GoLang

Own Notices from various sources.

Getting Started

Install Go via Brew

```
brew install go
```

Uninstall Go

Online: https://blog.dharnitski.com/2019/04/06/uninstall-go-on-mac/

If previously installed via Brew

```
brew uninstall dep
brew uninstall go
```

If previously installed via Pkgutil

```
pkgutil --pkgs | grep go   # find in the list
sudo pkgutil --forget org.golang.go
```

Go Versions Manager

Online: https://github.com/kevincobain2000/gobrew

Installing

```
curl -sLk https://git.io/gobrew | sh -  # Installation
gobrew use 1.16.4 # Download, install and use in one step

gobrew install 1.16.4  # install only
gobrew use 1.16.4 # change to this version

go uninstall 1.16 # uninstall a certain version
```

ENV VARIABLES (Important for VSCode)

VSCode needs GOPATH and GOBIN to detect the currently used version, if e.g. a package manager like gobrew is installed:

```
# ~/.zshrc excerpt

export PATH="$HOME/.gobrew/current/bin:$HOME/.gobrew/bin:$PATH"
export GOPATH="$HOME/.gobrew/current"
export GOBIN="$HOME/.gobrew/current/bin"
```

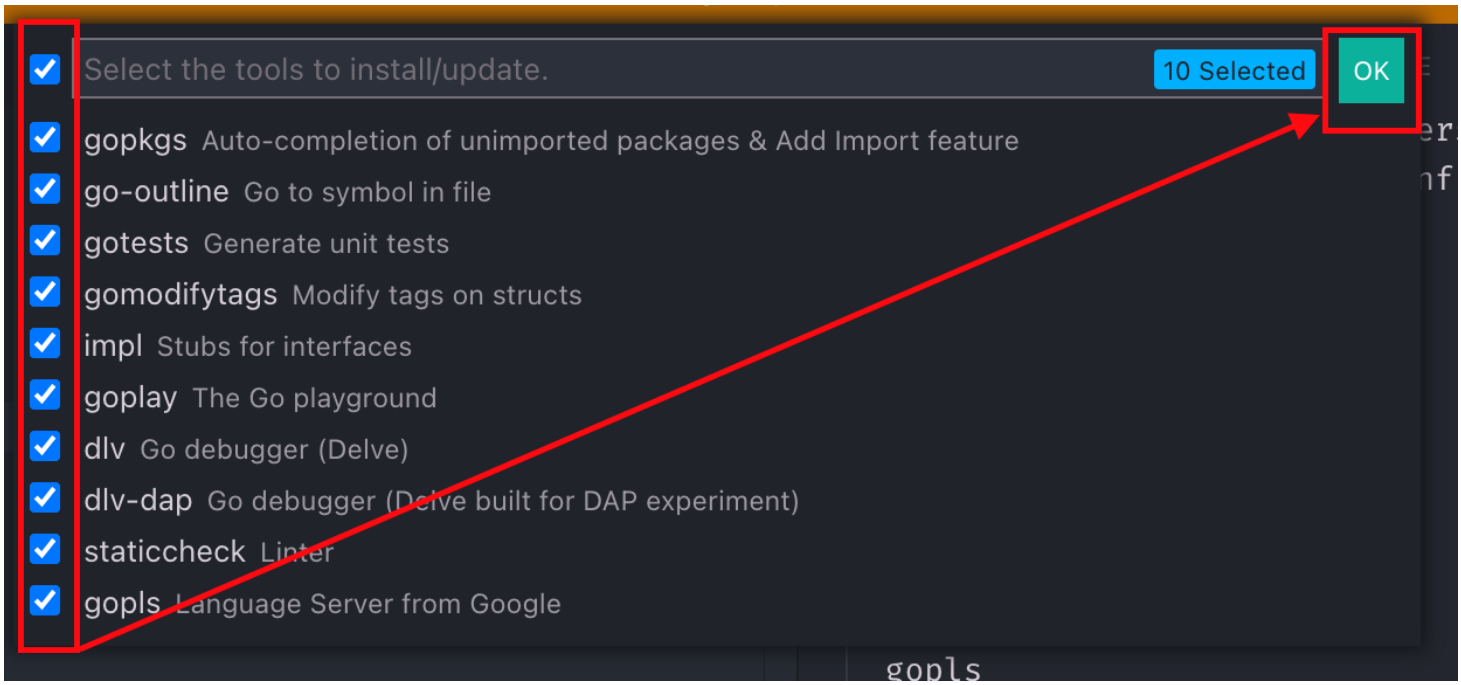
VsCode Install/Update Tools

> **Notice**: You have to repeat this step if you use a go versions manager and and change the current version!

> Further info for VSCode: https://github.com/golang/vscode-go/wiki/tools

> Further General Infos: https://pkg.go.dev/golang.org/x/tools

1. F1 -> Go: Install/Update Tools
2. Select all and click OK



You see then in output window, something like:

```
Installing 10 tools at the configured GOBIN:
  gotests
  gomodifytags
  impl
  goplay
  gopkgs
  go-outline  # within gopls in newer versions
  dlv
  dlv-dap
  staticcheck
  gopls

.. github.com/cweill/gotests/gotests@latest (..go/bin/gotests) SUCCEEDED
.. github.com/fatih/gomodifytags@latest (..go/bin/gomodifytags) SUCCEEDED
.. github.com/josharian/impl@latest (..go/bin/impl) SUCCEEDED
.. github.com/haya14busa/goplay/cmd/goplay@latest (..go/bin/goplay) SUCCEEDED
.. github.com/go-delve/delve/cmd/dlv@latest (..go/bin/dlv) SUCCEEDED
.. honnef.co/go/tools/cmd/staticcheck@latest (..go/bin/staticcheck) SUCCEEDED
.. golang.org/x/tools/gopls@v0.9.1 (..go/bin/gopls) SUCCEEDED

All tools successfully installed. You are ready to Go. :)
```

More information to the tools online:

Go Ppkgs

`gopkgs` is a tool that provides list of available Go packages that can be imported. This is an alternative to `go list all`, just faster.

Online: https://github.com/uudashr/gopkgs/

Go Outline

Simple utility for extracting a JSON representation of the declarations in a Go source file.

https://github.com/ramya-rao-a/go-outline

Go Imports

```
go install golang.org/x/tools/cmd/goimports@latest
```

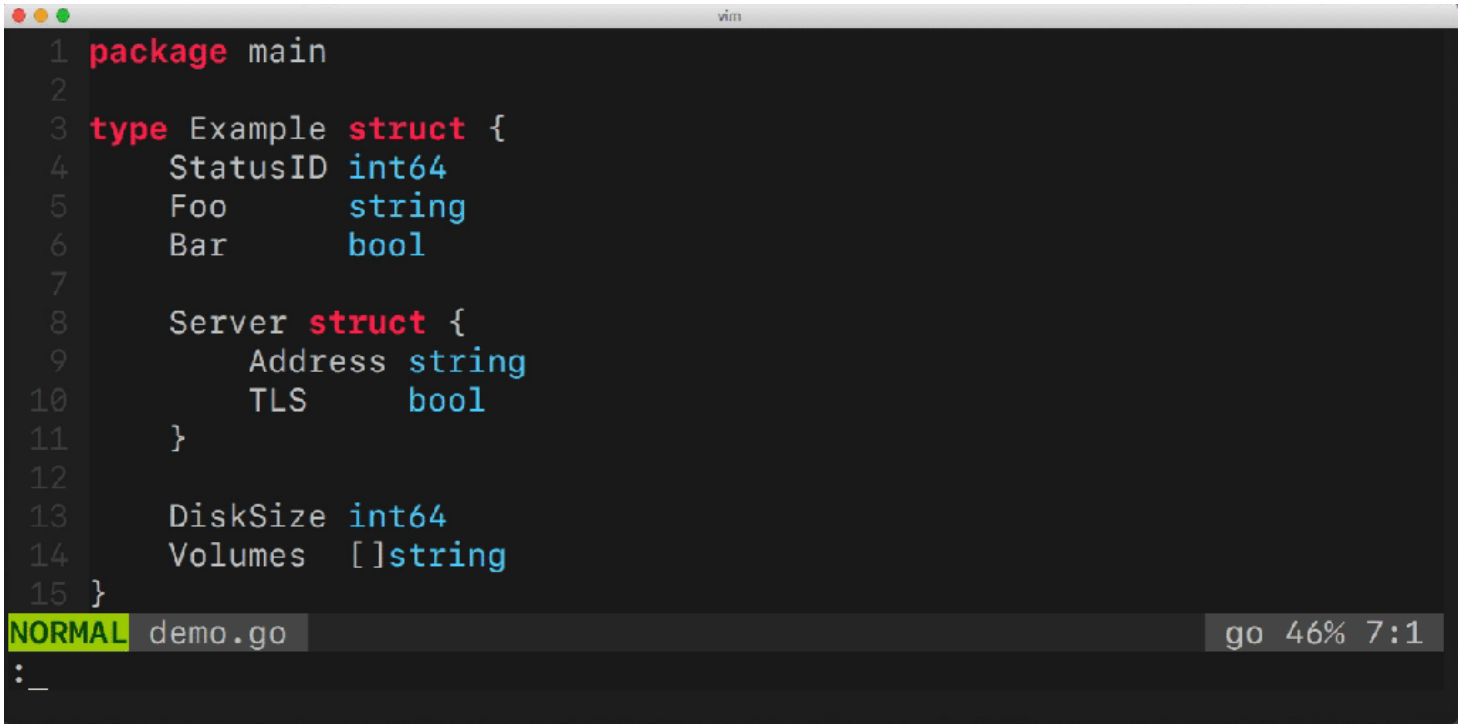
Go Tests

`gotests` makes writing Go tests easy. It's a Golang commandline tool that generates table driven tests based on its target source files' function and method signatures. Any new dependencies in the test files are automatically imported.

https://github.com/cweill/gotests/

Go Modify Tags

Go tool to modify/update field tags in structs. `gomodifytags` makes it easy to update, add or delete the tags in a struct field. You can easily add new tags, update existing tags (such as appending a new key, i.e: db, xml, etc…) or remove existing tags. It also allows you to add and remove tag options. It's intended to be used by an editor, but also has modes to run it from the terminal. Read the usage section below for more information.



https://www.github.com/fatih/gomodifytags

Go Impl

`impl` generates method stubs for implementing an interface.

```
$ impl 'f *File' io.ReadWriteCloser
func (f *File) Read(p []byte) (n int, err error) {
        panic("not implemented")
}

func (f *File) Write(p []byte) (n int, err error) {
        panic("not implemented")
}

func (f *File) Close() error {
        panic("not implemented")
}

# You can also provide a full name by specifying the package path.
# This helps in cases where the interface can't be guessed
# just from the package name and interface name.
$ impl 's *Source' golang.org/x/oauth2.TokenSource
func (s *Source) Token() (*oauth2.Token, error) {
    panic("not implemented")
}
```

https://www.github.com/josharian/impl

Go Play Ground Client

```
goplay some-example.go # opens browser
```

https://www.github.com/haya14busa/goplay/

Go Delve

A Debugger for the Go Programming Language

GitHub: https://www.github.com/go-delve/delve/ Getting Started: https://github.com/go-delve/delve/blob/master/Documentation/cli/getting_started.md

Go Tools from Dominik Honnef: Go `staticcheck`

`Staticcheck` is a state of the art linter for the Go programming language. Using static analysis, it finds bugs and performance issues, offers simplifications, and enforces style rules.

