MEAN – Go – RESTful APIs Full - Stack Web Development Training

Day 4

Go – Why?

Go - Basics

- Variables
- Primitives
- Constants
- Arrays and Slices
- Maps and Structs
- If and Switch Statements
- Looping
- Defer, Panic, and Recover
- Pointers
- Functions
- Interface
- Goroutines
- Channels

Go – Variables

- Variable declarations
  - var foo int
  - var foo int = 42
  - foo := 42
- Can’t redeclare variables, but can shadow them
- All variables must be used
- Naming conventions
  - Pascal or CamelCase
    - Capitalize acronyms (HTTP, URL)
  - As short as reasonable
    - Longer names for longer lives
- Visibility
  - Lower case first letter for package scope
  - Upper case first letter to export
  - No private scope
- Type conversions
  - destinationType(variable)
  - User strconv package for strings

Go - Primitives

- Boolean types
  - Values are true or false
  - Not an alias for other types (e.g. int)
  - Zero value is false
- Numeric types
  - Integers
    - Can't mix types in same family (uint8 + uint16 = error)
    - Bitwise operations – AND, OR, XOR, and NOT
    - Arithmetic operations – add, sub, mult, div
    - Unsigned integers - 8 bit(byte and uint8) – 32 bit (uint32)
    - Signed integers - Int type has varying size, but min 32 bits, 8 bit(int8) – 64 bit(int64)
  - Floating point
    - Arithmetic operations
    - 32 bit and 64 bit versions
    - Literal styles - Decimal(3.14), Exponential(13e18 or 2E10), Mixed(13.7e12)
  - Complex numbers
    - Zero value is (0 + 0i)
    - 64 and 128 bit versions
    - Built-in functions – complex, real, imag
    - Arithmetic operations
- Text types
  - Rune – UTF-8, alias for int32, special methods normally required to process
  - String – UTF-8, Immutable, Concatenated with (+) operator, convertible to []byte

Go - Constants

- Naming convention
- Typed constants
- Untyped constants
- Enumerated constants
- Enumeration expressions

Go – Constants (Summary)

- Immutable, but can be shadowed
- Replaced by the compiler at compile time
  - Value must be calculable at compile time
- Named like variables
  - PascalCase for exported constants
  - camelCase for internal constants
- Typed constants work like immutable variables
  - Can interoperate only with same type
- Untyped constants work like literals
  - Can interoperate with similar type
- Enumerated constants
  - iota allows related constants to be created easily
  - iota starts at 0 in each const block and increm. by one
  - Watch out of constant values that match zero values for variables
- Enumerated expressions
  - Operations that can be determined at compile time are allowed – arithmetic, bitwise operations, bitshifting

Go – Arrays and Slices

- Arrays
  - Creation
  - Built-in functions
  - Working with arrays
- Slices
  - Creation
  - Built-in functions
  - Working with slices

Go – Arrays and Slices (Summary)

- Arrays
  - Collection of items with same type
  - Fixed size
  - Access via zero-based index
    - a := [3]int{1, 3, 5} // a[1] == 3
  - len returns size of array
  - Copies refer to different underlying data (independent copy)
  - Declaration styles
    - a := [3]int{1, 2, 3}
    - a := […]int{1, 2, 3}
    - var a [3]int

Go – Arrays and Slices (Summary)

- Slices
  - Backed by array
  - Creation styles
    - Slice existing array or slice
    - Literal style
    - Via make function
      - A := make([]int, 10) // length == capacity == 10
      - A := make([]int, 10, 100) //length == 10, capacity == 100
  - len returns length of slice
  - cap returns length of underlying array
  - append adds elements to slice
    - May cause expensive copy operation if underling array is too small

Go – Maps and Structs

- Maps
  - What are they?
  - Creating
  - Manipulation
- Structs
  - What are they?
  - Creating
  - Naming conventions
  - Embedding
  - Tags

Go – Maps and Structs (Summary)

- Maps
  - Collections of value types that are accessed via keys
  - Created vie literals or via make
  - Members accessed via [key] syntax
    - myMap[“key”] = “value”
  - Check for presence via “value, ok
  - Multiple assignments refer to same underlying data

- Structs
  - Collections of different data types that describe a single concept
  - Keyed by named fields
  - Normally created as types, but anonymous structs are allowed
  - Structs are value types
  - No inheritance, but can use composition via embedding
  - Tags can be added to struct fields to describe field

Go – If and Switch Statements

- IF statements
  - Operators
  - If – else and if – else statements
- Switch statements
  - Simple cases
  - Cases with multiple tests
  - Falling through
  - Type switches

Go – If and Switch Statements (Summary)

- IF statements
  - Initializer
  - Comparison operators
  - Logical operators
  - Short circuiting
  - If – else statements
  - If – else if statements
  - Equality and floats
- Switch statements
  - Switching on a tag
  - Cases with multiple tests
  - Initializers
  - Falling through
  - Switches with no tag
  - Type switches
  - break

Go – Looping

- for statements
  - Simple loops
  - Exiting early
  - Looping through collections

