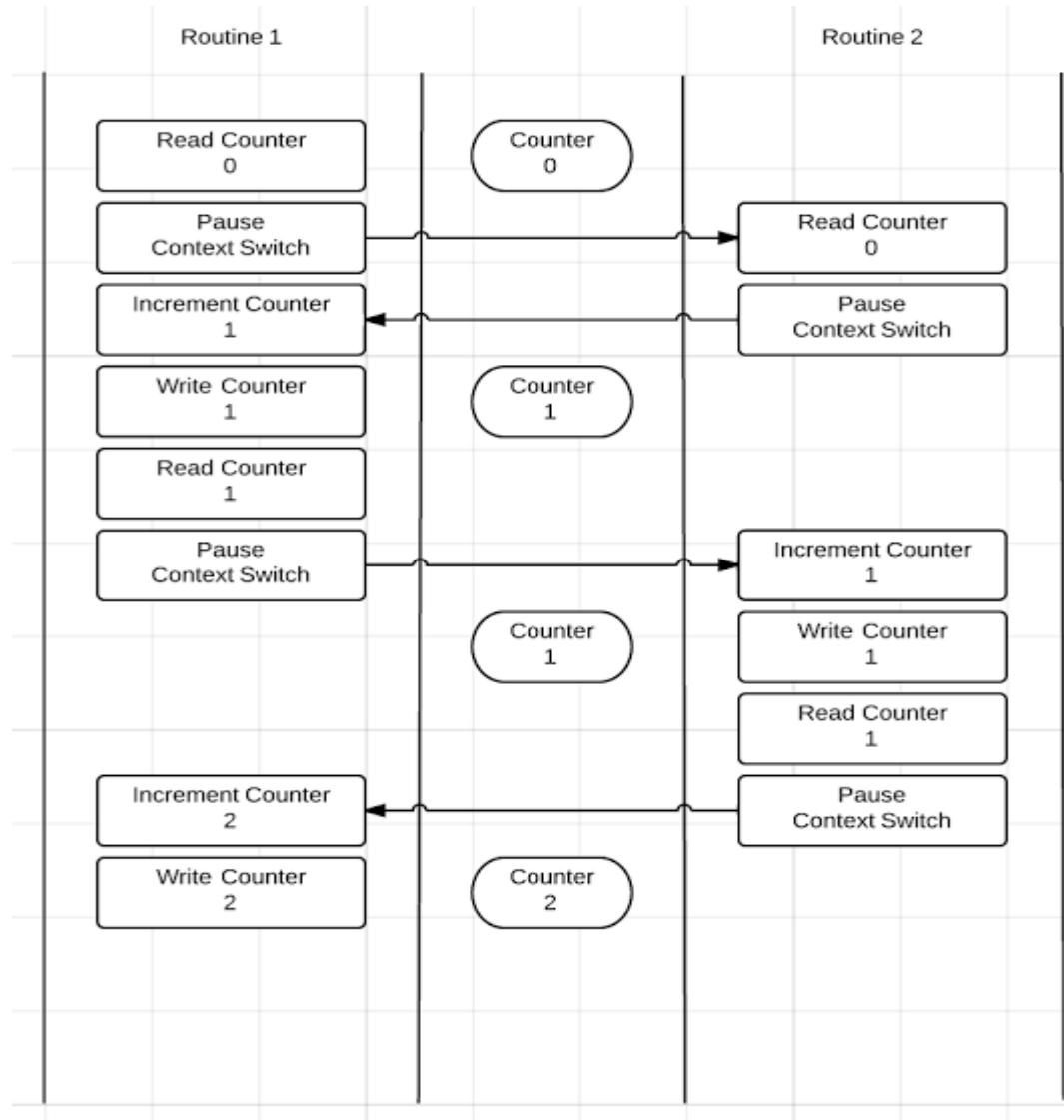
```
F:\go\src\awesomeProject\day5\cancellation>
```



Race Conditions



Race Conditions





Race Condition Detector

- A race condition is when two or more routines have access to the same resource, such as a variable or data structure and attempt to read and write to that resource without any regard to the other routines.
- This type of code can create the craziest and most random bugs you have ever seen.
- It usually takes a tremendous amount of logging and luck to find these types of bugs.
- **go build -race**



Race Condition Detector

```
F:\go\src\awesomeProject\day5\raceconditions>go build --race
F:\go\src\awesomeProject\day5\raceconditions>raceconditions
=====
WARNING: DATA RACE
Read at 0x0000011d45e8 by goroutine 8:
  main.Routine()
    F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:27 +0x4e

Previous write at 0x0000011d45e8 by goroutine 7:
  main.Routine()
    F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:29 +0x6a

Goroutine 8 (running) created at:
  main.main()
    F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:16 +0x7c

Goroutine 7 (finished) created at:
  main.main()
    F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:16 +0x7c
=====
Final Counter: 4
Found 1 data race(s)

F:\go\src\awesomeProject\day5\raceconditions>
```

Race detector has pulled out the two lines of code that is reading and writing to the global Counter variable. It also identified the point in the code where the routine was spawned.



Understanding Mutexes

- Mutual exclusion is a property of concurrency control which states that " NO TWO PROCESS SHOULD ACCESS THE SAME RESOURCE AT THE SAME TIME ".



Understanding Mutexes

Instruction	Goroutine 1	Goroutine 2	Bank Balance
1	Read balance \Leftarrow £50		£50
2		Read balance \Leftarrow £50	£50
3	Add £100 to balance		—
4		Add £50 to balance	—
5	Write balance \Rightarrow £150		£150
6		Write balance \Rightarrow £100	£100



How Concurrency and Parallelism works in Golang

- Concurrency: Concurrency is about dealing with lots of things at once.
- This means that we manage to get multiple things done at once in a given period of time.
- However, we will only be doing a single thing at a time.
- This tends to happen in programs where one task is waiting and the program decides to run another task in the idle time.
- In diagram, this is denoted by running the yellow task in idle periods of the blue task.