GitHub: https://github.com/dominikh/go-tools Online Documentation: https://staticcheck.io/docs/

gopls, the Go Language Server

> MacOS: XCode and Command Line Tools may be necessary

`gopls` (pronounced “Go please”) is the official Go language server developed by the Go team. It provides IDE features to any LSP-compatible editor.

You should not need to interact with gopls directly–it will be automatically integrated into your editor.

https://pkg.go.dev/golang.org/x/tools/gopls

https://www.golang.org/x/tools/gopls

Basics

Pre-Conditions

1. You have installed go and registered necessary PATHs in your shell configuration, e.g:

```
PATH="/usr/local/sbin:$PATH"
PATH="$HOME/.gobrew/current/bin:$HOME/.gobrew/bin:$PATH"
export GOPATH="$HOME/go"
PATH="$GOPATH/bin:$PATH"
```

If you want to use private packages and bypass proxy & co:

```
export GOPRIVATE=example.com/*,example2.com/*,ex3.com/whatever
export GONOSUMDB=example.com/*,example2.com/*,ex3.com/whatever
export GONOPROXY=example.com/*,example2.com/*,ex3.com/whatever
```

2. Choose your workspace within your path, e.g:

```
mkdir $GOPATH/src/MyGoProject/
cd    $GOPATH/src/MyGoProject/
```

3. Initialise Module and Create a Main File

```
go mod init
touch main.go
```

Hello World

A complete program is created by linking a single, unimported package called the `main package` with all the packages it imports, transitively. The main package must have package name main and `declare a function main` that takes no arguments and returns no value.

```
// main.go

package main

import "fmt"

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

```
go run main.go    # Runs directly without building
go build          # Works if there is a module
go build main.go  # or so
./main.go         # prints Hello World or
./MyGoProject      # the same like above
```

At this time you have the following files in your workspace:

```
MyGoProject       # Executable (created by `go build` without file name)
go.mod           # Text File, e.g. ModuleName and used Go Version etc.
main             # Executable (created by `go build main.go`)
main.go          # Source File
```

That's the `go.mod`

```
module MyGoProject

go 1.17
```

You can also use your GitHub repo as a module like:

```
go mod init github.com/webia1/my-go-project
```

Development Environment

Linting & Vetting

`golint` lints the Go source files named on its command line.

```
go install golang.org/x/lint/golint@latest
go lint ./...           # 3 DOTS
```

`vet` examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string. Vet uses heuristics that do not guarantee all reports are genuine problems, but it can find errors not caught by the compilers.

Vet is normally invoked through the go command. This command vets the package in the current directory (no installation required):

```
go vet                  # or
go vet my/project/...   # or
go vet ./...            # 3 DOTS!
```

`golangci-lint` combines `golint` and `go wet` , it runs linters in parallel, uses caching, supports yaml config, has integrations with all major IDE and has dozens of linters included. (Documentation: https://golangci-lint.run/)

```
brew install golangci-lint
```

`Staticcheck` is a state of the art linter for the Go programming language. Using static analysis, it finds bugs and performance issues, offers simplifications, and enforces style rules. https://staticcheck.io/. Configuration: https://staticcheck.io/docs/configuration/. (Installed by VSCode Tools or see below).

```
go install honnef.co/go/tools/cmd/staticcheck@latest
```

Makefiles

> **Makefiles for Go Developers**:
> https://tutorialedge.net/golang/makefiles-for-go-developers/

Go developers have adopted `make` as their solution (save as `Makefile`):

```
.DEFAULT_GOAL := build

fmt:
    go fmt ./...
.PHONY:fmt          # <-- Self chosen name for `fmt` part

lint: fmt            # <-- `fmt` is the pre-condition for `lint`
    golint ./...
.PHONY:lint

vet: fmt
    go vet ./...
.PHONY:vet
```

Makefiles are extremely picky: You must indent the steps in a target with a `tab`.

> A **`PHONY`** target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance.

Once the `Makefile` is in the `"src"` directory (any name can be chosen), type:

```
make
```

Types and Declarations

Common Concepts

Zero

Variable is declared but not assigned a value. (Like `null` in JS)

var vs. `:=`

```
var x int = 10          // is the same like
var x = 10              // because it is assigned, no need for type
x := 10                 // is the same like the two declarations above
var x int               // something like "let x: int = null" in TS
var x, y int = 10, 20   // more than one declarations
var x, y int            // with zero values
var x, y = 10, "Hi"     // with different types
x, y := 10, "Hi"        // same like above

// also a "declaration list" would be possible

var (
  a     int
  b           = 20
  c     int   = 30
  d, e        = 40, "Hi"
  f, g  string
)
```

The `:=` operator can reassign (not possible by using `var`):

```
x     := 10
x, y  := 30, "Hi"
```

One limitation for `:=` : At package level you must use var because it is not legal outside of functions.

Important notices:

- Initialisation with zero values -> better `var` than `:=`
- prefer something like `var x byte = 20` to `x := byte(20)`
- `:=` allows you to reassign too. Attention: → `Shadowing Variables`

As a general rule: Declare variables in the package block that are effectively immutable.

Explicit Type Conversion (= Automatic Type Promotion)

Go doesn't allow automatic type conversion, when variable types do not match.

Literals

- Integer Literals, based on 10, except:
  - `0b` binary
  - `0o` octal (0 with no letter after it is octal too, but don't use it)
  - `0x` hexadecimal
- Floating Point Literal
  - e.g. 7.11e23
  - `0x` hexadecimal
  - `p` exponent
  - `_` formatting big numbers
- Rune Literals (Chars in JS)
  - e.g. `('a')`, `('\171')`, `('\x47')`
  - 16 Bit Hexadecimal `('\u0061')`
  - 32 Bit Unicode `('\U00000061')`
  - Newline `('\n')`
  - Tabulator `('\t')`
  - Octal (rare)
- String Literals (very similar to JS)
  - Double Quotes
    - `"Hello World"`
    - `"My \"Hello World\""` If double quotes within -> escape them
  - or Backtick (also called Backquotes)
    - In this case, you don't have to escape double quotes within strings:
    - `` `My "Hello World"` ``

Built-in Types

- boolean: `bool`
- integer
- float
- string

Booleans

```
var myFlag bool       // no value assigned -> false
var myFlag = true     // it is bool and true
```

Integer

`NaN` -> Similar to JS

> **Important:** If you assign a type and then use a number **larger than the types range** to assign it, **it will fail**.

> If you convert to a type that has range lower than your current range, **data loss will occur**.

Special name `byte` is an alias for `uint8`, and the other special name `int` is CPU dependent (e.g. int32 or int64).

> There are some uncommon 64Bit CPU architectures with 32 bit signed integer: Go supports: `amd64p32`, `mip64p32`, and `mips64p32le`

> Go does not have generics and function overloadings (yet?).

Source: https://gosamples.dev/int-min-max/

To get the maximum and minimum value of various integer types in Go, use the `math` package constants. For example, to get the minimum value of the `int64` type, which is **-9223372036854775808**, use the `math.MinInt64` constant. To get the maximum value of the `int64` type, which is **9223372036854775807**, use the `math.MaxInt64`. To check the minimum and maximum values of different int types, see the following example and its output.

> For unsigned integer types, only the max constant is available because the minimum value of unsigned types is always 0.

Signed integers in Go

Source: https://golangdocs.com/integers-in-golang

Signed integer types supported by Go is shown below.

```
int8    // is -128 to 127
int16   // is -32768 to 32767
int32   // is -2147483648 to 2147483647
int64   // is -9223372036854775808 to 9223372036854775807
```

Unsigned integers in Go

```
uint8   // 0 to 255
uint16  // 0 to 65535
uint32  // is 0 to 4294967295
uint64  // 0 to 18446744073709551615
```

Type Conversion

We do typecast by directly using the name of the variable as a function to convert types:

```
package main

import ("fmt")

func main() {
    var x int32
    var y uint32     // range 0 to 4294967295
    var z uint8      // range 0 to 255
    fmt.Println("Type Conversion")
    x = 26700
    y = uint32(x)       // data preserved because number is inside range
    z = uint8(x)        // data loss due to out of range conversion
    fmt.Println(y, z)   // prints 26700 76
}
```

Integer Operations

- `+, -, *, /, % for modulus` Division by 0 causes so called `panic`
- `==, !=, >, >=, <, <=` Comparisons
- `<<, >>, &, |, ^, &^` Bit Manipulations (^ = XOR, &^ = AND NOT)

Floating Point Types

Go supports the `IEEE-754` 32-bit and 64-bit floating-point numbers. You can use all standard number operators with floats except `%` .

IEEEE-754: https://en.wikipedia.org/wiki/IEEE_754

> **Do not use them to represent money or whatever needs to have an exact decimal representation**

```
float32
float64
```

Source: https://gosamples.dev/float64-min-max/

The maximum value of the `float64` type in Go is **1.797693134862315708145274237317043567980 70e+308** and you can get this value using the `math.MaxFloat64` constant.

The minimum value above zero (smallest positive, non-zero value) of the `float64` type in Go is **4.940656458412465441765687928682213723650 5980e-324** and you can get this value using the `math.SmallestNonzeroFloat64` constant.

The maximum value of the `float32` type in Go is **3.402823466385288598117041834845169254 40e+38** and you can get this value using the `math.MaxFloat32` constant.

The minimum value above zero (smallest positive, non-zero value) of the `float32` type in Go is **1.401298464324817070923729583289916131280e-45** and you can get this value using the `math.SmallestNonzeroFloat32` constant.

Type Conversion

Loss of precision will occur when a 64-bit floating-point number is converted to 32-bit float.

Source: https://golangdocs.com/floating-point-numbers-in-golang