Go – Looping (Summary)

- for statements
  - Simple loops
    - for initializer; test; incrementer {}
    - for test {}
    - for {}
  - Exiting early
    - break
    - continue
    - labels
  - Looping through collections
    - arrays, slices, maps, strings, channels
    - for k, v := range collection {}

Go – Defer, Panic, Recover

- defer
  - Used to delay execution of a statement until function exits
  - Useful to group "open" and "close" functions together
    - Be careful in loops
  - Run in LIFO order
  - Arguments evaluated at time defer is executed, not at a time of call
- panic
  - When application ends in shutdown, things it cannot continue at all
    - Don't use it when file can't be opened, unless it is critical
    - Use for unrecoverable events – cannot obtain TCP port for web server
  - Function will stop executing
    - Deferred functions will still fire
  - If nothing handles panic, program will exit
- Recover
  - Used to recover from panics
  - Only useful in deferred functions
  - Current function will not attempt to continue, but higher functions in call stack will

Go – Pointers

- Creating pointers
- Dereferencing pointers
- The new functions
- Working with nil
- Types with internal pointers

Go – Functions

- Basic Syntax
- Parameters
- Return values
- Anonymous functions
- Functions as types
- Methods

Go – Functions (Summary)

- Basic Syntax
  - func foo() { … }
- Parameters
  - Comma delimited list of variables and types
    - func foo(bar string, baz int)
  - Params of the same type can be typed once
    - func foo(bar, baz int)
  - When pointers are passed in, the function can change the values in the caller
    - Always true for slices, maps
  - Use variadic params to send list of the same types in
    - Must be last param
    - Received as a slice
    - Func foo (bar string, baz …int) // baz contains the rest in int

Go – Functions (Summary)

- Return values
  - Single return values just list type
    - func foo() int
  - Multiple return value list types surrounded by parentheses
    - func foo() (int, error)
    - The (result type, error) paradigm is a very common idiom
  - Can use named return values
    - Initializes returned var
    - Return using return keyword on its own
  - Can return addresses of local var
    - Automatically promoted from local memory (stack) to shared memory (heap)

Go – Functions (Summary)

- Anonymous functions
  - Function don’t have names if they are:
    - Immediately invoked
      - func() { … } ()
    - Assigned to a variable or passed as an argument to a function
      - a := func() { … } a()

Go – Functions (Summary)

- Functions as types
  - Can assign function to var or use as arguments and return values in functions
  - Type signature is like function signature, with no param names
    - var f func(string, string, int) (int, error)

- Methods
  - Functions that executes in context of a type
  - Format
    - Func (g greeter) greet() { … }
  - Receiver (g greeter) can be value or pointer
    - Value receiver gets copy of type
    - Pointer receiver gets pointer to type

Go – Interface

- Basics
- Composing interfaces
- Type conversion
  - The empty interface
  - Type switches
- Implementing with values vs. pointers
- Best practices

Go – Interface (Best Practices)

- Use many, small interfaces
  - Single method interfaces are some of the most powerful and flexible
    - io.Writer, io.Reader, interfaces{}
- Don’t export interfaces for types that will consumed
- Don’t export interfaces for types that will be used by package
- Design functions and methods to receive interfaces whenever possible

Go – Interface (Summary)

- Basics

```go
type Writer interface {
  Write([]byte) (int, error)
}

type ConsoleWriter struct {}

//implementation
func(cw ConsoleWriter) Write(data []byte) (int, error) {
  n, err := fmt.Println(string(data))
  return n,err
}
```

Go – Interface (Summary)

- Composing interfaces

```
type Writer interface {
  Write([]byte)(int, error)
}

type Closer interface {
  Close() error
}

type WriterCloser interface {
  Writer
  Closer
}
```

Go – Interface (Summary)

- Type conversion

```
var wc WriterCloser = NewBufferedWriterCloser()
[redacted]

// --- interface conversion
bwc, ok := wc.(*BufferedWriterCloser)
```

- The empty interface and type switches

```
var i interface{} = '0' // 0 "0" '0' true
switch i.(type) {
case int:
  fmt.Println("i is an integer")
case string:
  fmt.Println("i is a string")
default:
  fmt.Println("i is unknown")
```

Go – Interface (Summary)

- Implementing with values vs. pointers
  - Method set of ***values*** is all methods with value receivers
  - Method set of ***pointer*** is all methods, regardless of receiver type

- Best practices

Go – Goroutines

- Creating Goroutines
- Paralellism
- Best Practices
- Synchronization
  - WaitGroups
  - Mutexes

Go – Goroutines (Best Practices)

- Don’t create goroutines in libraries
  - Let consumer control concurrency
- When creating a goroutine, know how it will end
  - Avoid subtle memory leaks
- Check for race conditions at compile time
  - go run –race main.go

Go – Goroutines (Summary)

