# Go CheatSheet

- Sabela Ramos, T3chFest 2020. -

## Basic Commands

To run inside the source directory, or indicating the path as a parameter.

- Compile: `go build`
- Install : `go install`
- Remove: `go clean -i`
- Test: `go test`
- Get a remote library: `go get`
- Format code: `go fmt`

## Typical workspace (located in `$GOPATH`)

```
bin/
   hello # command executable
src/
   github.com/golang/example/
      .git/ # Git repository metadata
      hello/
         hello.go #command source
         hello_test.go # test source
```

## Packages and imports

```
package mypackage //always the first line

import "fmt" // core library
import "github.com/golang/example/hello"

//rest of the source file
```

Use unique names for your packages and paths if you plan to make them available online (e.g., throug GitHub).

Include the URL when importing a remote package.

## If...else

```
err := myfunc()
if err != nil{
   return err
} else {
   log.Info("Success")
   return nil
}

err := myfunc()
if err!=nil{
   return err
}
log.Info("Success")
return nil

if err := myfunc(); err != nil {
   return err
}
log.Info("Success")
return nil
```

## Naming

- Start with lowercase: only accessible within the package.
- Start with uppercase: public.
- Usually, camelcase names.

## Variables

Basic types: `string`, `int`, `bool`, `byte`

Variables declared and not initialized have a zero value:

- 0 for numeric types
- `false` for the boolean type
- "" (the empty string) for strings

```
var(
   b = math.Sin(10)
   c string
)

func init(){} // complex initializations.

// The next three options are equivalent va

a := 0

var a = 0
```

## Slices

```
s := [] int {1, 2, 3}

s := make([] int, 100)

s[0] = 3

for i:= range s{...}
for i,value := range s{...}

s := append(s, 1, 2) // s can be nil.
```

## Loop

```
for i := 0; i < 10; i++ {
   doSomething(i)
}

for i, j := 0, 100; i < j; i,j = i+1, j-1
   doSomething(i,j)
}

for i > 10 {
   i = doSomething()
}

for key, value := range myslice {
   doSomething(key, value)
}
for _, value := range myslice{...}
for key := range myslice{...}

// Infinte loop
for { doSomething() }
```

Use `continue` or `break` to skip an iteration or to exit a loop.

## Structs

```
type thing struct{
   a int
   b string
}
var w thing          // not a pointer
t := new(thing)      // pointer
v := &thing{a:1}     // pointer
x := thing{b:"hi"} // not a pointer
```

## Pointers

The zero value of a pointer is nil.

```
var a int // a is not a pointer
&a          // address of a

var p *int // p is a pointer
*p          // content of p
```

## Maps

```
var age = map[string]int{
   "ana" : 36,
}

// insert
age["tom"] = 36

// read
value, ok := age["marge"]
if !ok{
   ...
}

// iterate
for key,value := range offset{...}
```

## Switch

```
// Only one case executes.
switch {
   case t == a :
      DoA()
   case t == b:
      DoB()
   default:
      DoC()
}

switch t {
   case a:
      DoA()
   case b:
      DoB()
   default:
      DoC()
}

switch t {
   case a, b:
      DoAB()
   default:
      DoC
}
//type switch:
swicth t := t.(type) {...}
```

## Other control statements

```
conn := openConn()
// Close conn when it goes out of scope.
defer conn.Close()

panic()    // launches execution errors.
recover() // gains control after error.
```

## Print & Log

```
// basic print
fmt.Printf("Hello %d\n", 2)
// print to a file
fmt.Fprintf (os.Stdout, "Hello%d\n,, 20)
// print to a string
s := fmt.Sprintf("Hello%d\n", 2)
```

When formatting:

- Use `%v` for any value
- Use `%T` for the type

For logging, instead of fmt.Printf, use:

- log.Infof
- log.Warningf
- log.Errorf

## Functions

```
func name (input args) (return values)
{code}

func myfunc (v int, b *foo) (foo, error){
   f := foo{v: b.v + v}
   return f, nil
}

f, err := myfunc(2, &foo{v: 1})

// add functions to a type
func (t *foo) funcA (a int) error{...}
func (t foo)  funcB (a int) error){...}

// use them
a := thing{}
err := a.funcA(2)
err := a.funcB(3)
```

## Interfaces:

*If something can do this, it can be used here.*

```
// interface
type mytype interface{ func print() }

// function that uses it
func myfunc(a mytype){...}

// type that provides it
type mystruct struct{ s string }
func (m mystruct) print(){
   fmt.Println(mys.name)
}

m := mystruct{s:"hello"}
myfunc(m) // prints hello
```

## Tests

```
package my package

import testing

func TestHello(t *testing.T){
   test_cases := [] struct{
      name string
      in int
      want string
   }{
      {
         name: mytest,
         in : 1,
         want: "Hello, 1",
      }
   }
   for _, tc := range cases{
      t.Run(tc.name, func(t *testing.T){
         gotHello := hello(tc.in)
         if gotHello != tc.want{
            t.Fatal(
               "hello(%d) got %s want %s",
               in, gotHello, want)
         }
      })
   }
}
```

## Go routines

Run functions without waiting for the result.

```
go list.Sort()

go func(){ //do something } ()
```

## Channels

Use channels to communicate Go routines.

```
ci := make (chan int)
c <- 1 //send
b <- c //receive in b
```

## Useful links:

- golang.org (download the binary for your OS)
- golang.org/doc/code.html the very basics
- tour.golang.org a tour of go with exercises
- golang.org/doc/effective_go.html
- play.golang.org

## To learn more:

- golang.org/doc
- github.com/golang/go/wiki/Projects
- go.dev