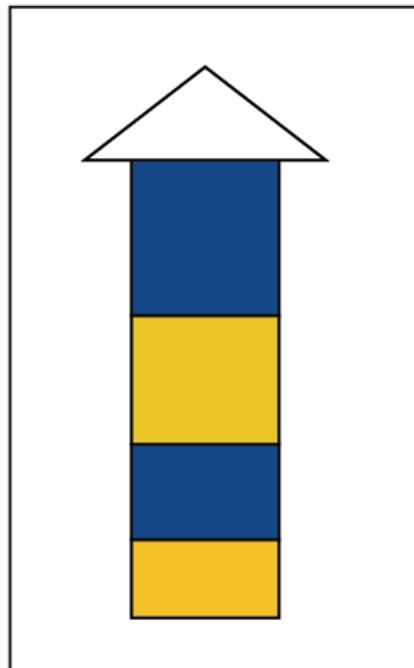
How Concurrency and Parallelism works in Golang

- Parallelism: Parallelism is about doing lots of things at once.
- This means that even if we have two tasks, they are continuously working without any breaks in between them.
- In the diagram, this is shown by the fact that the green task is running independently and is not influenced by the red task in any manner:



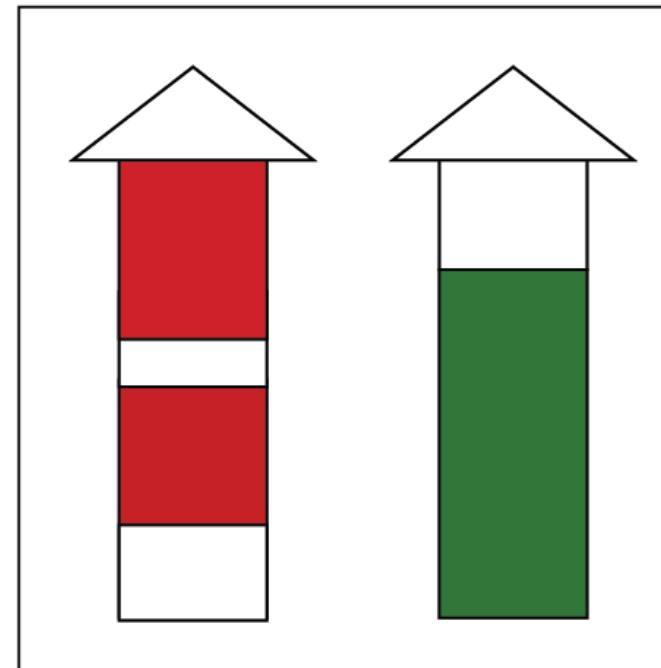
How Concurrency and Parallelism works in Golang

Concurrency



Concurrency is about *dealing with*
lots of things at once

Parallelism



Parallelism is about *doing*
lots of things at once



How Concurrency and Parallelism works in Golang

- Concurrency
- Imagine you start your day and need to get six things done:
 - Make hotel reservation
 - Book flight tickets
 - Order a dress
 - Pay credit card bills
 - Write an email
 - Listen to an audiobook



How Concurrency and Parallelism works in Golang

- The order in which they are completed doesn't matter, and for some of the tasks, such as writing an email or listening to an audiobook, you need not complete them in a single sitting. Here is one possible way to complete the tasks:
 - Order a dress.
 - Write one-third of the email.
 - Make hotel reservation.
 - Listen to 10 minutes of audiobook.
 - Pay credit card bills.
 - Write another one-third of the email.
 - Book flight tickets.
 - Listen to another 20 minutes of audiobook.
 - Complete writing the email.
 - Continue listening to audiobook until you fall asleep.



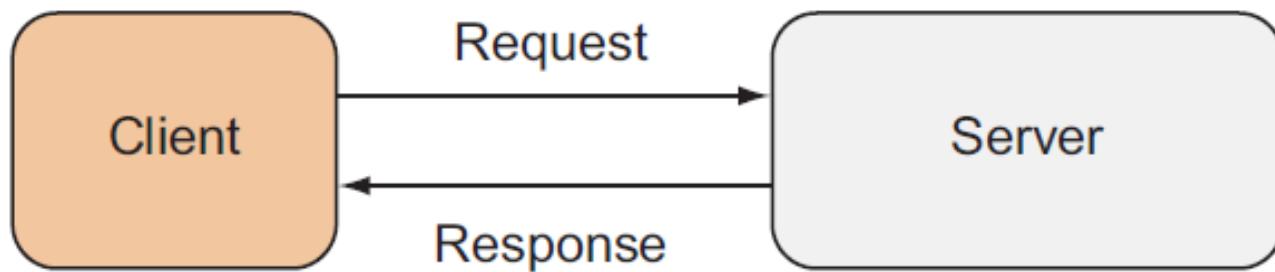
Web Development

Large-scale web applications typically need to be

- Scalable
- Modular
- Maintainable
- High-performance



Web Development





Parts of Web

- Handler
 - A handler receives and processes the HTTP request sent from the client.
 - It also calls the template engine to generate the HTML and finally bundles data into the HTTP response to be sent back to the client.
- Template engine
 - A template is code that can be converted into HTML that's sent back to the client in an HTTP response message.
 - Templates can be partly in HTML or not at all.
 - A template engine generates the final HTML using templates and data.



Parts of Web

- There are two types of templates with different design philosophies:
- Static templates or logic-less templates are HTML interspersed with placeholder tokens.
 - A static template engine will generate the HTML by replacing these tokens with the correct data.
 - Examples of static template engines are CTemplate and Mustache



Parts of Web

- Active Templates
 - Active templates often contain HTML too, but in addition to placeholder tokens, they contain other programming language constructs like conditionals, iterators, and variables.
 - Examples of active template engines are Java ServerPages (JSP), Active Server Pages (ASP), and Embedded Ruby (ERB).
 - PHP started off as a kind of active template engine and has evolved into its own programming language.