- Creating goroutines
  - Use keyword ***go*** in front of function call
  - When using anonymous functions, pass data as local variables
- Synchronization
  - Use sync.WaitGroup to wait for groups of goroutines to complete
  - Use sync.mutex and sync.RWMutex to protect data access
  - https://stackoverflow.com/questions/54982948/why-sync-mutex-exists
- Parallelism
  - By default, Go will use CPU threads equal to available cores
  - Change with runtime.GOMAXPROCS
  - More threads can increase performance, but too many can slow it down
- Best Practices

Go – Channels

- Basics
- Restricting data flow
- Buffered channels
- Closing channels
- for …range loops with channels
- Select statements

Go – Channels (Summary)

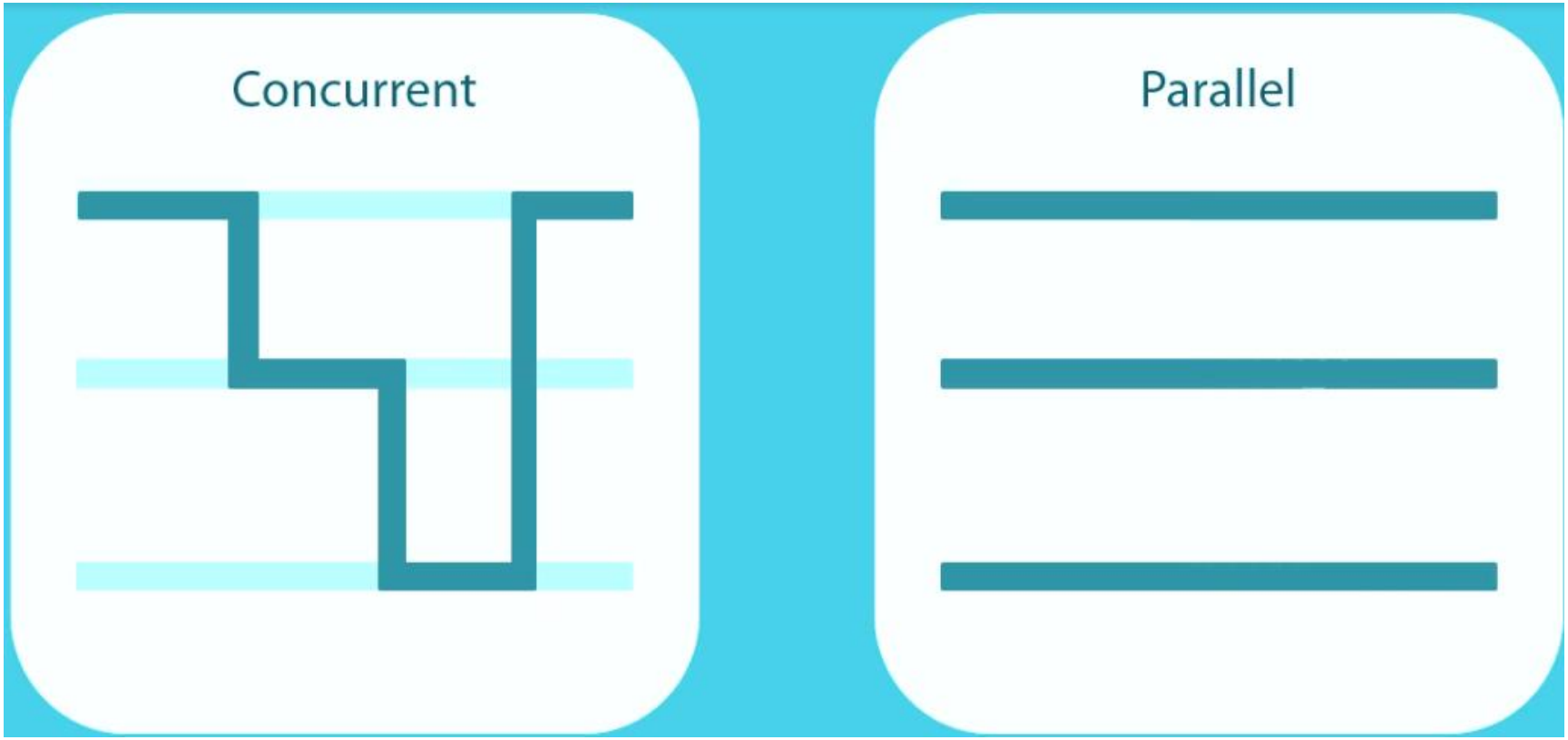
- Basics
  - Create a channel with make command
    - make(chan int)
  - Send message into channel
    - ch <- val
  - Receive message from channel
    - val := <-ch
  - Can have multiple senders and receivers
- Restricting data flow
  - Channel can be cast into send-only or receive only versions
    - Send-only: chan <- int
    - Receive –only: <-chan int
- Buffered channels
  - Channels block sender side until receiver is available
  - Block receiver side until message is available
  - Can decouple sender and receiver with buffered channels
    - Make(chan int, 50)
  - Use buffered channels when send and receiver have asymmetric loading
- for …range loops with channels
  - Use to monitor channel and process messages as they arrive
  - Loop exits when channel is closed
- Select statements
  - Allows goroutine to monitor several channels at once
    - Blocks if all channel block
    - If multiple channels receive value simultaneously, behavior is undefined

