Structure

Package

- Collection of go files

Creation:

```
package <NAME>
```

Main Package

- Entry point for application
- Requires a main function as entry

Creation:

```
package main

func main(){}
```

Module

- Collection of go packages

Create module:

```
go mod init <NAME>
```

- Name is github repo by convention

Imports and Common Packages

- Imported packages have to be used!

```
import "<PACKAGE_NAME>"
```

fmt

Printing:

```
import "fmt"

var x string = "abc"
fmt.Println(x)
```

With formatting:

```
fmt.Printf("var: %v", value)
```

Build

`go build <FILE>`

- produces binary

`go run <FILE>`

- compiles and runs in one command

Constants Variables and Basic Data Types

Variables

- Variables have to be used

```
var intNum int
```

Declaration options:

```
var b1 bool = true // typed declaration with initial value
var b2 = true // untyped declaration with initial value
var b3 bool // typed declaration without initial value
b4 := true // untyped declaration with initial value

// as above - multiple variables:
var var1, var2 int = 1, 2
var var1, var2 = 1, 2
var var1, var2 int
var1, var2 := 1, 2
```

Constants

- always have to be initialized

```
const myConst string = "const"
```

Basic Data Types

Boolean

```
var t bool = true
```

Integer

```
var intNum int // defaults to 32 or 64 bit depending on system architecture
var intNum8 int8 // -128 to 127
var intNum16 int16 // -32768 to 32767
var intNum32 int32 // -2147483648 to 2147483647
var intNum64 int64 // -9223372036854775808 to 9223372036854775807
```

```
var uintNum uint // defaults to 32 or 64 bit depending on system architecture
var uintNum8 uint8 // 0 to 255
var uintNum16 uint16 // 0 to 65535
var uintNum32 uint32 // 0 to 4294967295
var uintNum64 uint64 // 0 to 18446744073709551615
```

- Integer division always results in a **rounded down** integer

Float

- Default type is `float64` if no type is specified

```
var f32 float32 // -3.4e+38 to 3.4e+38
var f64 float64 // -1.7e+308 to +1.7e+308
```

Rune

- equal to char in C

```
var myRune rune = 'a' // single ticks
```

String

- **immutable**

```
var s string = "Hello!\n" // single line (line break character allowed) - always double tic
var ml string = `Hello!
` // multi line
var
```

- length of string can be calculated via:

```
len("test") // returns byte size!!! utf8!!!
utf8.RuneCountInString("test") // returns actual number of runes
```

Indexing:

```
var myString = "résumé"
var indexed = myString[0] // returns UTF-8 number as uint8
var pitfall = myString[1] // returns only first half of 2 byte character !!!

for i, v := range myString{
    // knows how many bytes each character has -> index 2 and 7 are omitted
}
```

Casting to rune array:

```
var myString = []rune(myString)
```

- `rune` is just an alias for `int32`

String builder:

```
import "strings"

var strSlice = []string{"s", "t", "r", "i", "n", "g"}
var strBuilder strings.Builder
for i:= range strSlice{
    strBuilder.WriteString(strSlice[i])
}
var catStr = strBuilder.String()
```

Typecasting

- Mixed type arithmetics are no allowed

```
var floatNum32 float32 = 10.1
var intNum32 int32 = 2
var result float32 = floatNum32 + float32(intNum32)
```

Default Values

```
var intNum int // defaults to 0
var floatNum float64 // defaults to 0
var myRune rune // defaults to 0
var myString string // defaults to ''
var myBool bool // defaults to false
```

Functions and Control Structures

Functions

```
func printMe(printVal string){
    fmt.Println(printVal)
}

func intDivision(numerator int, denominator int) int {
    var result int = numerator / denominator
    return result
}

func intDivisionWithRemainder(numerator int, denominator int) (int, int) {
    var result int = numerator / denominator
    var remainder int = numerator % denominator
    return result, remainder
}
```

Errors

- Default value is `nil`

Throwing:

```
func intDivisionWithRemainder(numerator int, denominator int) (int, int, error) {
    var err error
    if(denominator==0){
        err = errors.New("Cannot divide by Zero")
        return 0, 0, err
    }
    var result int = numerator / denominator
    var remainder int = numerator % denominator
    return result, remainder
}
```

Catching:

```
var result, remainder, err = intDivisionWithRemainder(0, 0)
if err!=nil{
    fmt.Printf(err.Error())
}
```