```go
package main

import (
    "fmt"
)

func main() {
    var f1 float32
    var f2 float64

    f2 = 1.234567890123
    f1 = float32(f2)

    fmt.Println(f1)         // prints "1.2345679"
}
```

Complex Numbers

Floating-point numbers are used in complex numbers as well. The real and imaginary parts are floats.

More information: https://golangdocs.com/complex-numbers-in-golang

```go
package main

import (
        "fmt"
        "math/cmplx"
)

func main() {
        x := complex(2.5, 3.1)
        y := complex(10.2, 2)
        fmt.Println(x + y)
        fmt.Println(x - y)
        fmt.Println(x * y)
        fmt.Println(x / y)
        fmt.Println(real(x))
        fmt.Println(imag(x))
        fmt.Println(cmplx.Abs(x))
}
```

Matrix (Matrizen)

No matrix support 😕

Strings

Similar to JS. `Zero value` is empty string.

Const

Very similar to TS. Constants can be typed or untyped.

If constants are untyped e.g. the following is allowed:

```
// it is a number but there is no specific type
const x = 10;

// therefore the following assignments are OK
var a int = x       // OK
var b float64 = x   // OK
var c byte = x      // OK
```

But if you give a type to it, you have to consider:

```
// it is an integer
const x int = 10;
var a int = x       // OK
var b float64 = x   // not OK because int != float
```

Unused Variables & Constants

Every declared local variable must be read. It is a compile-time error to declare a local variable and to not read its value.

But the compiler's unused-variable-check is not precise enough; it accepts a single read, even if there were writes afterwards, same with `go vet`. But `golangci-lint` can detect them.

The Go compiler does not prevent you from creating unread package-level-variables.

Suprisingly: Unused constants are OK 😃

Naming Variables and Constants

Very similar to JS/TS, any Unicode (letter/digit) is allowed. See other parts above (e.g. begining with digit).

`_` is a valid character but Go prefers `camelCase` instead of `snake_case`.

> And undercore `_` by itself is a special identifier name in Go (ignoring a parameter, or prop etc., see examples in the coming sections).

Preferred Go Style (as short as possible within block code):

- k, v → key, value
- i, j → common names for index variables

Composite Types

Arrays

> Completely different than arrays in JS/TS

Confusing definition (By Jon Bodner): “All of the elements in the array must be of the type that’s specified but this does not mean they are always of the same type”. Another quote: **Don’t use arrays unless you know the exact length you need ahead of time.**

```
var x [3]int                // 3 is the size of the array
                            // No values specified, i.e -> x = [0,0,0]
                            // Zero value for int is 0

var x = [3]int{10, 20, 30} // Values set



// Here is a so called sparse array :)

var x = [8]int{1, 3: 7, 5, 6: 8, 9}  // [1 0 0 7 5 0 8 9]

/**
  First value (index 0) is 1,
  7 is index no 3 (the next one, has the next index no)
  5 is index no 4,
  8 is index no 6 and so on, that means
  9 is the index no 7
  everything else is 0
*/

// You can also leave off number by using `...`

var x = [...]int{1, 2, 3}

x[0] = 10
fmt.Println(x)       // [10 2 3]
fmt.Println(len(x))  // 3

// Simulating more dimensional arrays:
// 2 arrays of length 3 with zero values

var x [2][3]int  // [[0 0 0] [0 0 0]]  -> How to modify them, see link below:
```

See more about: https://codezup.com/arrays-in-golang-multi-dimensional-arrays/

Slices

> Slices looks like arrays with some differences (notice the missing `...`).

> **Zero value for a slice** is **`nil`** (and not `0`).

> (To my mind: Zero is something like `null` in JS, `nil` is like `undefined`)

```
var x = [...]int{1, 2, 3}               // --> ARRAY
var x = []int{1, 2, 3}                  // --> SLICE

var x = [8]int{1, 3: 7, 5, 6: 8, 9}     // --> ARRAY
var x = []int{1, 3: 7, 5, 6: 8, 9}      // --> SLICE

var x [2][3]int                         // --> ARRAY
var x [][]int                           // --> SLICE
```

The only thing you can compare a slice with is `nil`. The `reflect package` contains a function calles `DeepEqual` can compare almost anything, including slices.

```
var x []int
fmt.Println (x == nil) // true
```

append (similar to push in JS)

`...` like spread operator in JS but different syntax (postfix instead of prefix).

> It is a compile-time error if you forget to assign the value returned from append. (Go is a **`call by value`** language → no object references like in JS, but real copies)

```
var x = []int{1, 2, 3}
x = append(x, 10)         // [1 2 3 10]
x = append(x, 5, 7, 9)    // [1 2 3 10 5 7 9]
y := []int{20, 30, 40}
x = append(x, y...)       // [1 2 3 10 5 7 9 20 30 40]
```

Runtime Capacity

Runtime capacity is like in C++:

> The rules as of Go 1.14 are to double the size of the slice when the capacity is less than 1,024 and then grow by at least 25% afterward.

Just as the built-in **`len`** function returns the current length of a slice, the built-in **`cap`** function returns the current capacity of a slice. It is used far less frequently than len.

Cap is typically used to determine whether a slice is big enough to accommodate new data or whether a call to make is required to create a new slice.

> The cap function also accepts an array as a parameter, although for arrays, cap always returns the same value as len.

```
var x []int
fmt.Println(x, len(x), cap(x))
x = append(x, 10)
fmt.Println(x, len(x), cap(x))
x = append(x, 11)
fmt.Println(x, len(x), cap(x))
x = append(x, 12)
fmt.Println(x, len(x), cap(x))
x = append(x, 13)
fmt.Println(x, len(x), cap(x))
x = append(x, 14)
fmt.Println(x, len(x), cap(x))
```

outputs:

```
[] 0 0
[10] 1 1
[10 11] 2 2
[10 11 12] 3 4
[10 11 12 13] 4 4
[10 11 12 13 14] 5 8
```

make

The built-in `make` function is responsible for creating an empty slice with a specified length or capacity.

> Your program will panic at runtime if you use a variable to set a capacity that is less than the length.

```
x := make([]int, 5)         // length of 5 and a capacity of 5.
x := make([]int, 5, 10)     // length of 5 and a capacity of 10.


/** The following is tricky:
We cannot directly index into it because it has length 0,
but we can append values to it instead:
*/

x := make([]int, 0, 10)     // length of 0 and a capacity of 10.
x = append(x, 1, 3, 7)
```

A slice's length always increases after an `append`! Make sure that you set the slice's length before using the `make`; otherwise, your slice may start off with a surprising number of zero values.

`nil` vs `zero` Declarations

```
// import fmt and reflect before

var data []int            // nil slice declaration
var data  = []int{}       // empty slice with zero-length

var x []int
var y = []int{}

fmt.Println(x, len(x))    // [] 0  Debugger:  []int len: 0, cap: 0, nil
fmt.Println(y, len(y))    // [] 0  Debugger:  []int len: 0, cap: 0, []

fmt.Println(x == nil)     // true
fmt.Println(y == nil)     // false

fmt.Println(reflect.TypeOf(x))                    // []int
fmt.Println(reflect.TypeOf(y))                    // []int
fmt.Println(reflect.TypeOf(y) == reflect.TypeOf(x)) // true

fmt.Println(reflect.ValueOf(x).Kind()) // slice
fmt.Println(reflect.ValueOf(y).Kind()) // slice

/**
  You cannot compare x == y
  invalid operation: cannot compare x == y
  (var x []int -> slice can only be compared to nil)
  compilerUndefinedOp
*/

// It is possible to create an int slice
// with zero length but greater capacity:

z := make([]int, 0, 10)         // Debugger: []int len: 0, cap: 10, []
fmt.Println(z, len(z), cap(z))  // [] 0 10

/**
Since its length is 0, we cannot directly
index into it, but we can append values to it:
*/

z := make([]int, 0, 10)
z = append(x,3,5,7); // []int len: 3, cap: 3, [3,5,7]

Same with nil-able declarations:

var v []int             // []int len: 0, cap: 0, nil
v = append(v, 3, 5, 7)  // []int len: 3, cap: 3, [3,5,7]
```

Use `make` if you roughly know how big your slice needs to be but don't know what values it will get.

The question is:

- whether you should specify a `nonzero length` or
- a `zero-length and a nonzero capacity` in the call to make.

There are three alternatives:

1. Slice as buffer → nonzero length
2. You know the size → specity the length and index into it. But if the set size was not big enough, you will get `panic` .
3. Or specify zero length, nonzero capacity and append to it. If the real size is smaller then there will be zero values at the end of the slice, if larger, your code will not panic.

Slice-slicing

See the following `slice expressions` that creates a slice form a slice:

```
//            0  1  2  3  4
var x = []int{2, 3, 5, 7, 9}
var a = x[:3]   // 0 (incl) till 3 (excl) -> [2 3 5]
var b = x[2:]   // 2 (incl) till end  -> [5 7 9]
var c = x[1:4]  // 1 (incl) till 4 (excl)  -> [3 5 7]
var d = x[:] // all -> [2 3 5 7 9]
```

Sharing Memory

Important: Slices are not copies, they are references.

```
var x = []int{2, 3, 5, 7, 9}
var b = x[0:2]
x[0] = 100

fmt.Println(x)  // [100 3 5 7 9]
fmt.Println(b)  // [100 3]
```

Many funny things happen:

```
var x = []int{2, 3, 5, 7, 9}
var b = x[:2]

fmt.Println(cap(x), cap(b))   // 5 5

x[0] = 100
b = append(b, 30)

fmt.Println("x:", x)    // x: [100 3 30 7 9]
fmt.Println("b:", b)    // b: [100 3 30]
```

A more confusing example:

> Never `append` to a `slice` if you want to avoid surprises, or use the trick (third parameter with position) after the example below.