Go - Reference

- https://gobyexample.com/

Go – Concurrency

- Concurrency vs. Parallelism



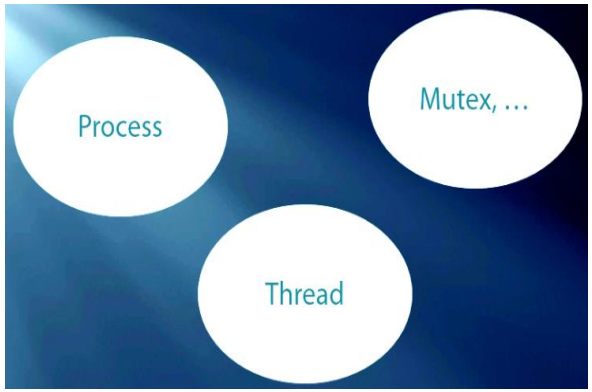
Go – Concurrency

- Concurrency Models
  - Processor threads
  - Events
  - Callbacks and Promises
  - Communicating Sequential Process

Go – Concurrency

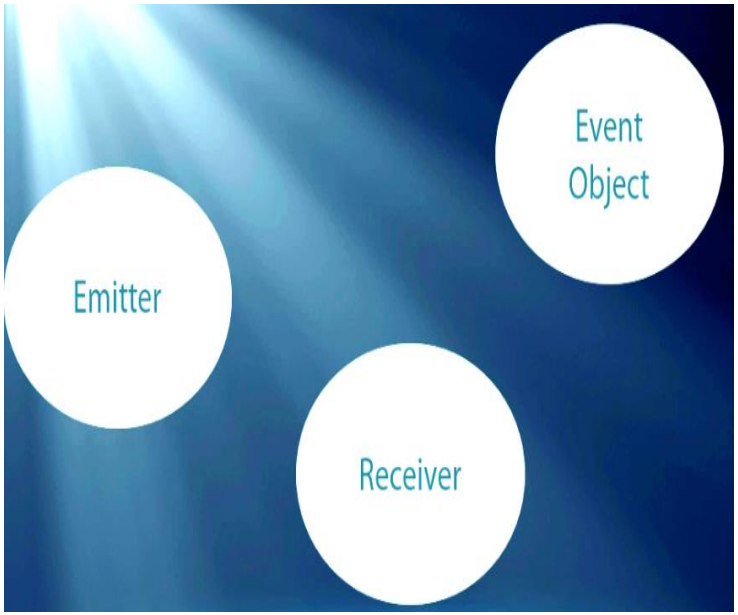
- Processor threads
  - Lowest level, therefore highest control
- Process : execution of a single instance of an application
- Threads : process contains one or more threads, sequence of programming instructions running in order
- Mutex : to control how many threads are allowed to access certain piece of code at one time

- Advantages
  - Control, Responsive UI, Performance (maybe)
- Advantages
  - Poor performance, memory consumption, shared memory, race conditions



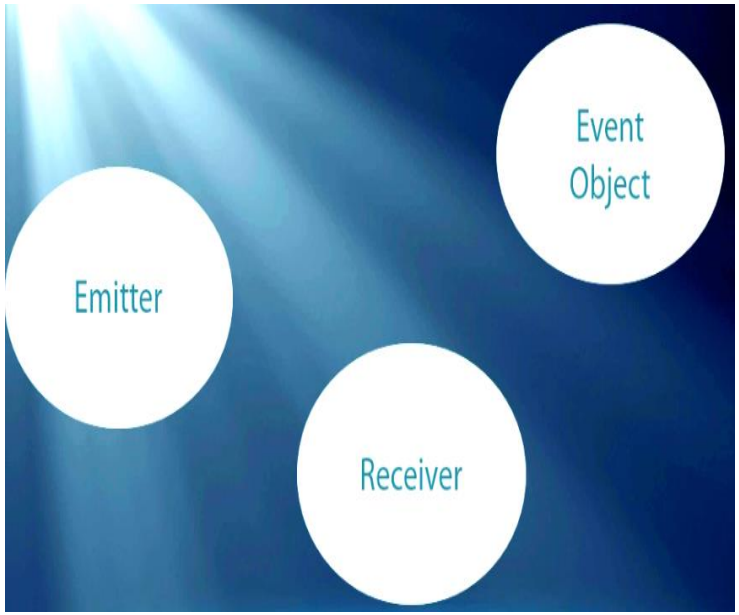
Go – Concurrency

- Events
  - Emitter : generates an event in response to some external stimulus
  - Receiver : responds to event that emitter sends out
  - Event object: an object for communication between emitter and receiver
- Advantages
  - Memory isolation
  - Sepearates callee from caller
- Disadvantages
  - Hard to trace
  - Hard to synchronize receivers



Go – Concurrency

- Callback
  - Callback: a receiver for an action needs to be taken after async. task is completed
  - Promise: advanced callback, register several functions to be completed
- Advantages
  - Memory isolation
  - Simplify Async. operation
- Disadvantages
  - Nested callbacks
  - HardHandling multiple receivers