Sample App

```
package main

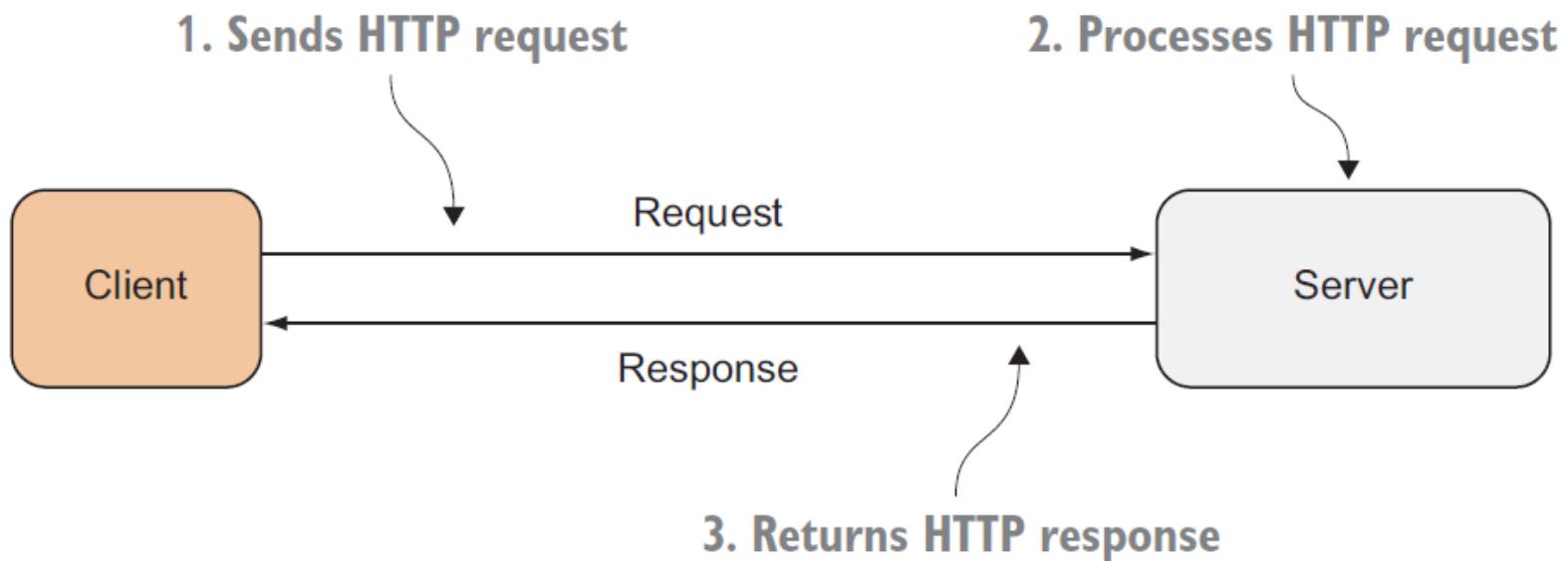
import (
    "fmt"
    "net/http"
)

func handler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "Hello World, %s!", request.URL.Path[1:])
}

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



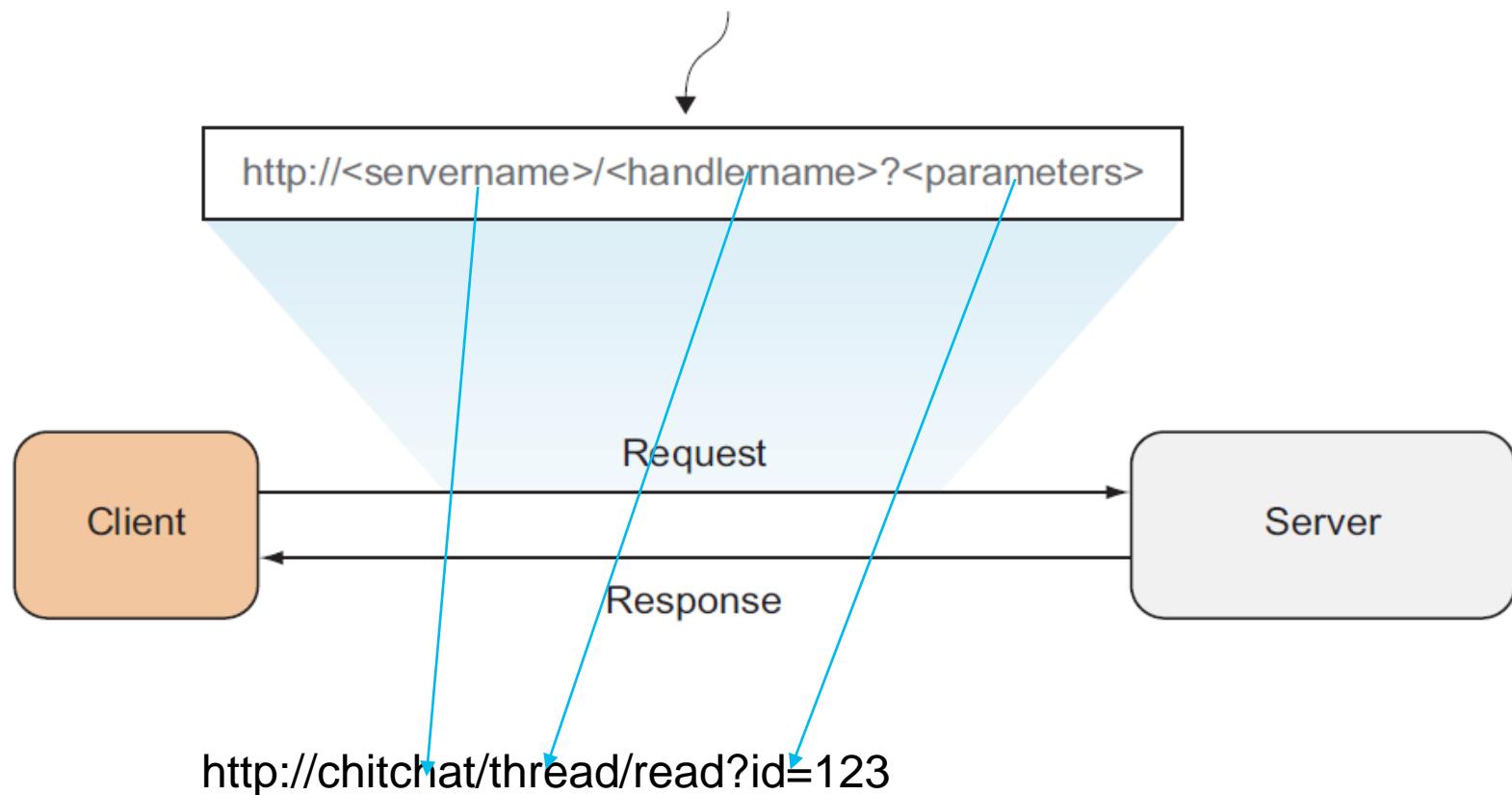
Application design





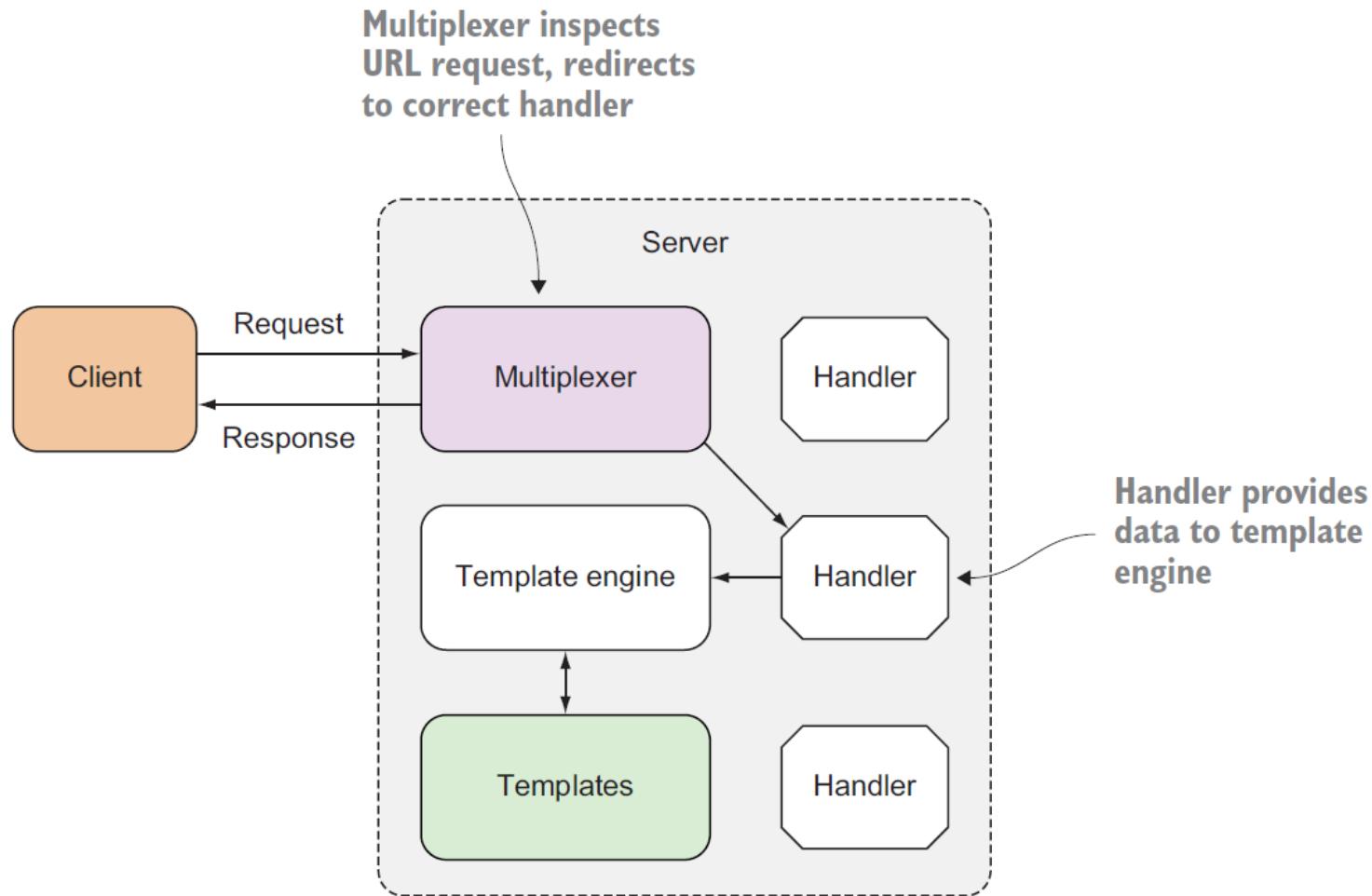
Application design

**Format of request is suggested by the web app,
in hyperlinks on HTML pages provided to client by server.**