```
x := make([]int, 0, 10)
x = append(x, 3, 5, 7, 9)
b := x[:2]
c := x[2:]

fmt.Println("x:", x)        // x: [3 5 7 9]
fmt.Println("b:", b)        // b: [3 5]
fmt.Println("c:", c)        // c: [7 9]

b = append(b, 20, 30, 40)
x = append(x, 11)
c = append(c, 13)

fmt.Println("x:", x)        // x: [3 5 20 30 13]
fmt.Println("b:", b)        // b: [3 5 20 30 13]
fmt.Println("c:", c)        // c: [20 30 13]
```

Notice the 3rd parameter in the slide expression: We limit the capacity of the subslices to their length.

```
x := make([]int, 0, 10)
x = append(x, 3, 5, 7, 9)

b := x[:2:2]          // <-- 3rd parameter
c := x[2:4:4]         // <-- 3rd parameter

fmt.Println("x:", x)  // x: [3 5 7 9]
fmt.Println("b:", b)  // b: [3 5]
fmt.Println("c:", c)  // c: [7 9]

b = append(b, 20, 30, 40)
x = append(x, 11)
c = append(c, 13)

fmt.Println("x:", x)  // x: [3 5 7 9 11]
fmt.Println("b:", b)  // b: [3 5 20 30 40]
fmt.Println("c:", c)  // c: [7 9 13]
```

Converting Array to Slices

You can take a slice from an Array too (same problems - memory sharing - see above, using third parameter helps here too).

```
x := [5]int{1, 3, 5, 7, 9}
b := x[:2:2]
c := x[2:4:4]

fmt.Println("x:", x)          // x: [1 3 5 7 9]
fmt.Println("b:", b)          // b: [1 3]
fmt.Println("c:", c)          // c: [5 7]

b = append(b, 20, 30, 40)
//      x = append(x, 11)  // you cannot append to an array
c = append(c, 13)

fmt.Println("x:", x)          // x: [1 3 5 7 9]
fmt.Println("b:", b)          // b: [1 3 20 30 40]
fmt.Println("c:", c)          // c: [5 7 13]
```

without the 3rd param you will get surprising results (like it is the case with slices). The same example above this time without the 3rd param:

```
x := [5]int{1, 3, 5, 7, 9}
b := x[:2]
c := x[2:4]

fmt.Println("x:", x)            // x: [1 3 5 7 9]
fmt.Println("b:", b)            // b: [1 3]
fmt.Println("c:", c)            // c: [5 7]

b = append(b, 20, 30, 40)
//      x = append(x, 11)  // you cannot append to an array
c = append(c, 13)

fmt.Println("x:", x)            // x: [1 3 20 30 13]
fmt.Println("b:", b)            // b: [1 3 20 30 13]
fmt.Println("c:", c)            // c: [20 30 13]
```

`copy` helps you to avoid memory sharing problems

Same size:

```
a := []int{1, 3, 5, 7, 9}
b := make([]int, 5)

x := copy(b, a)         // (destination <- source)

fmt.Println("a:", a)    // a: [1 3 5 7 9] source
fmt.Println("b:", b)    // b: [1 3 5 7 9] destination
fmt.Println("x:", x)    // x: 5 (number of copied elems)
```

Smaller size: from the beginning of source array

```
a := []int{1, 3, 5, 7, 9}
b := make([]int, 2)

x := copy(b, a)

fmt.Println("a:", a)    // a: [1 3 5 7 9] source
fmt.Println("b:", b)    // b: [1 3] destination
fmt.Println("x:", x)    // x: 2 (number of copied elems)
```

Bigger size: zero values at the end

```
a := []int{1, 3, 5, 7, 9}
b := make([]int, 7)

x := copy(b, a)

fmt.Println("a:", a)    // a: [1 3 5 7 9] source
fmt.Println("b:", b)    // b: [1 3 5 7 9 0 0] destination
fmt.Println("x:", x)    // x: 5 (number of copied elems)
```

From anywhere of the source slice

```
a := []int{1, 3, 5, 7, 9}
b := make([]int, 2)

x := copy(b, a[3:])

fmt.Println("a:", a)    // a: [1 3 5 7 9]
fmt.Println("b:", b)    // b: [7 9]
fmt.Println("x:", x)    // x: 2
```

From overlapping sections of the source slice to the source slice (copy and replace within)

Try out to explain it 😃

```
a := []int{1, 3, 5, 7, 9, 11, 13}

x := copy(a[:3], a[3:])

fmt.Println("a:", a)  // a: [7 9 11 7 9 11 13]
fmt.Println("x:", x)  // x: 3
```

Explanation:

```
1  3  5  replace with 7 9 11 13
different size therefore sub-result => 7 9 11

0  1  2  3  4  5  6
1  3  5  7  9  11 13
7  9  11 7  9  11 13  <- result
```

Second Example:

```
a := []int{1, 3, 5, 7, 9, 11, 13}

x := copy(a[4:], a[1:4])

fmt.Println("a:", a) // [1 3 5 7 3 5 7]
fmt.Println("x:", x) // 3
```

Strings, Runes, Bytes

Strings (re-assignable but immutable) have no capacity only length, written within double quotes. Runes has no length, written within single quotes.

```
str := "Hello World 🤫"
r := 'h'

fmt.Println(str, len(str), r, string(r))
// Hello World 11 104 h


var a string = "Hi there"
var b byte = a[6]
var x string = a[3:]

fmt.Println(a, b, x) // Hi there 114 there
```

Conversion

In general: Take care of the length!

```
str := "Oh 🤫"        // Smiley is 4 bytes
fmt.Println(len(str)) // 7
```

Strings to Slices

> Slices of runes -> uncommon

```
var str string = "Oh 🤫"
var x []byte = []byte(str)
var y []rune = []rune(str)
fmt.Println(x, y)

  // outputs
  // [79 104 32 240 159 171 162] [79 104 32 129762]

  // UTF8: little-endian versus big-endian -> no problems here
```

Maps

The zero value for a `map` is `nil`. Maps are not comparable. You can only check if they are equal to `nil`.

`len` is OK. Key can be any comparable type.

`map[keyType]valueType:`

If we want to read a value of a non-existing key, we get its zero value (e.g. if `int` → 0).

> you can use ++ increment operator to increment the numeric value for a map key. (→ Example)

```
var myFirstMap = map[string]int{
  "foo": 0,  // Attention: Comma is always required
}

var mySecondMap = map[string][]string{
  "foo": {"one", "two"},
  "bar": {"one", "two", "three"}, // Attention: Comma required
}

fmt.Println(myFirstMap, mySecondMap)
// map[foo:0] map[bar:[one two three] foo:[one two]]
```

With `make`

```
        myMap1 := make(map[int][]string, 10)
        myMap2 := make(map[string]int, 10)

        myMap1[13] = []string{"Hi", "there"}
        myMap2["foo"] = 4716

        fmt.Println(myMap1, myMap2)

  // map[13:[Hi there]]
  // map[foo:4716]
```

`ok` → Comma `ok` idiom

It can be any literal, must not be `ok` only.

```
myMap1 := map[string]int{
  "foo": 7,
  "bar": 11,
}

value, ok := myMap1["foo"]
fmt.Println(myMap1, value, ok)

value, ok = myMap1["bar"]
fmt.Println(myMap1, value, ok)

value, ok = myMap1["what"]
fmt.Println(myMap1, value, ok)

// outputs
// map[bar:11 foo:7] 7 true
// map[bar:11 foo:7] 11 true
// map[bar:11 foo:7] 0 false
```

Deleting

Delete function does not return a value!!

```
myMap1 := map[string]int{
  "foo": 7,
  "bar": 11,
  "baz": 2014,
}

delete(myMap1, "baz")

fmt.Println(myMap1) // map[bar:11 foo:7]
```

Maps as Sets

You can use maps as sets because maps because you cannot have duplicate keys in a map.

```
myMap1 := map[int]bool{}
myValues := []int{7, 1, 2, 3, 3, 2, 2, 2, 7}

for _, v := range myValues {
  myMap1[v] = true
}

fmt.Println(myMap1, myMap1[11])
// map[1:true 2:true 3:true 7:true]
// false
```

Using `structs` as values within a `set`

- `ok` is a local variable
- `_` is a convention means ignore first return value (value)

see Comma `ok` idiom above

```
// weird constellation for JS/TS-Devs :)
intSet := map[int]struct{}{}
vals := []int{1, 2, 2, 2, 3}
for _, v := range vals {
        intSet[v] = struct{}{}
}

if _, ok := intSet[3]; ok {
        fmt.Println("3 is in the set")
}

fmt.Println("Before Programm End")
```

Structs

- Syntax like interfaces in JS/TS.
- If they are defined wihtin a function, they can only be used in that function.
- `.` Dot Notation

```
type person struct {
        name string
        age  int
}

p1 := person{}
p1.name = "Michael"
p1.age = 55

p2 := person{
        name: "George",
        age:  56,
}

// if so all of them required
p3 := person{"Rita", 66}

// if so, age will be set to its zero value
p4 := person{
        name: "Aretha",
}

fmt.Println(p1, p2, p3, p4)
// {Michael 55} {George 56} {Rita 66} {Aretha 0}
```

Anonymous Structs

```
person := struct {
        name string
        age  int
}{
        name: "George Michael",
        age:  56,
}

fmt.Println(person)
{George Michael 56}
```

Comparing and Converting

Comparing

- possible if
  - composed of comparable types
- not possible if
  - slice/map fields
  - different types/order/props

Anonymous Structs can be compared

- if same props/order/types

Type conversion is possible only

- if same props/order/types

Scopes and Control Structures

Shadowing and detecting shadowing variables

- Shadowing is similar to in JS/TS

Detecting:

```
go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow@latest
```

If you use a `Makefile` :

```
vet:
  go vet ./...
  shadow ./...
.PHONY:vet
```

Go hat a so called `universe block` with so called `predeclared identifiers` .
Dedection within is `universe block` is difficult and not covered by most linters incl. the one above.

If

If can have its block variables like:

```
// rand must be configured -> rand.Seed

rand.Seed(time.Now().UnixNano())

if n := rand.Intn(10); n == 0 {
  fmt.Println("It is zero", n)
} else {
  fmt.Println("Not zero", n)
}
```

for - 4 different formats

- `for` is only looping key word in Go.
- `break` or `continue` (=skip) are allowed
- labeling possible
  - with `labelName:`
  - in a seperate line above `for`
  - then within a `for` e.g. `continue labelName`

C Style

You must use `:=` because `var` is not allowed here

```
for i :=0; i < 10; i++ {
  // ...
}
```

Condition only (like while in JS/TS)

```
i := 0
for i < 100 {
  // ...
}
```

Infinit loop with break

```
for {
  // ....
  if whatEver {
    break
  }
}
```

for range

In General

Idiomatic:

- `i,v` index/value (arr/slice/string) or
- `k,v` key/value (maps/…)
- for-range value is a copy

```
somePrimNumbers := []int{2, 3, 5, 7, 9}
for index, value := range somePrimNumbers {
  fmt.Println(index, value)
}
```

Ignoring Index

```
somePrimNumbers := []int{2, 3, 5, 7, 9}
for _, v := range somePrimNumbers {
  fmt.Println(v)
}
```

Range in a set (key only example)

```
        someMapAsSet := map[string]bool{
  "George":   true,
  "Michael":  true,
  "Aretha":   true,
}

for k := range someMapAsSet {
  fmt.Println(k)
}
```

Iterating over maps

Order is always different, but fmt.Println sorts keys in ascending order.