Go – Concurrency

- Communication Sequential Process (CSP)
  - Actor: Receiving info, process it, and passing the result to the next actor, relatively simple, no need to worry about async. concurrency
  - Message: object that is passed between two actors, after passed, the info is not accessible to the sender

- Advantages
  - Fully decoupled from other actors
  - Able to have multiple handlers
  - Memory isolation
- Disadvantages
  - Complicated Mental model
  - Hard to trace

Go – Concurrency

- No Thread Primitives
- Goroutines
- Channels (communication between goroutines)

Go – Concurrency (Goroutines)

- Light, scalable, schedule-able
- Memory requirements: processor thread 1MB, green thread 2KB
  - Goroutine is virtual thread, uses less than 2kb, can grow based on needs
- Parallelism : GOMAXPROCS

  Sets max. number of CPUs that can be executing simultaneously and returns the previous setting (>1 = more cores)

Go – Concurrency (Goroutines)

- Demo #1 Basic

src ▸ tutorial ▸ 15concurrency ▸ 01basicgoroutine ▸ main.go ▸ {} main ▸ main

```go
package main

import (
  //"fmt"
  "time"
  "runtime"
 )

 func main() {

  godur, _ := time.ParseDuration("10ms")
  runtime.GOMAXPROCS(2) // > 1 = parallelism, no more in order

   go func() {
     for i := 0; i < 100; i++ {
       println("Hello")
       time.Sleep(godur) // alternate
     }
   }()
   go func() {
    for i := 0; i < 100; i++ {
      println("Go")
      time.Sleep(godur) // alternate
    }
  }()

  dur, _ := time.ParseDuration("1s")
  time.Sleep(dur) // otherwise, no output, goroutine func main runs too fast and destroy the rest

 }
```

Go – Concurrency (Goroutines)

- Demo #2 Async Web Services

src ▸ tutorial ▸ 15concurrency ▸ 02asyncwebservices ▸ main.go

```go
package main

import (
    "net/http"
    "io/ioutil"
    "encoding/xml"
    "fmt"
    "time"
    "runtime"
)

func main() {

    runtime.GOMAXPROCS(4) // try paralellism

    start := time.Now()

    stockSymbols := []string{
        "googl",
        "msft",
        "aapl",
        "bbry",
        "hpq",
        "vz",
        "t",
        "tmus",
        "s",
    }

    numComplete := 0
```

src ▸ tutorial ▸ 15concurrency ▸ 02asyncwebservices ▸ main.go ▸ {} main ▸ main

```go
    numComplete := 0

    for _, symbol := range stockSymbols {
        go func(symbol string) { // faster web service looking up quotes with go routines
            resp, _ := http.Get("http://dev.markitondemand.com/MODApis/Api/v2/Quote?symbol=" + symbol)
            defer resp.Body.Close()
            body, _ := ioutil.ReadAll(resp.Body)

            quote := new(QuoteResponse)
            xml.Unmarshal(body, &quote)
            fmt.Printf("%s: %.2f\n", quote.Name, quote.LastPrice)
            numComplete++
        }(symbol)
    }

    for numComplete < len(stockSymbols) {
        time.Sleep(10 * time.Millisecond)
    }

    elapsed := time.Since(start)
    fmt.Printf("Execution time: %s", elapsed) //let's make a benchmark for a service call
}

type QuoteResponse struct{
```

Go – Concurrency (Goroutines)

- Demo #2 Async Web Services

src ▸ tutorial ▸ 15concurrency ▸ 02asyncwebservices ▸

```
53  type QuoteResponse struct{
54      Status string
55      Name string
56      LastPrice float32
57      Change float32
58      ChangePercent float32
59      TimeStamp string
60      MSDate float32
61      MarketCap int
62      Volume int
63      ChangeYTD float32
64      ChangePercentYTD float32
65      High float32
66      Low float32
67      Open float32
68  }
```

Go – Concurrency (Goroutines)