Control Structures

- **!!! no ternary operator**

If

```
if true {}

if false {
    // do nothing
}else if true {
    // do sth
}else {}

if false || true{
    // do sth
}

if false && true{
    // do nothing
}
```

Switch

- `break` is implied -> doesn't need to be written explicitly

```
switch{
    case false:
        // do nothing
    case true:
        // do sth
    default:
        // default case
}
```

Conditional switch statements:

```
switch value{
    case 0:
        // do sth
    case 1,2:
        // do sth
    default:
        // do sth
}
```

Goto

Defer

- called after `return` statement

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close() // closes the file after function returns

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close() // closes the file after function returns

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

Emphasis:

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
// returns 2
```

Panic and Recover

When the function F calls panic, execution of F stops, any deferred functions in F are executed normally, and then F returns to its caller. To the caller, F then behaves like a call to panic. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking panic directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

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

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i)) // start panicking and defer
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

Output:

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
```

```
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

Arrays

- fixed length
- all elements have to have the same type
- indexable
- contiguous in memory

```
var intArr [3]int32
var intArr [3]int32 = [3]int32{1,2,3}
var intArr := [3]int32{1,2,3}
var intArr := [...]int32{1,2,3} // infer length
```

Access elements:

```
intArr[0] = 1
intArr[1:3] // elements 1 and 2
```

Iterate:

```
for index, value := range intArr{}
```

Slices

- wrapper around array
- no fixed length
- creates new array every time capacity is exceeded

```
var intSlice []int32 = []int32{4,5,6}
intSlice = append(intSlice, 7) // append element
cap(intSlice) // capacity
```

Spread operator:

```
var intSlice2 []int32 = []int32{8,9}
intSlice = append(intSlice, intSlice2...)
```

Specify length and optionally capacity:

```
var intSlice3 []int32 = make(int32[], 3) // length 3
var intSlice3 []int32 = make(int32[], 3, 8) // length 3, capacity 8
```

Iterate:

```
for index, value := range intSlice3{}
```

Maps

- keys in `[]`
- for a key not in the map it returns value types default value
- second return type `ok`

```
var myMap map[string]uint8 = make(map[string]uint8)
var myMap2 = map[string]uint8{"Adam":23, "Sarah":45}

fmt.Println(myMap2["Adam"])
var age, ok = myMap2["Adam"] // ok is boolean (in the map?)

delete(myMap2, "Adam")
```

Iterate:

```
for name := range myMap2{}

for name, age := range myMap2{}
```

Loops

For-Loop

```
for i<10{
    i = i + 1
}

for {
    if i >=10{
        break
    }
}

for i:=0; i<10; i++ {

}
```

Structs and Interfaces

Structs

```
type gasEngine struct{
    mpg uint8
    gallons uint8
}

func main(){
```

```
    var myEngine gasEngine // defaults to zero valued struct

    var myEngineInit gasEngine = gasEngine{mpg:25, gallons:15}
}
```

Nested Structs

```
type gasEngine struct{
    mpg uint8
    gallons uint8
    ownerInfo owner
}

type owner struct{
    name string
}

func main(){
    var myEngine gasEngine = gasEngine{25, 15 owner{"Alex"}}
    fmt.Println(myEngine.mpg, myEngine.gallons, myEngine.ownerInfo.name)
}
```

OR:

```
type gasEngine struct{
    mpg uint8
    gallons uint8
    owner
}

type owner struct{
    name string
}

func main(){
    var myEngine gasEngine = gasEngine{25, 15 owner{"Alex"}}
    fmt.Println(myEngine.mpg, myEngine.gallons, myEngine.name)
}
```

Anonymous Structs

```
func main(){
    var myEngine2 = struct{
        mpg uint8
        gallons uint8
    }{25,15}
}
```

- not reusable

Struct methods

```
type gasEngine struct{
    mpg uint8
    gallons uint8
}

func (e gasEngine) milesLeft() uint8 {
    return e.gallons * e.mpg
}

func canMakeIt(e gasEngine, miles uint8){
    if miles <= e.milesLeft(){
        fmt.Println("You can make it there")
    }else{
        fmt.Println("You wont make it")
    }
}

func main() {
    var myEngine gasEngine = gasEngine{25, 15}
    fmt.Printf("Total miles left in tank: %v", myEngine.milesLeft())
}
```

- can access fields in struct

Interfaces