```
m := map[string]int{
  "one":   1,
  "two":   2,
  "three": 3,
}

for i := 0; i < 3; i++ {
  fmt.Println("Iteration no", i+1)
  for k, v := range m {
    fmt.Println(k, v)
  }
}
```

Iterating over Strings

```
str := "Hi There"

for i, v := range str {
  fmt.Println(i, string(v))
}

// 0 H
// 1 i
// 2
// 3 T
// 4 h
// 5 e
// 6 r
// 7 e
```

Iterating over String Slices

```
str := []string{"Hi", "there"}

for i, v := range str {
  fmt.Println(i, v)
}

// 0 Hi
// 1 there
```

Using for-range like a arr.forEach in JS/TS

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

for _, v := range initialValues {
        doubles = append(doubles, v*2)
}

fmt.Println(initialValues) // [1 2 3]
fmt.Println(doubles) // [2 4 6]
```

Nested for-range loops with labels

No skipped values:

```
        initialValues := [][]int{{1, 2, 3}, {2, 4, 6}}

myLabelLevel0:
        for i, level0Val := range initialValues {
                for _, level1Val := range level0Val {
                        if level1Val == 11 {
                                continue myLabelLevel0
                        }
                }
                fmt.Println("Iteration", i+1, "Level 1: No values skipped")
        }

// Iteration 1 Level 1: No values skipped
// Iteration 2 Level 1: No values skipped
```

Skipped Values:

```
        initialValues := [][]int{{1, 2, 3}, {2, 4, 6}}

myLabelLevel0:
        for i, level0Val := range initialValues {
                for _, level1Val := range level0Val {
                        if level1Val == 3 {
                                fmt.Println("Iteration", i+1, "Level 1: Skipped value",
                                continue myLabelLevel0
                        }
                }
                fmt.Println("Iteration", i+1, "Level 1: No values skipped")
        }

// Iteration 1 Level 1: Skipped value 3
// Iteration 2 Level 1: No values skipped
```

Switch

You don’t have to write break, but you can, is the same. If you want to break out of the whole switch statement you have to use a label.

The so called **blank switches** are possible:

- `switch { ... }` Check any predefined variable within switch or
- write it so: `switch a { ... }` and check for a

```
  myNumbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
myForLoop:
        for _, myNumber := range myNumbers {
                switch myNumber {
                case 1, 2, 3:
                        fmt.Println(myNumber, "within 1 - 3")
                case 4:
                        fmt.Println(myNumber, "must be four")
                case 5, 6:
                        // these numbers will be skipped
                case 10:
                        fmt.Println("till we get 10 and break")
                        break myForLoop
                default:
                        fmt.Println(myNumber, "after 6")
                }
        }

        /**
        1 within 1 - 3
        2 within 1 - 3
        3 within 1 - 3
        4 must be four
        7 after 6
        8 after 6
        9 after 6
        till we get 10 and break
        */
```

One more example: Combining it with random numbers

```go
package main

import (
"fmt"
"math/rand"
_ "reflect"
"time"
)

func main() {
  rand.Seed(time.Now().UnixNano())
  switch n := rand.Intn(10); {
  case n == 0:
    fmt.Println("Zero")
  case n > 5:
    fmt.Println("Greater five", n)
  default:
    fmt.Println("That's OK", n)
  }
}
```

`goto`

Please do not use it!

Functions

Blank returns possible: `return` without further info. Better avoid them.

Simple Example

Similar to JS/TS

```go
func main() {
        x := simpleDivision(7, 10)
        fmt.Println("x: ", x) // x:  0.7
}

func simpleDivision(a float32, b float32) float32 {
        if b == 0 {
                return 0
        } else {
                return a / b
        }
}
```

Simulating named and optional params

Use a `struct` for this purpose. At least one prop must be present, others can be absent.

```
func main() {

        s1 := Singer{
                name: "Michael Jackson",
                // age:  55,   // <-- would be OK
        }

        PrintSinger(s1) // {Michael Jackson 55}

        fmt.Println("Before Programm End")

}

func PrintSinger(singer Singer) {
        fmt.Println(singer)
}

type Singer struct {
        name string
        age  int
}
```

Variadic Input Params and Slices

Like spread operator `...` in JS/TS, but:

- if as param -> the only or the las param
- Differences
    - `func addToInitial(initial int, numbers ...int)` ← between
    - `addToInitial(2, []int{2, 4, 6}...)` ← after

```go
func main() {

        fmt.Println(addToInitial(1))          // []
        fmt.Println(addToInitial(2, 3))       // [5]
        fmt.Println(addToInitial(1, 2, 3, 4)) // [3 4 5]
        a := []int{2, 3}
        fmt.Println(addToInitial(2, a...))              // [4 5]
        fmt.Println(addToInitial(2, []int{2, 4, 6}...)) // [4 6 8]

        fmt.Println("Before Programm End")

}

func addToInitial(initial int, numbers ...int) []int {
        out := make([]int, 0, len(numbers))
        for _, v := range numbers {
                out = append(out, initial+v)
        }
        return out
}
```

Multiple Return Values

Named or not named, see below:

```
func main() {

        result, remainder, err := divRemNamed(7, 3)
        // or result, remainder, err := divRemNamed(7, 3)
        if err != nil {
                fmt.Println(err)
                os.Exit(1)
        }
        fmt.Println(result, remainder)  // 2 1

        fmt.Println("Before Programm End")

}

func divRem(numerator int, denominator int) (int, int, error) {
        if denominator == 0 {
                return 0, 0,
                        errors.New("Division by 0")
        }
        return numerator / denominator, numerator % denominator, nil
}

func divRemNamed(numerator int, denominator int) (result int, reminder int, err error) {
        if denominator == 0 {
                return 0, 0,
                        errors.New("Division by 0")
        }
        return numerator / denominator, numerator % denominator, nil
}
```

Function as Values

> TODO: Compare it with JS

As long has the several functions have the same signature, one can shape code like below. Notice the values of the Map “OpMap”.

First the output, it is as expected:

```
Result:  6
Result:  2
Result:  8
Result:  2
Before Programm End
```

```
package calculator

func Add(a float64, b float64) float64 { return a + b }
func Sub(a float64, b float64) float64 { return a - b }
func Mul(a float64, b float64) float64 { return a * b }
func Div(a float64, b float64) float64 { return a / b }

type OpFuncType func(float64, float64) float64

var OpMap = map[string]OpFuncType{
        "+": Add,
        "-": Sub,
        "*": Mul,
        "/": Div,
}
```

```
package main

import (
        "fmt"
        "strconv"
        "webia1/MyGoProject/src/calculator"
)

func main() {

        someExpressions := [][]string{
                {"4", "+", "2"},
                {"4", "-", "2"},
                {"4", "*", "2"},
                {"4", "/", "2"},
        }

        for _, se := range someExpressions {
                a, err := strconv.ParseFloat(se[0], 64)
                if err != nil {
                        continue
                }

                op := se[1]
                opKind, ok := calculator.OpMap[op]
    // opKind = webia1/MyGoProject/src/calculator.Add
                if !ok {
                        fmt.Println("Op not within OpMap")
                }

                b, err := strconv.ParseFloat(se[2], 64)
                if err != nil {
                        continue
                }
```

```
                result := opKind(a, b)
                fmt.Println("Result: ", result)

        }

        fmt.Println("Before Programm End")
}
```

Anonymous Functions & IIFEs (like in JS/TS)

Similar to JS:

```
        for i := 0; i <= 3; i++ {
                func(no int) {
                        fmt.Println("No: ", no)
                }(i)
        }
```

Closures (similar to JS/TS)

```
// TODO: Example?
```

Functions as Params

Using slice.Sort btw:

```
        type Person struct {
                Fullname string
                Age      int
        }

        people := []Person{
                {"Michael Jackson", 55},
                {"George Michael", 56},
        }

        sort.Slice(people, func(i int, j int) bool {
                return people[i].Fullname < people[j].Fullname
        })
        fmt.Println("People: ", people)
// People:  [{George Michael 56} {Michael Jackson 55}]


        sort.Slice(people, func(i int, j int) bool {
                return people[i].Age < people[j].Age
        })
        fmt.Println("People: ", people)
// People:  [{Michael Jackson 55} {George Michael 56}]
```

Returning Functions from Functions

Similar to JS/TS

> TODO: Check whether the functions wihin functions have to be anonymous.
> Because `getFullnames` gives an error within main()

```
func main() {

        singers := []Person{
                {"Michael Jackson"},
                {"George Michael"},
        }

        artists := []Person{
                {"Jack Sparrow"},
                {"Mickey Mouse"},
        }

        s := getFullnames(singers)
        a := getFullnames(artists)

        fmt.Println(s(0), " - ", s(1))
        fmt.Println(a(0), " - ", a(1))

        fmt.Println("Before Programm End")
}

func getFullnames(from []Person) func(int) string {
        return func(no int) string {
                return from[no].Fullname
        }
}

// outputs:
// Michael Jackson  -  George Michael
// Jack Sparrow  -  Mickey Mouse
```

defer

> TODO: Take your time because very different

Pointers

Similar to C++, with heavy limitations

```
var s = "Hi"
var ps = &s
fmt.Println(s, ps)  // Hi 0xc000010250
*ps = "Hello"
fmt.Println(s, ps)  // Hello 0xc000010250
```