Application design





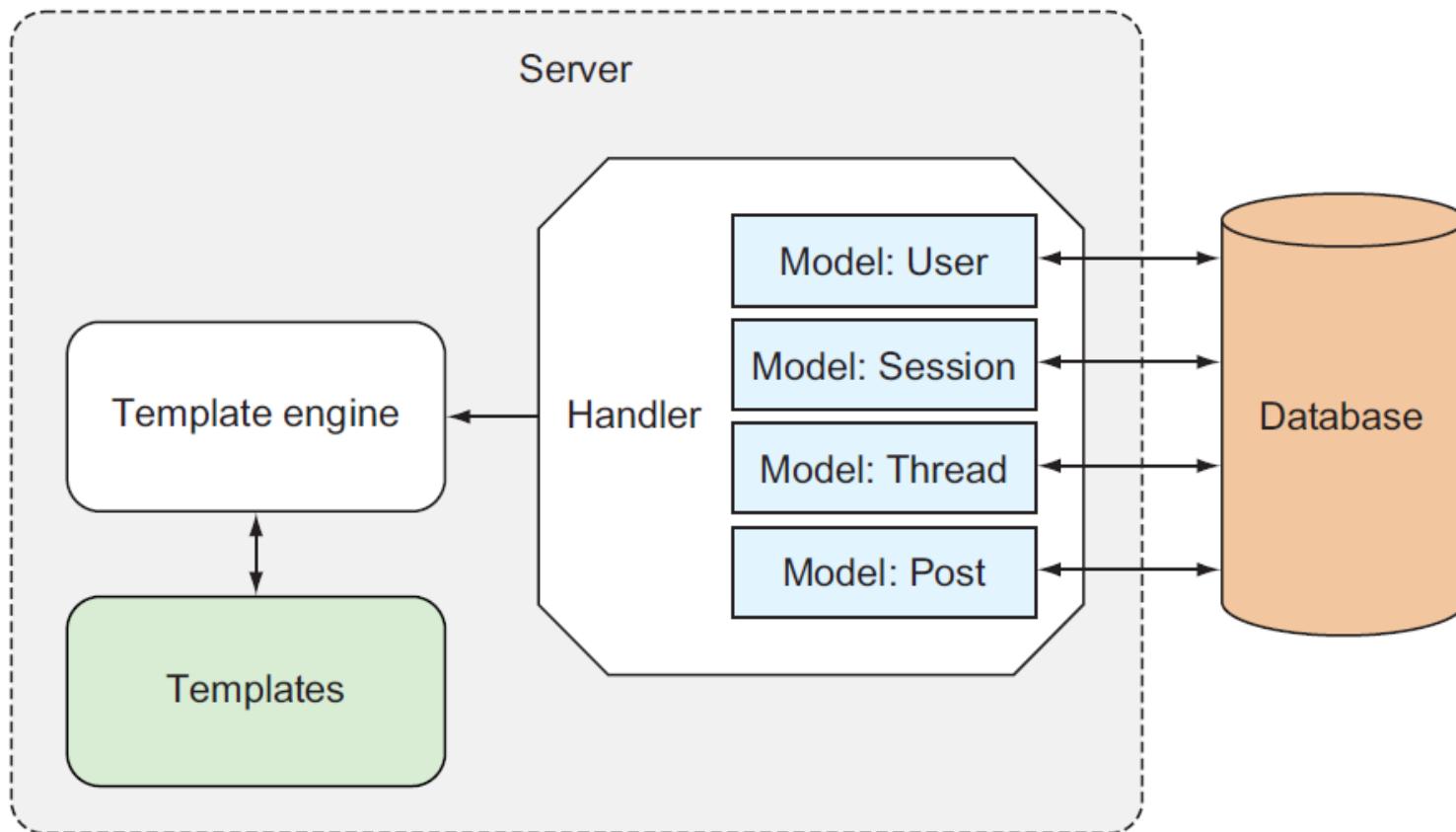
Data model

ChitChat's data model is simple and consists of only four data structures, which in turn map to a relational database.

The four data structures are

- User—Representing the forum user's information
- Session—Representing a user's current login session
- Thread—Representing a forum thread (a conversation among forum users)
- Post—Representing a post (a message added by a forum user) within a thread

Data model





Basic Authentication Http Server