```
type engine interface{
    milesLeft() uint8 // signature
}

type gasEngine struct{
    mpg uint8
    gallons uint8
}

type electricEngine struct{
    mpkwh uint8
    kwh uint8
}

func (e gasEngine) milesLeft() uint8 {
    return e.gallons * e.mpg
}
```

```
func (e electricEngine) milesLeft() uint8 {
    return e.kwh * e.mpkwh
}

func canMakeIt(e engine, miles uint8){
    if miles <= e.milesLeft(){
        fmt.Println("You can make it there")
    }else{
        fmt.Println("You wont make it")
    }
}

func main() {
    var myEngine gasEngine = gasEngine{25, 15}
    fmt.Printf("Total miles left in tank: %v", myEngine.milesLeft())
}
```

Pointers

- star syntax
- initially `nil`
- if initialized with memory location -> memory location has default value

```
var np *int32 // nil pointer
var p *int32 = new(int32)
var i int32
fmt.Printf("The value p points to is: %v\n", *p) // dereferencing like C
fmt.Printf("The address p points to is: %v\n", p)
fmt.Printf("The value of i is: %v\n", i)

*p = 10 // assign value
p = &i // create pointer to i
*p = 1 // changes the value of i
```

Pointers with Functions

Without pointer:

```
import "fmt"

func main(){
    var thing1 = [5]float64{1,2,3,4,5}
    fmt.Printf("\nThe memory location of the thing1 array is: %p", &thing1)
    var result [5]float64 = square(thing1)
    fmt.Printf("\nThe result is: %v", result)
```

```
    fmt.Printf("\nThe value of thing1 is: %v", thing1) // unaffected
    // (square works on copy)
}

func square(thing2 [5]float64) [5]float64{
    fmt.Printf("\nThe memory location of the thing2 array is: %p", &thing2)
    for i:= range thing2{
        thing2[i] = thing2[i]*thing2[i]
    }
    return thing2
}
```

With pointer:

```
import "fmt"

func main(){
    var thing1 = [5]float64{1,2,3,4,5}
    fmt.Printf("\nThe memory location of the thing1 array is: %p", &thing1)
    var result [5]float64 = square(&thing1)
    fmt.Printf("\nThe result is: %v", result)
    fmt.Printf("\nThe value of thing1 is: %v", thing1) // squared
}

func square(thing2 *[5]float64) [5]float64{
    fmt.Printf("\nThe memory location of the thing2 array is: %p", &thing2)
    for i:= range thing2{
        thing2[i] = thing2[i]*thing2[i]
    }
    return thing2
}
```

Goroutines

- enable **concurrency** (!= parallel execution)
- mostly achieve **parallel execution** on multi-core CPUs

```
package main

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

```go
var wg = sync.WaitGroup{}
var dbData = []string{"id1", "id2", "id3", "id4", "id5"}

func main(){
    t0 := time.Now()
    for i:= 0; i < len(dbData); i++{
        wg.Add(1) // add 1 to wait group
        go dbCall(i) // go keyword -> doesn't wait for completion
    }
    wg.Wait() // wait for the counter to be 0
    fmt.Printf("\nTotal execution time: %v", time.Since(t0))
}

func dbCall(i int) {
    // Simulate DB call delay
    var delay float32 = rand.Float32()*2000
    time.Sleep(time.Duration(delay)*time.Millisecond)
    fmt.Println("The result fromm the DB is:", dbData[i])
    wg.Done() // decrement wait group counter by one
}
```

Thread Safety with Mutex:

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
    "sync"
)

var m = sync.Mutex{}
var wg = sync.WaitGroup{}
var dbData = []string{"id1", "id2", "id3", "id4", "id5"}
var results = []string{}

func main(){
    t0 := time.Now()
    for i:= 0; i < len(dbData); i++{
        wg.Add(1) // add 1 to wait group
        go dbCall(i) // go keyword -> doesn't wait for completion
    }
    wg.Wait() // wait for the counter to be 0
    fmt.Printf("\nTotal execution time: %v", time.Since(t0))
    fmt.Printf("\nThe results are %v", results)
```