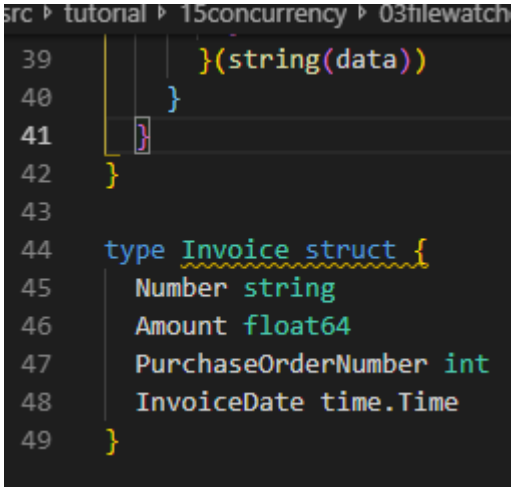
- Demo #3 File Watcher

src ▸ tutorial ▸ 15concurrency ▸ 03filewatcher ▸ main.go ▸ {} main ▸ main

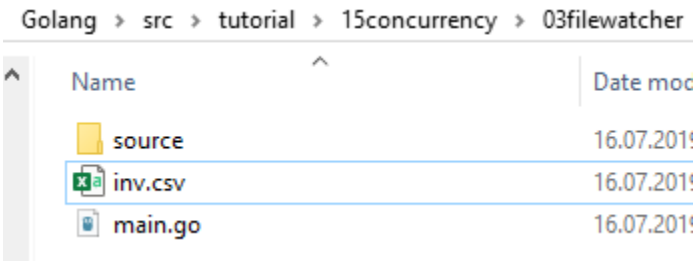
```go
package main

import (
  "fmt"
  "os"
  "time"
  "io/ioutil"
  "strings"
  "encoding/csv"
  "strconv"
)

const watchedPath = "./../03filewatcher/source"

func main() {
  for {
    d, _ := os.Open(watchedPath)
    files, _ := d.Readdir(-1)
    for _, fi := range files {
      filePath := watchedPath + "/"+ fi.Name()
      f, _ := os.Open(filePath)
      data, _ := ioutil.ReadAll(f)
      f.Close()
      os.Remove(filePath)

      go func(data string) {
        reader := csv.NewReader(strings.NewReader(data))
        records, _ := reader.ReadAll()
        for _, r := range records {
          invoice := new(Invoice)
          invoice.Number = r [0]
          invoice.Amount, _ = strconv.ParseFloat(r[1], 64)
          invoice.PurchaseOrderNumber, _ = strconv.Atoi(r[2])
          unixTime, _ := strconv.ParseInt(r[3], 10, 64)
          invoice.InvoiceDate = time.Unix(unixTime, 0)

          fmt.Printf("Received invoice '%v' for $%.2f and submitted\n", invoice.Number, invoice.Amount)
        }
      }(string(data))
```

Go – Concurrency (Goroutines)

- Demo #3 File Watcher




```
01-1234,42.27,0,0
01-8888,14250.49,0,0
```



Go – Concurrency (Channels)

- Communication, provide safe way to to send message through your app
- Can be distributed to actors without them knowing about each others (decoupled architecture, easy to test)
- Way to isolate, the sender has no access to the data in the receiver
- No sender or receiver must wait until te other one is ready due to channel‘s internal synch.

Go – Concurrency (Channels)

- Demo #1 Basic Channels

rc ▸ tutorial ▸ 15concurrency ▸ 04basicchannel ▸ main.go ▸ {} main ▸ main

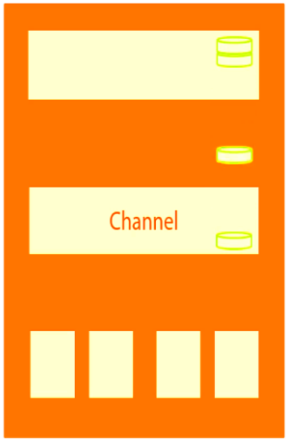
```go
package main

import(
  "fmt"
)

func main() {
  ch := make (chan string, 1) // without 1, it will wait, until deadlock
  ch <- "Hello"

  fmt.Println(<-ch)
}
```

Go – Concurrency (Channels)

- Demo #2 Buffered Channels

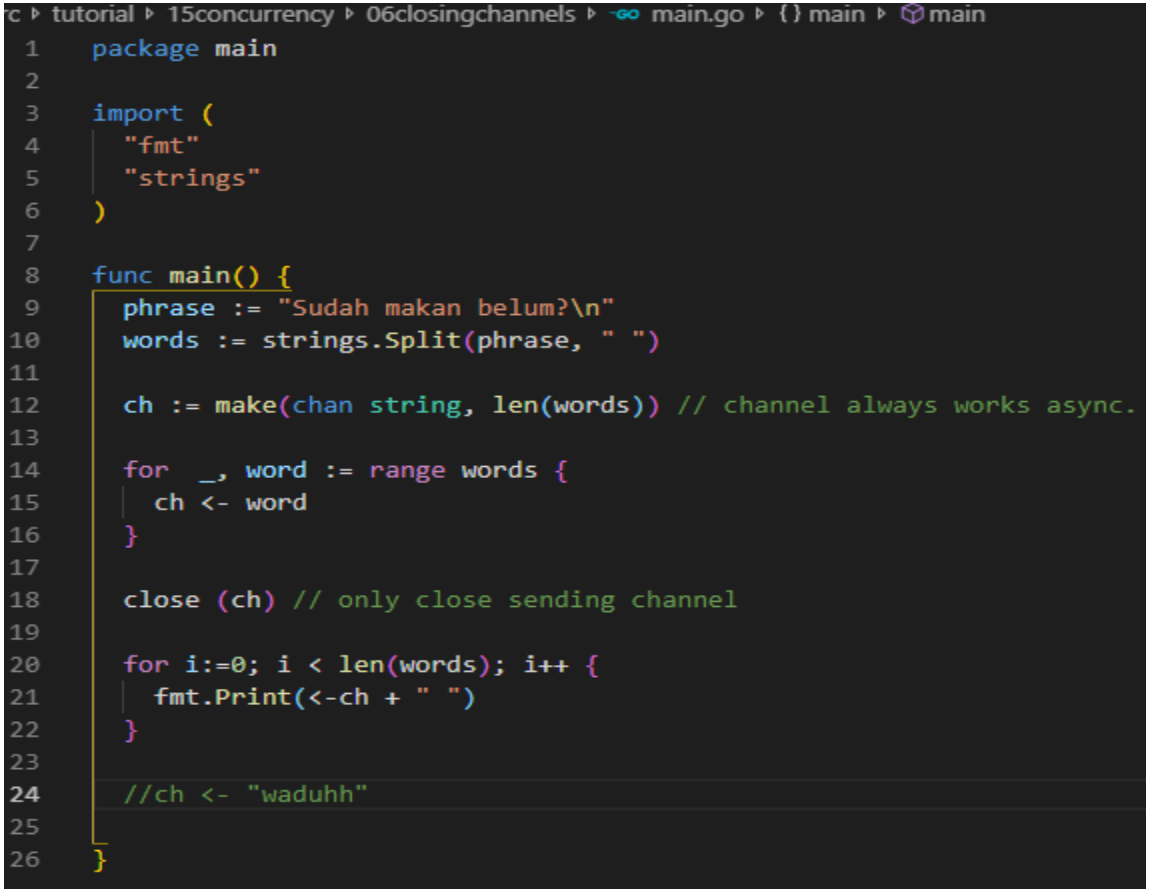
src ▸ tutorial ▸ 15concurrency ▸ 05bufferedchannel ▸ main.go ▸ {} main ▸ main