Pointer Type

```
var x = 3
var pointerToX *int  // <- Pointer Type
fmt.Println(x) // 3

pointerToX = &x
fmt.Println(pointerToX) // 0xc0000ba008
```

Always check for `nil`

```
var i *int
fmt.Println(i)          // <nil>
fmt.Println(i == nil)   // true
fmt.Println(*i)  // Exception has occurred: panic
// "runtime error: invalid memory address
// or nil pointer dereference"
```

`new` creates a zero value pointer

```
var i = new(int)
fmt.Println(i) // 0xc0000ba008
fmt.Println(i == nil) // false
fmt.Println(*i) // 0
```

For structs use `&` before (weird)

> not applicaple to primitive literals, like numbers, booleans and strings

```
type Foo struct{}
var f = &Foo{}          // Type &Foo, value {}
fmt.Println(f == nil)   // false
fmt.Println(f, *f)      // &{} {}
```

With functions

Type of `&` Value is `*` Type, see the following function `getStringPointer` below.

```
package main
import "fmt"

func main() {
        s := "Hi"
        fmt.Println(getStringPointer(s)) // 0xc000010250
}

func getStringPointer(s string) *string {
        return &s
}
```

Another Example:

```
package main

import (
        "fmt"
)

func main() {
        var someString string
        updateS(&someString, "Hi")
        fmt.Println(someString)

        fmt.Println("Before Programm End")
}

func updateS(s *string, value string) {
        *s = value
}
```

With JSON

When a function expects an interface: You can use pointer parameters to modify a variable (only then).

```
package main

import (
        "encoding/json"
        "fmt"
)

func main() {

        type person struct {
                Fullname string `json: "fullname"`
                Age      int    `json: "age"`
        }

        p := person{}

        err := json.Unmarshal([]byte(`{"fullname": "Michael Jackson", "age": 55}`), &p)

        fmt.Println(err) // <nil>
        fmt.Println(p)   // {Michael Jackson 55}

        fmt.Println("Before Programm End")

}
```

Slices as Buffers

> TODO: Add an example

Types

Either Abstract and Concrete in Go:

Abstract Type: Specifies what do do, but not how to do Concrete Type: What & how

Basics

```go
package main

import (
	"fmt"
	"reflect"
)

func main() {

	f := 3
	g := "Hi"
	h := 3.14

	var i = map[string]int{
		"a": 3,
	}

	fmt.Println(reflect.TypeOf(f)) // int
	fmt.Println(reflect.TypeOf(g)) // string
	fmt.Println(reflect.TypeOf(h)) // float64
	fmt.Println(reflect.TypeOf(i)) // map[string]int
}
```

Methods (Difference to functions)

Notice the so called `receiver` specification, it appears between the `function name => method name in this case` and function keyword `func` :

```
package methodexample

import "fmt"

type Person struct {
        Fullname string
        Age      int
}

func (p Person) PersonLogger() string {
        return fmt.Sprintf("%s age %d", p.Fullname, p.Age)
}

// ---------------------------

package main

import (
        "encoding/json"
        "fmt"
        "webia1/MyGoProject/src/methodexample"
)

func main() {

        p := methodexample.Person{}

        err := json.Unmarshal([]byte(`{"fullname": "Michael Jackson", "age": 55}`), &p)

        if err == nil {
                output := methodexample.Person.PersonLogger(p)
                fmt.Println(output)
        }
}
```

Pointer Receivers and Value Receivers

Rule:

- Method modifies the receiver and needs to handle `nil` instances => it must use a pointer receiver

- If not, you can use a value receiver

```
package counterexample

import (
        "fmt"
        "time"
)

type Counter struct {
        no      int
        updated time.Time
}

func (c *Counter) Inc() {
        c.no++
        c.updated = time.Now()
}

func (c Counter) Log() string {
        return fmt.Sprintf("No: %d, updated: %v", c.no, c.updated)
}

//----------------------

package main

import (
        "fmt"
        "webia1/MyGoProject/src/counterexample"
)

func main() {

        var c counterexample.Counter
        fmt.Println(c.Log())
        c.Inc()
        fmt.Println(c.Log())
}

// Outputs:
// No: 0, updated: 0001-01-01 00:00:00 +0000 UTC
// No: 1, updated: 2022-08-18 09:46:34.473224 +0200 CEST m=+0.000195143
```

`nil` instances

A method with a value receiver can’t check for nil, it panics if invoked with a nil receiver. Therefore you need pointer receivers, even it they not modify the struct (=> simulated class).

```
package treeexample

type IntTree struct {
        val         int
        left, right *IntTree
}

func (it *IntTree) Insert(val int) *IntTree {
        if it == nil {
                return &IntTree{val: val}
        }
        if val < it.val {
                it.left = it.left.Insert(val)
        } else if val > it.val {
                it.right = it.right.Insert(val)
        }
        return it
}

func (it *IntTree) Has(val int) bool {
        switch {
        case it == nil:
                return false
        case val < it.val:
                return it.left.Has(val)
        case val > it.val:
                return it.right.Has(val)
        default:
                return true
        }
}

//-------------------------

package main

import (
        "fmt"
        "webia1/MyGoProject/src/treeexample"
)

func main() {

        var it *treeexample.IntTree

        var vals = []int{2, 1, 3, 3, 3, 4, 2, 7, 1, 5, 3}

        for _, v := range vals {
                it = it.Insert(v)
        }
```

```
  fmt.Println(it.Has(7)) // true

}
```

Method vs. Function & Method Expressions

All the following are possible:

```
package main

import (
        "fmt"
)

func main() {

        myAdder1 := SimpleAdder{
                initalValue: 1,
        }
        myF1 := myAdder1.AddTo
        myF2 := SimpleAdder.AddTo // <- METHOD EXPRESSION

        fmt.Println(myAdder1.AddTo(2)) // 3
        fmt.Println(myF1(2))           // 3
        fmt.Println(myF2(myAdder1, 2)) // 3

}

type SimpleAdder struct {
        initalValue int
}

func (s SimpleAdder) AddTo(val int) int {
        return s.initalValue + val
}
```

Typing/SubTyping & Conversions

Subtyping is not the same as inheritance. However, because most programming languages use inheritance to implement subtyping, the definitions are frequently confused in common usage.

Even the types seem to be compatible in the example below they have to be converted:

```
type Points int
type HighPoints int

var i int = 10
var p Points = 20
var h HighPoints = 30

h = p // Compilation Error
p = i // Compilation Error

p = Points(h)                                   // OK
p = Points(i)                                   // OK
h = HighPoints(p)                       // OK
h = HighPoints(i)                       // OK
i = int(p)                                              // OK
i = int(h)                                              // OK
```

iota

Like unnamed Enums in TS:

```
const (
        a = iota
        b
        c
)

        const (
        d = iota + 1
        e
        f
)

const (
        i = iota + 1
        _ // skip
        j
        k
)

        type Direction int

        const (
                North Direction = iota
                East
                South
                West
        )

fmt.Println(a, b, c) //  0 1 2
fmt.Println(d, e, f) //  1 2 3
fmt.Println(i, j, k) //  1 3 4
fmt.Println(North, East, South, West) // 0 1 2 3
```

Composition

Reason: Remember -> `fish` inherits `run()` and has to override it. `run()` better as interface => dog implements it but fish does not.

Another example in Go with structs in this matter, called EMBEDDING. Embedding is not inheritance.

```
package main

import (
        "fmt"
)

func main() {

        var m = Human{
                Creature: Creature{
                        Kind: "Mammal",
                },
                Fullname: "Michael Jackson",
        }

        m.setAge(55)
        fmt.Println(m) // {{Mammal} Michael Jackson 55}

        fmt.Println("Before Programm End")

}

type Creature struct {
        Kind string
}

type Human struct {
        Creature
        Fullname string
        Age      int
}

func (h *Human) setAge(age int) {
        h.Age = age
}
```

Interfaces

Rule: Accept interfaces, return structs (Due to decoupling principle)

A go-interface is a collection of methods as well as it is a custom type. In Go, it is necessary to implement all the methods declared in the interface for implementing an interface

- Idiomatic: `er` at the end Interfaces in Go.
- interfaces provide type-safety & decoupling
- They are `implicit interfaces` ; they specify, what callers need.

```
package main

import (
        "fmt"
)

func main() {

        var s Singer = Singers{
                Name: "Michael Jackson",
        }

        fmt.Println(s)
        s.Sing()
        s.Talk()

        fmt.Println("Before Programm End")

}

type Singer interface {
        Talk()
        Sing()
}

type Singers struct {
        Name string
}

func (s Singers) Talk() {
        fmt.Println("Talking")
}

func (s Singers) Sing() {
        fmt.Println("Singing")
}
```

Standard Interfaces

If there is an Interface in Standardlibrary, what you could use, use it.

`io.Reader` or `io.Writer` makes the life easier, e.g. whether you write to a file or a value in memory.

Standard interfaces -> Decorator Pattern

Embedding Interfaces

Embedding Interfaces is also possible, like:

```
type Fooer interface {
        foo()
}

type Barer interface {
        bar()
}

type FooBarer interface {
        Fooer
        Barer
}
```

Interfaces and `nil` (weird)

Since an interface with a non-nil-type is not equal to nil, you must use reflection, see the following weird example:

```
        var str *string
        var ier interface{}

        fmt.Println(reflect.ValueOf(ier).IsValid()) // false

        fmt.Println(reflect.ValueOf(ier).IsNil())
        // Compiletime Error Panic: reflect.Value.IsNil on zero Value

        fmt.Println(str == nil) // true
        fmt.Println(ier == nil) // true
        ier = str
        fmt.Println(str == nil) // true
        fmt.Println(ier == nil) // false

        fmt.Println(reflect.ValueOf(ier).IsValid()) // true
        fmt.Println(reflect.ValueOf(ier).IsNil())   // true

        fmt.Println("Before Programm End")
```

Empty Interface

Is something like `any` in TS. Typical use case: placeholder for data of uncertain schema read from external sources, e.g. a JSON file.

```
var whatEver interface{}

whatEver = 10
whatEver = "Hi"
whatEver = struct {
        name string
}{
        name: "Michael Mayr",
}

fmt.Println(whatEver)
fmt.Println("Before Programm End")
```

