FIRST STEPS WITH GO

- Machine language (Binary) < Assembly (Easier to read) < High level

- Compiled languages: Translates instructions before running the code. Fast.
- Interpreted " : " " While " " "
  occurs every execution, requires an interpreter. Slower. The interpreter makes coding easier, memory management and infers variable types.

- Go Advantages:
  - Runs fast
  - Garbage collection, even tho it's compiled. Efficient
  - Is "object oriented" with simpler objects
  - Concurrency efficient

- OOP: Has Structs instead of classes, no inheritance, constructors
  Has generics.
- Concurrency (1.1.3):
  - Performance limits, more transistors -> higher clock rates
    but temperature constraints
  - Parallelism: Multiple tasks at the same time on different cores.
    Difficulties: Start/Stop?, Sharing Data on tasks, memory conflict

- Concurrency is the management of multiple tasks at the same time
- Good for workspace, easy to share
- Directory hierarchy is not enforced
- There must be one package called main

```
package main
import "fmt"        ~> Included in GoPATH
func main() {
    fmt.Println("hello, world\n")
}
```

go build - Compiles the program. Creates an executable
go doc - Prints documentation of package.
go fmt - formats source code file
go get - downloads packages, even online
go list - lists all "
go run - Compiles and runs the executable.

Var x int
↑ keyword for declaration

If a variable isn't initialized, { x = 0 Int, x = "" String

Can use x := 100 to declare and initialize
Only inside a function

Second Module

Pointers:

- Address to data in memory:
  - & Address of a variable / function (referencing)
  - * Returns data at an address (dereferencing)

```
Var x int = 2
Var y int
Var ip *int

ip = &x  // ip points to x
y = *ip  // y is now 2
```

- New: Alternative way to create a variable.
  - new() function creates "" and returns a pointer

```
ptr := new(int)
*ptr = 3
```

Variable Scope: places where it can be accessed. If $b_i >= b_j$ inside "transitive"

Lexical Scoping: Comparing blocks of code

Deallocating Memory: Make Space available. Else, we run out of memory.

Stack vs Heap:

```
var x = 4
func f() {
    fmt.Printf("%d", x)
}
```

→ heap

- Heap is persistent
- Manually in C.

```
func f() {
    var x = 4
    fmt.Printf("%d", x)
}
```

→ Stack

- Function calls, deallocates automatically when complete.

```
func foo() *int {
    x := 1
    return &x
}

func main () {
    var y *int
    y = foo()
    fmt.Printf("%d", *y)
}
```

If foo() didn't return the pointer, x value would have been deallocated

Go has garbage collection, so it knows when a variable is in use and when it's not.

- Comments: //, /* */
- Printing: fmt.Printf(String)
  format: fmt.Printf("%s", x)

Type Conversions:

$x = T(x)$

ASCII: 8 bit number. UTF-8 (the same codes of ASCII or more.)

UNICODE: 32 bit Character Code.

Code Point: Unicode Characters

Rune: a code Point in GO A = 0x41

Strings:

- Read Only, Can't be modified. - Double quotes "Hi"
- Each byte is a Rune.

Functions:

- IsDigit(r rune) - IsSpace(r) - IsLetter(r)
- IsLower(r) - IsPunct(r) - ToUpper(r)
- ToLower(r) - Compare(a,b) - Contains(a,b)
- HasPrefix(a,b) - Index(a,b) - Replace(a, old, new, n)
- TrimSpace(s) " " -> "" (Strconv): - Atoi(s) string to int
- Itoa(s) int to string
- FormatFloat(f, fmt, prec, bitSize) : Float to String
- ParseFloat(s, bitSize) : String to Float.

Constants:

```
Type Grades int
Const (
    A Grades = iota   - Assigns a constant a unique integer.
    B                 - Starts in 1
    C
)
```

Control flow: Determines the order of the statements

```
if <condition> {
    <consequente>
}

for <init>; <cond>; <update> {
}

Switch x {
Case 1: // x == 1
    fmt.Printf("Case 1")
Case 2:
    //
Default:
    fmt.Printf("No Case")
}

Switch {  // No tag
Case x > 1:
    //
Case x < 1
    //
Default
    //
}
```

Continue // Skips the rest of the code after it.

Scan: Pointer as an argument

```
var appleNum int
fmt.Printf("")
num, err := fmt.Scan(&appleNum)
```

Module 3:

- Arrays initialized with all entries 0.

```
Var x [5]int = [5]{1,2,3,4,5}
    x := [...]int{1,2,3} // infers size.

for i, v range x {                    // i index, v value v=x[i]
  fmt.Printf("at %d, val %d", i, v)}
```

Slices: Window on a possibly longer array.

- Variable size
- Pointer indicate start
- Length number of elements len()
- Capacity max number of elements, start of slice — end array. Cap()

```
arr := [...]string{"a","b","c","d"}
S1 := arr[1:3]  //["b","c"]
Sli := []int{1,2,3} // Slice, not array.
```

- make() Create a Slice (array), 2 argument (Type, length); 3 (type, length, capacity)

append() adds to the end of the array, increases size of array in core

Hash Tables: Hash function computes slot for a key.
- Faster search time O(1) - May have collisions
- Arbitrary keys

Maps: Implementation of HT.

```
var idMap map[String][int]
idMap = make(map[String]int)
```

```
idMap := map[String]int{"J": 1}
```

Returns 0 if a key is not present.

```
id, P := idMap["Joe"] // id value, P present of key, Bool
len()

for Key, Val := range idMap {
    fmt.Println(Key, Val)}
```

Structs: "Class"

```
type Person struct {
    name string
    addr string }
```

Dot notation accessing Struct Fields

```
p1.name = "Jo"
X = p1.addr
```

Use new() initializes fields to 0. or: Person{name: "J", addr: ""}

Module 4

RFC: Request for Comments. Internet protocols and formats. interactions for trading data, so it can be understood. Protocol example: HTML, URI, HTTP

Protocol Packages: encode and decode protocol formats. "net/http" "net"

JSON Javascript Object Notation represents structure data. Attribute-Value pairs, struct or map. Unicode, readable

Marshalling: Go object into JSON representation.

```
p1 = Person{...}
barr, err := json.Marshal(p1) // barr: JSON representation as [] byte
                              // err: nil
```

Unmarshalling:

```
var p2 Person
err := json.Unmarshal(barr, &p2) // barr: byte array contains JSON object
                                 // &p2: address of p2
```

Object must fit JSON <> byte

Files:

- linear access, Open, Read, Write, Close, Seek ioutil Package

```
dat, e := ioutil.ReadFile("test.txt") // not needed Open, Close - large Files Problems.
                                      // dat <> byte filled with contents of other file.

dat = "Hello"
err := ioutil.WriteFile("output.txt", dat, 0777)
// name of the file  byte array  Permission to read/write
```

os.Open() // opens file ; os.Close() ; os.Read() reads into a []byte, can be limited to the size of it

os.Write()

f, err := os.Open("...")

bsm := make([]byte, 10)

nb, err := f.Read(bsm)

f.Close()

f.WriteString("...")