```
func BasicAuth(handler http.HandlerFunc, realm string) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request)
    {
        user, pass, ok := r.BasicAuth()
        if !ok || subtle.ConstantTimeCompare([]byte(user),
            []byte(ADMIN_USER)) != 1 || subtle.ConstantTimeCompare([]byte(pass),
            []byte(ADMIN_PASSWORD)) != 1
        {
            w.Header().Set("WWW-Authenticate", `Basic realm=`+realm+``)
            w.WriteHeader(401)
            w.Write([]byte("You are Unauthorized to access the
                application.\n"))
            return
        }
        handler(w, r)
    }
}
```

Optimizing HTTP server responses with GZIP compression



- GZIP compression means sending the response to the client from the server in a .gzip format rather than sending a plain response and it's always a good practice to send compressed responses if a client/browser supports it.
- By sending a compressed response we save network bandwidth and download time eventually rendering the page faster.
- What happens in GZIP compression is the browser sends a request header telling the server it accepts compressed content (.gzip and .deflate) and if the server has the capability to send the response in compressed form then sends it.

Optimizing HTTP server responses with GZIP compression



```
package main
import
(
    "io"
    "net/http"
    "github.com/gorilla/handlers"
)
const
(
    CONN_HOST = "localhost"
    CONN_PORT = "8080"
)
func helloworld(w http.ResponseWriter, r *http.Request)
{
    io.WriteString(w, "Hello World!")
}
func main()
{
    mux := http.NewServeMux()
    mux.HandleFunc("/", helloworld)
    err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT,
        handlers.CompressHandler(mux))
    if err != nil
    {
        log.Fatal("error starting http server : ", err)
        return
    }
}
```

Optimizing HTTP server responses with GZIP compression



The screenshot shows the Postman application interface. At the top, there is a header bar with a 'GET' method, the URL 'http://localhost:7070', and various tool buttons. Below the header is a section titled 'Untitled Request' with tabs for 'Params', 'Authorization', 'Headers (8)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers' tab is currently selected. In the 'Headers' section, there is a table with columns 'KEY', 'VALUE', and 'DESCRIPTION'. A single row is present with the key 'Content-Encoding' and value 'gzip'. A blue arrow points from the text 'Content-Encoding' in the 'Headers' table to the 'gzip' value in the same table. At the bottom of the interface, there is a summary bar showing the status '200 OK', time '605 ms', size '200 B', and a 'Save Response' button.

KEY	VALUE	DESCRIPTION
Content-Encoding	gzip	Description

KEY	VALUE
Content-Type	text/plain; charset=utf-8
Vary	Accept-Encoding
Date	Sat, 16 Jan 2021 16:13:47 GMT
Content-Length	36



Creating a simple TCP server

- Whenever you have to build high performance oriented systems then writing a TCP server is always the best choice over an HTTP server, as TCP sockets are less hefty than HTTP.
- Go supports and provides a convenient way of writing TCP servers using a net package.



Creating a simple TCP server

```
package main
import
(
    "log"
    "net"
)
const
(
    CONN_HOST = "localhost"
    CONN_PORT = "8080"
    CONN_TYPE = "tcp"
)
func main()
{
    listener, err := net.Listen(CONN_TYPE, CONN_HOST+":"+CONN_PORT)
    if err != nil
    {
        log.Fatal("Error starting tcp server : ", err)
    }
    defer listener.Close()
    log.Println("Listening on " + CONN_HOST + ":" + CONN_PORT)
    for
    {
        conn, err := listener.Accept()
        if err != nil
        {
            log.Fatal("Error accepting: ", err.Error())
        }
        log.Println(conn)
    }
}
```



TCP server read data from incoming connections

```
func handleRequest(conn net.Conn)
{
    message, err := bufio.NewReader(conn).ReadString('\n')
    if err != nil
    {
        fmt.Println("Error reading:", err.Error())
    }
    fmt.Println("Message Received from the client: ", string(message))
    conn.Close()
}
```



Implementing HTTP request routing

```
func login(w http.ResponseWriter, r *http.Request)
{
    fmt.Fprintf(w, "Login Page!")
}
func logout(w http.ResponseWriter, r *http.Request)
{
    fmt.Fprintf(w, "Logout Page!")
}
func main()
{
    http.HandleFunc("/", helloWorld)
    http.HandleFunc("/login", login)
    http.HandleFunc("/logout", logout)
    err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, nil)
    if err != nil
    {
        log.Fatal("error starting http server : ", err)
        return
    }
}
```

Implementing HTTP request routing using Gorilla Mux