Reading a JSON File

```
package readingjsonexample

import (
        "encoding/json"
        "io/ioutil"
)

type DataType map[string]interface{}

func GetSomeJSON() DataType {
        var data = DataType{}
        var jsonPath = "readingjsonexample/some.json"

        content, err := ioutil.ReadFile(jsonPath)

        if err == nil {
                json.Unmarshal(content, &data)
        }

        return data

}

//------------------

package main

import (
        "fmt"
        "webia1/MyGoProject/src/readingjsonexample"
)

func main() {

        fmt.Println(readingjsonexample.GetSomeJSON())
        fmt.Println("Before Program End")

}
```

Type Assertions

Type Assertion is very different from a type conversion:

- Type conversion
  - can be applied to both
    - concrete types and interfaces
  - checked at compilation time
- Type Assertion (see the example below)
  - can only be applied to interface types
  - checked at runtime (they can fail)
- Conversions → change
- Assertions → reveal

```
package main

import (
        "fmt"
)

type MyInt int

func main() {

        var i interface{}
        var j MyInt = 20
        i = j
        if i2, ok := i.(string); !ok {
                // i.(string) => panic
                fmt.Println("i2", i2) // "" // zero string value
                recover()
        }

        if i3, ok := i.(int); !ok {
                // i.(int) => panic
                fmt.Println("i3", i3) // 0 // zero int value
                recover()
        }

        i4 := i.(MyInt) // OK // 20

        fmt.Println(i4)

        fmt.Println("Before Program End")

}
```

Type Switches

Use, when an interface can have multiple possible types:

```
package main

import (
        "fmt"
        "reflect"
)

func main() {

        var i interface{}

        checkTypes(i)
        checkTypes("")
        checkTypes(0)
        checkTypes(3)
        checkTypes('a')
        checkTypes("Hi")
        checkTypes(map[string]int{"foo": 3})

        fmt.Println("Before Program End")

}

func checkTypes(i interface{}) {
        switch i.(type) {
        case nil:
                fmt.Println(i, "is nil")
        case int:
                fmt.Println(i, "is int")
        default:
                fmt.Printf("%v \n", i)
                fmt.Println(reflect.TypeOf(i))
        }
}
```

Function Types (!important)

Function types allow functions to implement interfaces. Common use case: HTTP handlers.

You could also use normal function parameter but if your function depends on many other functions, it is better to use interface parameter and define a function type to bridge a function to the interface.

Following example shows the usage:

```
type Handler interface {
        ServerHTTP(http.ResponseWriter, *http.Request)
        // .... and many other methods
}

// Functions in Go are first-class concepts
type HandlerFunc func(http.ResponseWriter, *http.Request)

// Method for HandlerFunc "Class"
func (f HandlerFunc) ServerHTTP(w http.ResponseWriter, r *http.Request) {
        f(w, r)
}
```

Implicit Interfaces vs Dependency Injection

There is library for that: https://github.com/google/wire

Full Example:

```
package main

import (
        "errors"
        "fmt"
        "net/http"
)

// A simple utility function "MyLogger"
func MyLogger(msg string) {
        fmt.Println(msg)
}

// A simple data store
type MyDataStore struct {
        someData map[string]string
}

// A method for it
func (mds MyDataStore) GetUserByID(id string) (string, bool) {
        name, ok := mds.someData[id]
        return name, ok
}

// Factory function creates an instande of MyDataStore

func NewMyDataStore() MyDataStore {
        return MyDataStore{
                someData: map[string]string{
                        "1": "Michael Jackson",
                        "2": "George Michael",
                },
        }
}

// Interfaces for DIP

type SomeDataStore interface {
        GetUserByID(id string) (string, bool)
}

type SomeLogger interface {
        LogMessage(message string)
}

// Adapter between Interface and Logger

type LoggerAdapter func(message string)

func (lg LoggerAdapter) LogMessage(message string) {
```

```
        lg(message)
}

// Dependencies defined, now Business Logic:

type SimpleLogic struct {
        l  SomeLogger
        ds SomeDataStore
}

func (ml SimpleLogic) GreetUser(id string) (string, error) {
        ml.l.LogMessage("UserID: " + id)
        name, ok := ml.ds.GetUserByID(id)
        if ok {
                return "Welcome " + name, nil
        } else {
                return "", errors.New("No User")
        }
}

func NewSimpleLogic(l SomeLogger, ds SomeDataStore) SimpleLogic {
        return SimpleLogic{
                l:  l,
                ds: ds,
        }
}

type SomeLogic interface {
        GreetUser(id string) (string, error)
}

type Controller struct {
        logger SomeLogger
        logic  SomeLogic
}

func (c Controller) HandleGreeting(w http.ResponseWriter, r *http.Request) {
        c.logger.LogMessage("Hi")
        userID := r.URL.Query().Get("id")
        message, err := c.logic.GreetUser(userID)
        if err != nil {
                w.WriteHeader(http.StatusBadRequest)
                w.Write([]byte(err.Error()))
                return
        } else {
                w.Write([]byte(message))
        }
}

func NewController(l SomeLogger, logic SomeLogic) Controller {
        return Controller{
```

```
                logger: l,
                logic:  logic,
        }
}

func main() {

        loggerAdapter := LoggerAdapter(MyLogger)
        dataStore := NewMyDataStore()
        logic := NewSimpleLogic(loggerAdapter, dataStore)
        controller := NewController(loggerAdapter, logic)
        http.HandleFunc("/hi", controller.HandleGreeting)
        http.ListenAndServe(":8080", nil)

        // http://localhost:8080/hi?id=1
        // Welcome Michael Jackson

        fmt.Println("Before Program End")

}
```

Generics

Starting with version 1.18, Go has added support for generics, also known as type parameters.

https://gobyexample.com/generics

Errors

`error` is a built-in interface:

```
type error interface {
        Error() string
}
```

`nil` is the zero value for any interface type.

Basics

Runtime Error

That will give you: `runtime error: integer divide by zero`

```
func simpleDiv(a int, b int) (int, error) {
        return a / b, nil
}

func main() {

        result, err := simpleDiv(1, 0)

        if err != nil {
                fmt.Println(err)
        } else {
                fmt.Println(result)
        }
}
```

Catching Runtime Error

But if you write the code like below, you will be able to catch the error:

```
package main

import (
        "errors"
        "fmt"
)

func simpleDiv(a int, b int) (int, error) {
        if b == 0 {
                return 0, errors.New("division by zero")
        }
        return a / b, nil
}

func main() {

        result, err := simpleDiv(1, 0)

        if err != nil {
                fmt.Println(result, err) // 0 division by zero
        } else {
                fmt.Println(result, err) // result <nil>
        }

}
```

Formatting with `fmt.Errorf`

```
func simpleDiv(a int, b int) (int, error) {
        if b == 0 {
                return 0, fmt.Errorf("division by %d", b)
        }
        return a / b, nil
}

func main() {

        result, err := simpleDiv(1, 0)

        if err != nil {
                fmt.Println(result, err) // 0 division by 0
        } else {
                fmt.Println(result, err)
        }
}
```

Sentinel Errors

sentinel: Wache, Wachposten, Markierung, Hinweiszeichen, Schildwache

See more online:

- https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully
- https://dave.cheney.net/tag/errors

Sentinel errors are one of the few variables declared at pacakage level. Their names start with `Err` (Exception `io.EOF`). They should be threated as read-only. (Go compiler cannot enforce this).

> There are even sentinel errors that signify that an error did not occur, like `go/build.NoGoError`, and `path/filepath.SkipDir` from `path/filepath.Walk`. (See URLs above)

Sentinel errors are usually used to indicate that you cannot start or proceed.

Before you define a sentinel error, make sure you need one. **Once defined, it becomes part of your public API, and you have committed to making it available in all future backward-compatible releases.**

It's far better to reuse one of the standard library's existing ones or define an error type that includes information about the condition that caused the error to be returned.

> Jon: If you have an error condition that indicates a specific state in your application has been reached where no further processing is possible and no contextual information is required to explain the error state, a sentinel error is the correct choice.

> Dave: So, my advice is to avoid using sentinel error values in the code you write. There are a few cases where they are used in the standard library, but this is not a pattern that you should emulate.

Example 1

https://go.dev/play/p/qwi4ligYZYh

```
package main

import (
        "errors"
        "fmt"
        "reflect"
)

var ErrDivideByZeroDarling = errors.New("ho ho ho, division by zero")

func Divide(numerator int, denominator int) (int, error) {
        if denominator == 0 {
                return 0, ErrDivideByZeroDarling  // <-- Using the sentinel error
        }
        return numerator / denominator, nil
}

func main() {
        a, b := 10, 0
        result, err := Divide(a, b)
        if err != nil {
                switch {
                case errors.Is(err, ErrDivideByZeroDarling):
                        fmt.Println(reflect.TypeOf(err))
                        fmt.Println(err, errors.Is(err, ErrDivideByZeroDarling))
                        fmt.Println("Debugger")
                default:
                        fmt.Printf("unexpected division error: %s\n", err)
                }
                return
        }

        fmt.Printf("%d / %d = %d\n", a, b, result)
}
```

Example 2 (archive/zip)

Source: https://go.dev/play/p/7kNKASai6ML

```
package main

import (
        "archive/zip"
        "bytes"
        "fmt"
)

/**
// Predefined sentinel errors in zip-package
var (
        ErrFormat    = errors.New("zip: not a valid zip file")
        ErrAlgorithm = errors.New("zip: unsupported compression algorithm")
        ErrChecksum  = errors.New("zip: checksum error")
)
*/

func main() {
        data := []byte("Some content but not zip")
        notAZipFile := bytes.NewReader(data)
        _, err := zip.NewReader(notAZipFile, int64(len(data)))

        // (*err.(data)).s: "zip: not a valid zip file"
        if err == zip.ErrFormat {
                fmt.Println("Format Error: ", err)
        }
        fmt.Println("Before Program End")
}
```

Using Constants for Sentinel Errors => Don’t do it

https://dave.cheney.net/2016/04/07/constant-errors

Errors are values

https://go.dev/play/p/DogOvKKwQkb

Custom Errors

See in the example below `GenerateError` function and `var gennErr error`.

Wenn using custom errors, never define a variable to be of the type of your custom error. Either explicitly return nil wenn no error occurs or define the variable to be of type `error`.