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    phrase := "These are the times that try men's souls. \n"
    words := strings.Split(phrase, " ")

    ch := make(chan string, len(words)) // channel always works async.

    for  _, word := range words {
        ch <- word
    }

    for i:=0; i < len(words); i++ {
        fmt.Print(<-ch + " ")
    }

}
```



Go – Concurrency (Channels)

- Demo #2 Closing Channels



Go – Concurrency (Channels)

- Demo #3 Ranging Over a Channel

tutorial ▸ 15concurrency ▸ 07rangingoverchannel ▸ main.go ▸ {} main ▸ main

```go
package main

import (
  "fmt"
  "strings"
)

func main() {
  phrase := "Sudah makan belum?\n"
  words := strings.Split(phrase, " ")

  ch := make(chan string, len(words)) // channel always works async.

  for  _, word := range words {
    ch <- word
  }

  close(ch) // close the channel so that the next open loop will not wait, forever
/*
  for {
    if msg, ok := <- ch; ok {
      fmt.Print(msg + " ")
    } else {
      break
    }
  }

*/
  // more elegant using range, no open loop
  for msgA := range ch {
    fmt.Print(msgA + " ")
  }

}
```

Go – Concurrency (Channels)

- Demo #4 Switching between Channels

src ▸ tutorial ▸ 15concurrency ▸ 08switchingchannels ▸ main.go ▸ {} main ▸ main

```go
package main

import (
  "fmt"
)

func main(){
  msgCh := make(chan Message, 1)
  errCh := make(chan failedMessage, 1)

  msg := Message{
    To: []string{"admin@pramuka.co.id"},
    From: "bisaaja@kotalama.org",
    Content: "Rahasia Orang Dalam",
  }

  failedMessage := failedMessage{
  ErrorMessage: "Message intercepted by Ronda Malam",
  OriginalMessage: Message{},
  }

  msgCh <- msg
  errCh <- failedMessage // this will be detected by select block , FILO
  /*
  fmt.Println(<-msgCh)
  fmt.Println(<-errCh)
  */

  // use select block
  select {
```

src ▸ tutorial ▸ 15concurrency ▸ 08switchingchannels ▸ main.go ▸ {}

```go
  // use select block
  select {
  case receivedMsg := <- msgCh:
    fmt.Println(receivedMsg)
  case receivedError := <- errCh:
    fmt.Println(receivedError)
    default: fmt.Println("No message received")
  }