```
var GetRequestHandler = http.HandlerFunc
(
    func(w http.ResponseWriter, r *http.Request)
    {
        w.Write([]byte("Hello World!"))
    }
)
var PostRequestHandler = http.HandlerFunc
(
    func(w http.ResponseWriter, r *http.Request)
    {
        w.Write([]byte("It's a Post Request!"))
    }
)
var PathVariableHandler = http.HandlerFunc
(
    func(w http.ResponseWriter, r *http.Request)
    {
        vars := mux.Vars(r)
        name := vars["name"]
        w.Write([]byte("Hi " + name))
    }
)
```

Working with Templates, Static Files, and HTML Forms



- Creating your first template
- Serving static files over HTTP
- Serving static files over HTTP using Gorilla Mux
- Creating your first HTML form
- Reading your first HTML form
- Validating your first HTML form
- Uploading your first file



Template file Parsing

```
Microsoft Windows [Version 10.0.19041.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd F:\go\src\awesomeProject\webmaster\first-template

C:\WINDOWS\system32>f:

F:\go\src\awesomeProject\webmaster\first-template>dir
 Volume in drive F is New Volume
 Volume Serial Number is 5641-E892

Directory of F:\go\src\awesomeProject\webmaster\first-template

16/01/2021  11:33 PM    <DIR>          .
16/01/2021  11:33 PM    <DIR>          ..
16/01/2021  11:33 PM           675 first-template.go
16/01/2021  11:21 PM    <DIR>          templates
               1 File(s)      675 bytes
               3 Dir(s)  43,791,962,112 bytes free

F:\go\src\awesomeProject\webmaster\first-template>go run first-template.go
```



Blank Imports

- A use case is when what you want is only the init function doing some initialization (for example registering themselves so that you don't have to explicitly call them).
- An example is the registering of some database drivers import (
 - "database/sql"
 - _ "github.com/ziutek/mymysql/godrv"
 -)

WEB DENTAL CARE PROJECT

Go run .

<http://localhost:6060/assets/index.html>

Go Tool

Go Tool cheatsheet



Go Test Tool

```
Microsoft Windows [Version 10.0.19041.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd F:\go\src\ciscows\day5\dentalapp

C:\WINDOWS\system32>f:

F:\go\src\ciscows\day5\dentalapp>go test ./
ok      _/F_/go/src/ciscows/day5/dentalapp      1.040s

F:\go\src\ciscows\day5\dentalapp>
```



Go Coverage

- go test -cover ./...
- go test -coverprofile=profile.out ./...
- go tool cover -html=profile.out
- If you want you can go a step further and set the -covermode=count flag to make the coverage profile record the exact number of times that each statement is executed during the tests.
- go test -covermode=count -coverprofile=profile.out ./...
- go tool cover -html=profile.out



Go coverage

An Overview of Go's Tooling - Al... X Go Coverage Report X +

File | C:/Users/Balasubramaniam/AppData/Local/Temp/cover455487439/coverage.html#file0

F:\golsrc\discows\day5\identapp\main.go (30.0%) not tracked not covered covered

```
// This is the name of our package
// Everything with this package name can see everything
// else inside the same package, regardless of the file they are in
package main

// These are the libraries we are going to use
// Both "fmt" and "net" are part of the Go standard library
import (
    "encoding/json"
    // "fmt" has methods for formatted I/O operations (like printing to the console)
    "fmt"
    // The "net/http" library has methods to implement HTTP clients and servers
    "net/http"
    "github.com/gorilla/mux"
)

type Department struct {
    Name      string `json:"name"`
    Description string `json:"description"`
}

var departments []Department

func getDepartmentHandler(w http.ResponseWriter, r *http.Request) {
    departments, err := GetDepartments()
    // Convert the "departments" variable to json
    departmentListBytes, err := json.Marshal(departments)

    // If there is an error, print it to the console, and return a server
    // error response to the user
    if err != nil {
        fmt.Println(fmt.Errorf("Error: %v", err))
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
}
```

Type here to search

00:20 22/01/2021 ENG 18



Test all dependencies before release

```
F:\go\src\ciscows\day5\dentalapp>go test all
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\apexlogs\apexlogs.go:13:2: cannot find package "github.com/apex/logs" in any of:
    D:\Go\src\github.com\apex\logs (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\apex\logs (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\apexlogs\apexlogs.go:10:2: cannot find package "github.com/tj/go-buffer" in any of:
    D:\Go\src\github.com\tj\go-buffer (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\tj\go-buffer (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\cli\cli.go:12:2: cannot find package "github.com/fatih/color" in any of:
    D:\Go\src\github.com\fatih\color (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\fatih\color (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\Log\handlers\delta\delta.go:13:2: cannot find package "github.com/aybabtme/rgbterm" in any of:
    D:\Go\src\github.com\aybabtme\rgbterm (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\aybabtme\rgbterm (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\delta\delta.go:14:2: cannot find package "github.com/tj/go-spin" in any of
:
    D:\Go\src\github.com\tj\go-spin (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\tj\go-spin (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\es\es.go:11:2: cannot find package "github.com/tj/go-elastic/batch" in any of:
    D:\Go\src\github.com\tj\go-elastic\batch (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\tj\go-elastic\batch (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\graylog\graylog.go:6:2: cannot find package "github.com/aphistic/golf" in any of:
    D:\Go\src\github.com\aphistic\golf (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\aphistic\golf (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\kinesis\kinesis.go:8:2: cannot find package "github.com/aws/aws-sdk-go/aws" in any of:
    D:\Go\src\github.com\aws\aws-sdk-go\aws (from $GOROOT)
    C:\Users\Balasubramaniam\go\src\github.com\aws\aws-sdk-go\aws (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\kinesis\kinesis.go:9:2: cannot find package "github.com/aws/aws-sdk-go/aws"
```



Performing Static Analysis

```
F:\go\src\ciscows\day5\dentalapp>go vet .
F:\go\src\ciscows\day5\dentalapp>go vet main.go
# command-line-arguments
vet.exe: ./main.go:26:22: undeclared name: GetDepartments

F:\go\src\ciscows\day5\dentalapp>go vet store.go
# command-line-arguments
vet.exe: ./store.go:27:36: undeclared name: Department

F:\go\src\ciscows\day5\dentalapp>
```



Distribution List

Administrator: Command Prompt