Simulating Exception Handling using `panic`

Check the part:

```
        defer func() {
                if recover() != nil {
                        fmt.Println("is neither")
                }
        }()
```

in the whole example below:

```
package main

import "fmt"

// Positive returns true if the number is positive, false if it is negative.
func Positive(n int) bool {
        if n == 0 {
                panic("undefined")
        }
        return n > -1
}

func Check(n int) {
        defer func() {
                if recover() != nil {
                        fmt.Println("is neither")
                }
        }()
        if Positive(n) {
                fmt.Println(n, "is positive")
        } else {
                fmt.Println(n, "is negative")
        }
}

func main() {
        Check(1)
        Check(0)
        Check(-1)
}
```

Error Types (Dave)

> >> See here online

```
if err, ok := err.(SomeType); ok { … }
```

Opaque errors (Dave)

> >> See here online

```
import “github.com/quux/bar”

func fn() error {
        x, err := bar.Foo()
        if err != nil {
                return err
        }
        // use x
}
```

Wrapping Errors (Example: non existing file)(Jon)

`fmt.Errorf` has a special verb `%w` .

https://go.dev/play/p/N4PNzQCbKXN

```
package main

import (
        "errors"
        "fmt"
        "os"
)

func fileChecker(name string) error {
        f, err := os.Open(name)
        if err != nil {
                return fmt.Errorf("in fileChecker: %w", err)
        }
        f.Close()
        return nil
}

func main() {
        err := fileChecker("not_here.txt")
        if err != nil {
                fmt.Println(err)
                if wrappedErr := errors.Unwrap(err); wrappedErr != nil {
                        fmt.Println(wrappedErr)
                }
        }
        fmt.Println("Before Program End")
}
```

> If you want to create a new error that contains the message from another error, but don't want to wrap it, use `fmt.Errorf` to create an error, but use the `%v` verb instead of `%w`.

```
err := someFunctionsThatReturnsAnError()
if err != nil {
        return fmt.Errorf("internal failure: %v", err)
}
```

Is and As

See the example online: https://go.dev/play/p/qwi4ligYZYh

> you can also use `reflect.DeepEqual()` to compare anything.

Using existing `Is`

`os.ErrNotExist` comes from `io/fs.ErrNotExist`, details:

- https://pkg.go.dev/io
- https://pkg.go.dev/io/fs

```
package main

import (
        "errors"
        "fmt"
        "os"
)

func fileChecker(fullpath string) error {
        f, err := os.Open(fullpath)
        if err != nil {
                return fmt.Errorf("that's from fileChecker and this is from os.Open: %w'
        }
        f.Close()
        return nil
}

func main() {

        err := fileChecker("WrongNameWith.wrongExtension")
        if err != nil {
                if errors.Is(err, os.ErrNotExist) {
                        fmt.Println("Custom Message instead of err: ", err)
                }
        }

        // Outputs:
        // Custom Message instead of err:  that's from fileChecker and this is from os.(
        // open WrongNameWith.wrongExtension: no such file or directory

        fmt.Println("Debugger")
}
```

Implementing own `Is` method

If you want to implement an own error type `errors.Is` may not be compatible/comparable with it.

See in the example below the following line:

```
localMe, ok := target.(MyErr); // comparability checking
```

```
type MyErr struct {
        Codes []int
}

func (me MyErr) Error() string {
        return fmt.Sprintf("codes: %v, me.Codes")
}

func (me MyErr) Is(target error) bool {
        if localMe, ok := target.(MyErr); ok {
                return reflect.DeepEqual(me, localMe)
        }
        return false
}
```

Own `Is` method, Example 2

One of them has to be given: `Resource` or `Code`. Study the following example:

```go
package main

import (
	"errors"
	"fmt"
)

type ResourceErr struct {
	Resource string
	Code     int
}

func (re ResourceErr) Error() string {
	return fmt.Sprintf("%s: %d", re.Resource, re.Code)
}

func (re ResourceErr) Is(target error) bool {

	if other, ok := target.(ResourceErr); ok {
		ignoreResource := other.Resource == ""
		ignoreCode := other.Code == 0
		matchResource := other.Resource == re.Resource
		matchCode := other.Code == re.Code
		return matchResource && matchCode ||
			matchResource && ignoreCode ||
			ignoreResource && matchCode
	}

	return false
}

func main() {

	err1 := ResourceErr{
		Resource: "Database",
		Code:     123,
	}

	err2 := ResourceErr{
		Resource: "Network",
		Code:     456,
	}

	if errors.Is(err1, ResourceErr{Resource: "Database"}) {
		fmt.Println("err1:", err1)
		// err1:  resource Database: code 123
	}
	if errors.Is(err1, ResourceErr{Code: 123}) {
		fmt.Println("err1:", err1)
		// err1:  resource Database: code 123
```

```
        }

        if errors.Is(err2, ResourceErr{Resource: "Network"}) {
                fmt.Println("err2:", err2)
                // err2:  resource Network: code 456
        }

        if errors.Is(err2, ResourceErr{Code: 456}) {
                fmt.Println("err2:", err2)
                // err2:  resource Network: code 456
        }

        if errors.Is(err1, ResourceErr{Resource: "Database", Code: 123}) {
                fmt.Println("err1:", err1)
                // err1:  resource Database: code 123
        }

        if errors.Is(err2, ResourceErr{Resource: "Network", Code: 456}) {
                fmt.Println("err2:", err2)
                // err2:  resource Network: code 456
        }

        if errors.Is(err1, ResourceErr{Resource: "Hohoho"}) {
                fmt.Println("err1:", err1)
                // No output
        }

        if errors.Is(err1, ResourceErr{Resource: "Hohoho", Code: 123}) {
                fmt.Println("err1:", err1)
                // No output
        }

        if errors.Is(err2, ResourceErr{Resource: "Hohoho"}) {
                fmt.Println("err2:", err2)
                // No output
        }

        if errors.Is(err2, ResourceErr{Resource: "Hohoho", Code: 456}) {
                fmt.Println("err2:", err2)
                // No output
        }
}
```

`Error.As`

`Error.As` checks if a returned error (or wrapping err) matches a specific type.

- Input Parameter
  - Error being examined

- Pointer to the type (or to an interface - second example below)

If the second param is not an pointer to an error or interface, the method will panic.

Example 1

```
someErr := someFunctionReturnsAnError()
        var myErr MyErr // zero value of MyErr
        if errors.As(someErr, $myErr) {
        fmt.Println(myErr.Code)
}
```

Example 2

```
        someErr := someFunctionReturnsAnError()
  var coder interface {
                Code() int
        }
        if errors.As(someErr, &coder) {
                fmt.Println(coder.Code())
        }
```

Overwriting `errors.Is`

If you want to match an error of one type and return another, you can overwrite `errors.As` .

> TODO: Add an example

Wrapping Errors with defer

If you don't want to repeat everytime the same message when you wrap multiple errors, you can simplify the code by using `defer` . Both examples below do the same:

```
func DoSomeThings(val1 int, val2 string) (string, error) {
        val3, err := doThing1(val1)
        if err != nil {
                return "", fmt.Errorf("in DoSomeThings: %w", err)
        }
        val4, err := doThing2(val2)
        if err != nil {
                return "", fmt.Errorf("in DoSomeThings: %w", err)
        }
        result, err := doThing3(val3, val4)
        if err != nil {
                return "", fmt.Errorf("in DoSomeThings: %w", err)
        }
        return result, nil
}
```

> TODO Update defer part

Replaceable with:

```
func DoSomeThings(val1 int, val2 string) (_ string, err error) {


  defer func() {
                if err != nil {
                        err = fmt.Errorf("in DoSomeThings: %w", err)
                }
        }() // <-- notice `()`

        val3, err := doThing1(val1)
        if err != nil {
                return "", err
        }
        val4, err := doThing2(val2)
        if err != nil {
                return "", err
        }
        return doThing3(val3, val4)

}
```

“Exception” handling

`panic`

Go generates a panic, when go runtime is don’t know how to proceed. You can call it directly too (example below). It takes only one param of any type (usually

string).

```
package main

import (
        "fmt"
        "os"
)

func myPunicFunc(msg string) {
        panic(msg)
}

func main() {

        myPunicFunc(os.Args[0])

        fmt.Println("Debugger")
}
```

outputs:

```
anic: MyGoProject/src/__debug_bin

goroutine 1 [running]:
main.myPunicFunc({0x7ff7bfeff7b0, 0x3a})
        MyGoProject/src/main.go:9 +0x39
main.main()
        MyGoProject/src/main.go:14 +0x45
```

panic -> defer -> recover

When panic happens the current function exits immediately and any attached `defers` run, till main is reached, then the program exits with a message and a stack trace.

Go provides a way to capture and graceful shutdown or to prevent shutdown; examine the following example (recover within defer):

```
package main

import (
        "fmt"
)

func divideTenBy(i int) {

        defer func() {
                if v := recover(); v != nil {
                        fmt.Println(v)
                }
        }()     // <--- NOTICE THE ENCLOSING PARENS

        fmt.Println(10 / i)
}

func main() {

        someInts := []int{1, 0, 2, 3, 4}
        for _, v := range someInts {
                divideTenBy(v)
        }
}
```

outputs:

```
10
runtime error: integer divide by zero # <-- HANDLED
5
3
2
```

Recipe

- **`panic`** : For fatal situations
- **`recover`** : Gracefully handle
- Or exit with `os.Exit(1)` and log the situation before if there is e.g. hardware defects or memory issues.
    - `recover` does not make clear, what could fail. We can print a message if something fails and continue. But use it if you want to keep your secret parts of your app secret if you creating a library. Don't let `panic` expose them. In case of a `panic` use `recover` to convert it to an error and let the consumers decide how to handle it.

Getting a Stack Trace from an Error →

Details online: https://pkg.go.dev/github.com/pkg/errors

```
type stackTracer interface {
        StackTrace() errors.StackTrace
}
```

`trimpath` : Removing full paths in StackTrace

```
go build -trimpath  .
```

```
# excerpt from `go help build`
-trimpath
                remove all file system paths from the resulting executable.
                Instead of absolute file system paths, the recorded file names
                will begin either a module path@version (when using modules),
                or a plain import path (when using the standard library, or GOPATH).
```

Miscellaneous

Run/Build watch with nodemon

See the `nodemon.json` for configuration. Running with:

```
nodemon --signal SIGTERM
```

Debugging with VSCode

While you are `main.go` is opened, us the following configuration:

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debugging GoLang",
      "type": "go",
      "request": "launch",
      "mode": "auto",
      "program": "${fileDirname}"
    }
  ]
}
```

Trouble Shooting

`invalid version: unknown revision`

```
go clean --modcache
go get -u
```