}

type Message struct {
  To []string
  From string
  Content string
}

type failedMessage struct{
  ErrorMessage string
  OriginalMessage Message
}
```

Go – Concurrency (Goroutines & Channels)

- Pipe and Filter
- Extract, Transform and Load

Go – Concurrency (Goroutines & Channels)

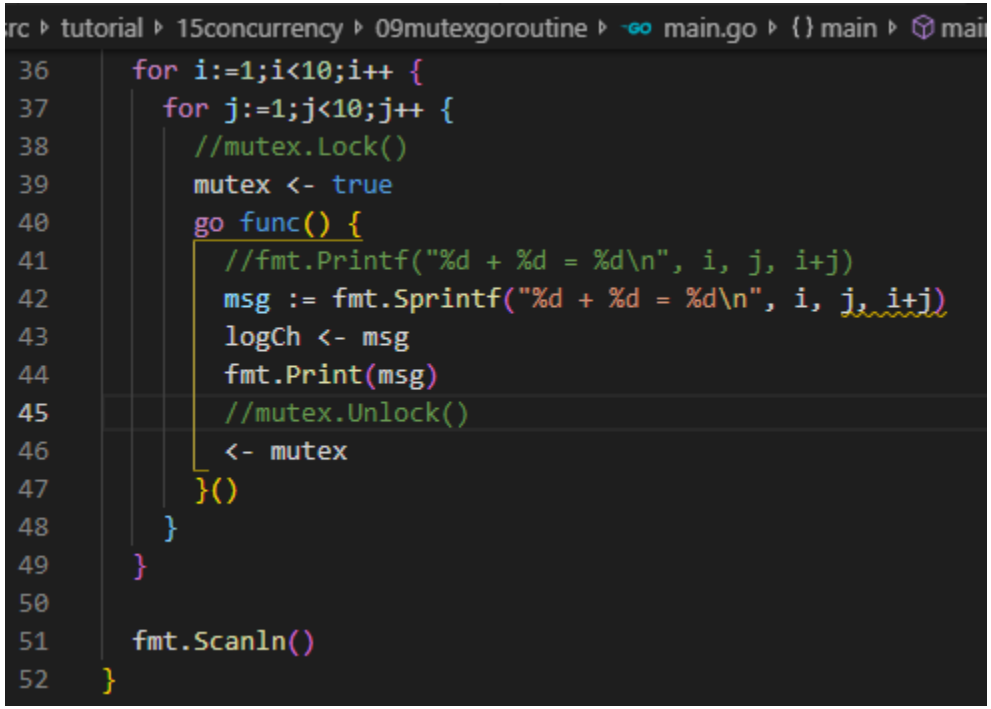
- DEMO #1 Mutex Lock + Goroutine

src ▸ tutorial ▸ 15concurrency ▸ 09mutexgoroutine ▸ main.go ▸ {} main ▸ main

```go
package main

import (
  "fmt"
  "runtime"
  //"sync"
  "os"
  "time"
)

func main() {
  runtime.GOMAXPROCS(4)
  //mutex := new(sync.Mutex) // with core = 4 and locking , uncloking, this progams runs slower than single threaded one
  mutex :=make(chan bool, 1) // mutex with channel, but confusing for other dev.

  f,  _ := os.Create("./log.txt")
  f.Close()

  logCh := make(chan string, 50)

  go func(){
    for {
      msg, ok := <- logCh
      if ok {
        f,  _ := os.OpenFile("./log.txt", os.O_APPEND, os.ModeAppend)

        logTime := time.Now().Format(time.RFC3339)
        f.WriteString(logTime + " - " + msg)
        f.Close()
      } else {
        break
      }
    }
  }()

  for i:=1;i<10;i++ {
```

Go – Concurrency (Goroutines & Channels)

- DEMO #1 Mutex Lock + Goroutine



Go – Concurrency (Goroutines & Channels)

- DEMO #2 Events

Go – Concurrency (Goroutines & Channels)

- DEMO #3 Callbacks

Go – Concurrency (Goroutines & Channels)

- DEMO #3 Promises

Go – Concurrency (Goroutines & Channels)

- DEMO #4 Pipe, and Filter Pattern

Go – Concurrency (Goroutines & Channels)

- DEMO #5 Extract, Transform, and Load

Go – Concurrency (Goroutines & Channels)

- DEMO #5 Extract, Transform, and Load

Go – Concurrency

- No Thread Primitives
- Goroutines
- Channels (communication between goroutines)

Go – MongoDB

- Installation and Connecting to DB

https://github.com/tfogo/mongodb-go-tutorial

https://github.com/mongodb/mongo-go-driver#installation

if you do not have "dep" to set up (https://github.com/golang/dep);

```
go get -u github.com/golang/dep/cmd/dep
```

Option 1:

If you want to install the **mongo-db driver** on the **system** (I think use **"dep"**, so use with **option 2**), use the following commands;

```
go get -v -u go.mongodb.org/mongo-driver
cd $GOPATH/src/go.mongodb.org/mongo-driver/
git checkout -b v1.0.1 v1.0.1
dep ensure
dep status
```

Option 2:

"dep" controls project **dependencies**. If you have a project that you have **previously created** and **"import"**, when you use **"dep init"** to the **root folder of your project**, it adds the related dependencies to the "vendor" folder, which is also included in your project and created automatically. So it can be easily moved to other environments.

(dep ensure -add "go.mongodb.org/mongo-driver/mongo@~1.0.1") adds **mongodb dependencies to your project,** which was previously **"dep init"**. You enter into your project (must have at **least 1 ".go"** file) and run the use commands:

```
mkdir $GOPATH/src/mongotest/
cd $GOPATH/src/mongotest/
echo "package main" > main.go
dep init
dep ensure -add "go.mongodb.org/mongo-driver/mongo@~1.0.1"
dep ensure -update -no-vendor -v
```

When you look under the **vendor** folder, you will find that it has added **"mongodb" dependencies**. if you use **"dep ensure -update" and you don't use in project** (import "go.mongodb.org/mongo-driver/mongo"), **it will automatically delete dependencies under vendor.** (If you don't use something in "GO", it is delete :))