```
}

func dbCall(i int) {
    // Simulate DB call delay
    var delay float32 = rand.Float32()*2000
    time.Sleep(time.Duration(delay)*time.Millisecond)
    fmt.Println("The result fromm the DB is:", dbData[i])
    m.Lock() // set mutex semaphore
    results = append(results, dbData[i])
    m.Unlock() // release mutex semaphore
    wg.Done() // decrement wait group counter by one
}
```

Thread Safety with Read/Write Mutex:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
    "sync"
)

var m = sync.RWMutex{}
var wg = sync.WaitGroup{}
var dbData = []string{"id1", "id2", "id3", "id4", "id5"}
var results = []string{}

func main(){
    t0 := time.Now()
    for i:= 0; i < len(dbData); i++{
        wg.Add(1) // add 1 to wait group
        go dbCall(i) // go keyword -> doesn't wait for completion
    }
    wg.Wait() // wait for the counter to be 0
    fmt.Printf("\nTotal execution time: %v", time.Since(t0))
    fmt.Printf("\nThe results are %v", results)
}

func dbCall(i int) {
    // Simulate DB call delay
    var delay float32 = rand.Float32()*2000
    time.Sleep(time.Duration(delay)*time.Millisecond)
    save(dbData[i])
    log()
```

```
    wg.Done()
}

func save(result string){
    m.Lock()
    results = append(results, result)
    m.Unlock()
}

func log(){
    m.RLock() // checks full lock but ignores other RLocks
    fmt.Printf("\nThe current results are: %v", results)
    m.RUnlock()
}
```

Channels

- hold data
- thread safe
- listen for data

Unbuffered Channels

- only hold 1 value at the same time

!!! The following doesn't work and gives a deadlock !!!*

```
package main

func main(){
    var c = make(chan int) // channel holds a single integer
    c <- 1 // add value 1 to channel
    // BLOCK and wait for channel to be read -> deadlock
    var i = <- c // retrieve value from channel (empties channel)
}
```

Correctly:

```
package main

import "fmt"

func main(){
    var c = make(chan int)
    go process(c)
    fmt.Println(<-c)
}
```

```
func process(c chan int){
    c <- 123
}
```

Multiple writes:

```
package main

import "fmt"

func main(){
    var c = make(chan int)
    go process(c)
    for i:= range c{
        fmt.Println(i)
    }
}

func process(c chan int){
    defer close(c) // notify all listeners
    for i:=0; i<5; i++{
        c <- i
    }
}
```

Buffered Channels

- able to hold multiple values at the same time (in **buffer**)

```
package main

import (
    "fmt"
    "time"
)

func main(){
    var c = make(chan int, 5) // specify buffer size
    go process(c)
    for i:= range c{
        fmt.Println(i)
        time.Sleep(time.Second*1) // <- slower than process
    }
}

func process(c chan int){
```

```
    defer close(c)
    for i:=0; i<5; i++{
        c <- i
    }
    fmt.Println("Exiting process") // <- exits before main() returns
}
```

Select Statements

- listens to multiple channels
- executes first ready statement or at random

```
package main

import (
    "fmt"
    "time"
)

func main() {

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

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for range 2 {
        select {
            case msg1 := <-c1:
                fmt.Println("received", msg1)
            case msg2 := <-c2:
                fmt.Println("received", msg2)
            }
    }
}
```

Generics

```go
package main

import "fmt"

func main(){
    var intSlice = []int{1, 2, 3}
    fmt.Println(sumSlice[int](intSlice))

    var float32Slice = []float32{1, 2, 3}
    fmt.Println(sumSlice[int](float32Slice))
}

func sumSlice[T int | float32 | float64](slice []T) T{ // pass types instead of value
    var sum T // T is placeholder for actual type
    for _, v := range slice{
        sum += v
    }
    return sum
}
```

Any Type

```go
import "fmt"

func main(){
    var intSlice = []int{}
    fmt.Println(isEmpty[int](intSlice))

    var float32Slice = []float32{}
    fmt.Println(isEmpty[float32](float32Slice))
}

func isEmpty[T any](slice []T) bool{
    return len(slice)==0
}
```

With Structs

```go
package main

import (
    "fmt"
)
```

```go
type gasEngine struct{
    gallons float32
    mpg float32
}

type electricEngine struct{
    kwh float32
    mpkwh float32
}

type car [T gasEngine | electricEngine]struct {
    carMake string
    carModel string
    engine T
}

func main(){
    var gasCar = car[gasEngine]{
        carMake: "BMW",
        carModel: "325d",
        engine: gasEngine{
            gallons: 12.4,
            mpg: 40,
        },
    }
    var electricCar = car[electricEngine]{
        carMake: "BMW",
        carModel: "i4",
        engine: electricEngine{
            kwh: 57.5,
            mpkwh: 4.17,
        },
    }
}
```