```
F:\go\src\ciscows\day5\dentalapp>go tool dist list
aix/ppc64
android/386
android/amd64
android/arm
android/arm64
darwin/amd64
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
freebsd/arm64
illumos/amd64
js/wasm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
linux/mips64
linux/mips64le
linux/mipsle
linux/ppc64
linux/ppc64le
linux/riscv64
linux/s390x
netbsd/386
netbsd/amd64
netbsd/arm
netbsd/arm64
openbsd/386
```





Diagnosing Problems and Making Optimizations

- A nice feature of Go is that it makes it easy to benchmark your code.
- If you're not familiar with the general process for writing benchmarks there are good guides [here](#) and [here](#).
- To run benchmarks you'll need to use the go test tool, with the -bench flag set to a regular expression that matches the benchmarks you want to execute



Diagnosing Problems and Making Optimizations

```
F:\go\src\ciscows\day5\dentalapp>go test -bench=. ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      1.251s
F:\go\src\ciscows\day5\dentalapp>
```

```
F:\go\src\ciscows\day5\dentalapp>go test -bench=. ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      1.251s
F:\go\src\ciscows\day5\dentalapp>go test -bench=. -benchmem ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      0.251s
F:\go\src\ciscows\day5\dentalapp>
```

-benchmem flag, which forces memory allocation statistics to be included in the output.



Profiling

- Go makes it possible to create diagnostic profiles for CPU use, memory use, goroutine blocking and mutex contention.
- You can use these to dig a bit deeper and see exactly how your application is using (or waiting on) resources.



Profiling

- There are three ways to generate profiles:
- If you have a web application you can import the `net/http/pprof` package.
- This will register some handlers with the `http.DefaultServeMux` which you can then use to generate and download profiles for your running application.
- This post provides a good explanation and some sample code.
- For other types of applications, you can profile your running application using the `pprof.StartCPUProfile()` and `pprof.WriteHeapProfile()` functions. See the `runtime/pprof` documentation for sample code.
- Or you can generate profiles while running benchmarks or tests by using the various `-***profile` flags like so:



Profiling

- \$ go test -run=^\$ -bench=^BenchmarkFoo\$ - memprofile=/tmp/memprofile.out .
- \$ go test -run=^\$ -bench=^BenchmarkFoo\$ - blockprofile=/tmp/blockprofile.out .
- \$ go test -run=^\$ -bench=^BenchmarkFoo\$ - mutexprofile=/tmp/mutexprofile.out .
- go test -run=^\$ -bench=^BenchmarkFoo\$ - cpuprofile=cpuprofile.out .
- go tool pprof -http=:5500 cpuprofile.out



Tracing

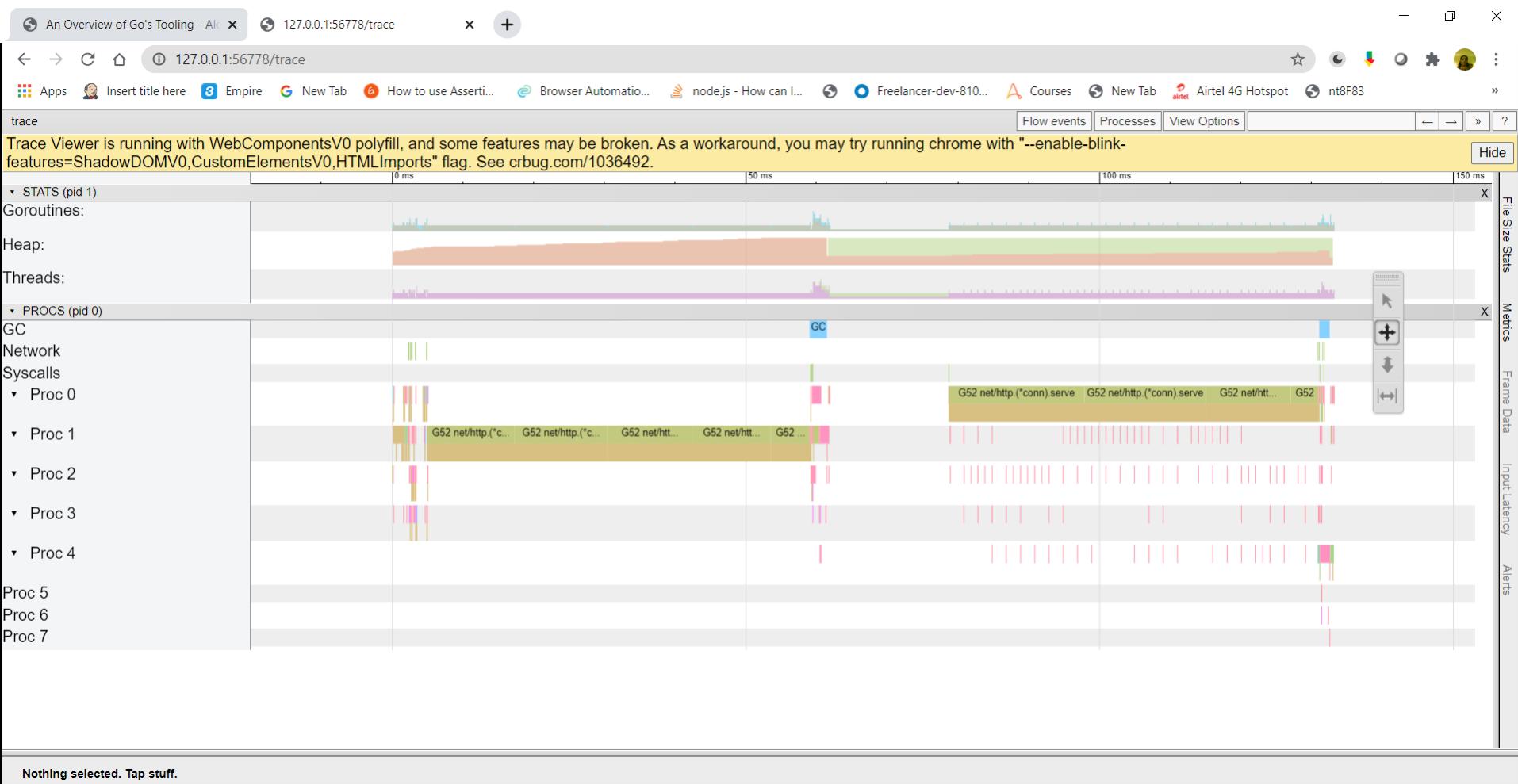
```
F:\go\src\ciscows\day5\dentalapp>go tool trace trace.out
2021/01/22 00:43:14 Parsing trace...
2021/01/22 00:43:14 Splitting trace...
2021/01/22 00:43:14 Opening browser. Trace viewer is listening on http://127.0.0.1:56872

F:\go\src\ciscows\day5\dentalapp>go test -run=^$ -bench=^BenchmarkFoo$ -trace=trace.out .
PASS
ok      ./F_/go/src/ciscows/day5/dentalapp      0.302s

F:\go\src\ciscows\day5\dentalapp>go tool trace trace.out
2021/01/22 00:43:42 Parsing trace...
2021/01/22 00:43:42 Splitting trace...
2021/01/22 00:43:42 Opening browser. Trace viewer is listening on http://127.0.0.1:56917
```



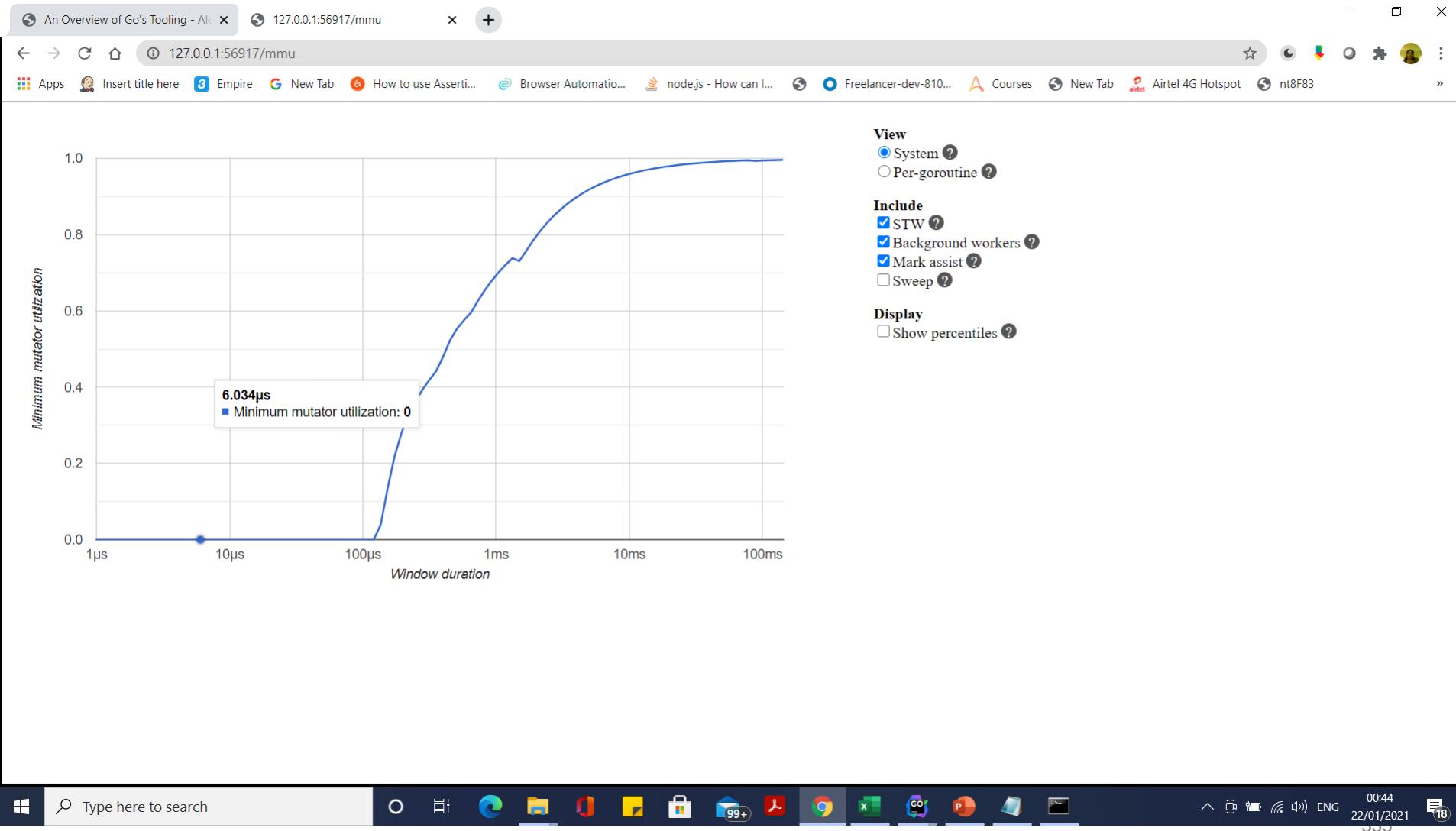
Tracing



Nothing selected. Tap stuff.



Tracing





cgo

- Writing the C code
- Create a new folder in your \$GOPATH/src. I've called it cgo-tutorial. Inside this folder, create a header file called greeter.h with the following content:
 - #ifndef _GREETER_H
 - #define _GREETER_H
- int greet(const char *name, int year, char *out);
- #endif



cgo

- Writing the C code
- Create a new folder in your \$GOPATH/src. I've called it cgo-tutorial. Inside this folder, create a header file called greeter.h with the following content:
 - #ifndef _GREETER_H
 - #define _GREETER_H
- int greet(const char *name, int year, char *out);
- #endif



cgo

- `#include "greeter.h"`
- `#include <stdio.h>`
- `int greet(const char *name, int year, char *out) {`
- `int n;`
-
- `n = sprintf(out, "Greetings, %s from %d! We come in peace :)", name, year);`
- `return n;`
- }



cgo

- Now we can run `gcc -c greeter.c` to make sure that our library actually compiles.
- Writing the go code
- Create a file called `main.go`, and add the following to the top:
 - `package main`
 - `/ #cgo CFLAGS: -g -Wall`
 - `// #include <stdlib.h>`
 - `// #include "greeter.h"`
 - `import "C"`
 - `import (`
 - `"fmt"`
 - `"unsafe"`
 - `)`



cgo

- func main() {
- name := C.CString("Gopher")
- defer C.free(unsafe.Pointer(name))
- year := C.int(2018)
- }

Questions





Module Summary

- In this module we discussed
 - Overview of Maven
 - Maven archetypes
 - Maven life cycle phases
 - The pom.xml file
 - Creation of Java projects using Maven
 - Creation of war files

