



# GOLANG, PREMIERS PAS

Un langage simple et efficace



# Qui suis-je ?

## Thomas Saquet

(Non. Pas comme le hobbit.)

- ✉️ thomas@solution-libre.fr
- 🐦 X : @tsaquet / @qtgeekes
- DMETHOD BlueSky : @tsaquet.bsky.social
- .twitch Twitch : qtgeekes
- ▶️ Youtube : Qu'est-ce que tu GEEKes ?

### Formateur / Vulgarisateur

Formation DevOps depuis 2015  
Vidéos Youtube depuis 2017  
Streaming Twitch depuis 2020

### Expert Dev / DevOps / Linux

Professionnel depuis 2008  
Développement, conseil,  
accompagnement



9h00

**Début du cours**  
(Appel Pepal !)

10h30

**Pause**  
(15 minutes)

12h30

**Pause Déjeuner**  
(1 heure)

15h15

**Pause**  
(15 minutes)

17h00

**Fin du cours**



# Organisation du cours

 N'hésitez pas à prendre des notes, à réécrire les explications avec vos mots.

 Posez des questions  Le but c'est que vous compreniez tout !

 A la fin, c'est noté !



# Thèmes abordés dans le cours

- Chapitre 1 : Introduction
  - Dans lequel nous aborderons les concepts de base de golang
- Chapitre 2 : Interfaces et JSON
  - Dans lequel nous verrons la puissance de deux outils importants, les interfaces et le json
- Chapitre 3 : Concurrence
  - Dans nous lequel parlerons la concurrence de



CHAPITRE 1

# INTRODUCTION



# INTRODUCTION

- **Golang, origine** nous présentera les bases du langage Golang.
- **Les outils intégrés** nous présentera les outils fournis avec le langage Golang.
- **La gestion de la mémoire** nous permettra de voir comment la mémoire est gérée en Go.
- **L'architecture d'un programme** nous détaillera comment sont architecturés les programmes Go.
- **Golang, les Maps** nous présentera la structure de données map en Go.
- **Les bases - TP** nous permettra de tester les concepts de base de Golang.



# INTRODUCTION

**Golang, origine**

Les outils intégrés

La gestion de la mémoire

L'architecture d'un programme

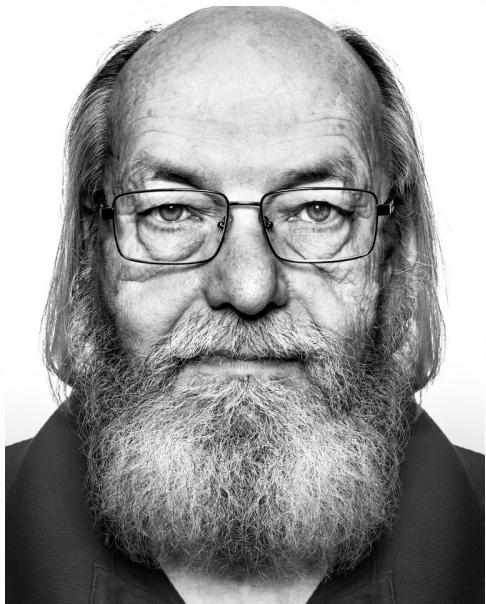
Golang, les Maps

Les bases - TP



# GOLANG, ORIGINE

- Né chez Google en 2007
- Principalement conçu par :



*Ken Thompson*

## Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



*Rob Pike*

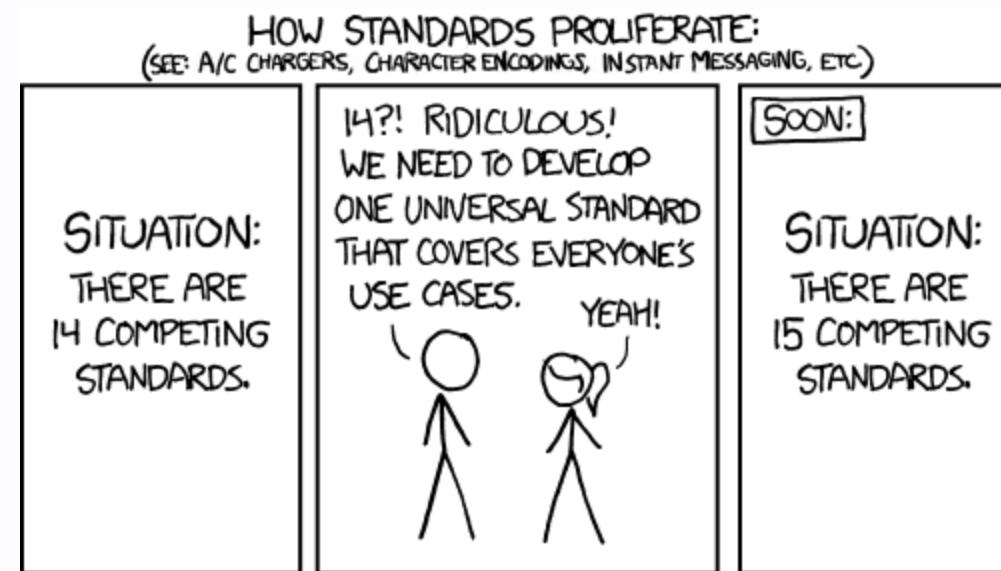


# GOLANG, ORIGINE

## Pourquoi ?

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP





# GOLANG, ORIGINE

## Pourquoi ?

- Temps de compilation du C++ trop longs
- Besoin de typage statique

```
def multiplyByFour(argsList):
    output = ???

    for arg in argsList:
        output += arg - 4

    return output
# (pour rappel, ceci est valide en python:
# argsList = ["name1", "long name1", 1, 2, 3])
```

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# GOLANG, ORIGINE

## Concrètement

- Un langage à syntaxe "C-like", impératif
- **Pas de POO** ("Composition over inheritance")
- Compilé
- Statiquement typé (avec inférence de type)
  - Pas indiqués explicitement dans le code source

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# GOLANG, ORIGINE

## Avantages

- Bibliothèque standard et des outils très complets
- Concurrence et parallélisme
- Garbage collection et sécurité mémoire
- Cross Compilation et static linking

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# GOLANG, ORIGINE

## Bibliothèque ?

- Bibliothèque standard très complète (<https://golang.org/pkg/>) :  
http, archives (tar, zip, bzip...), crypto, driver de DB, XML, JSON, hashing, testing, os...

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# GOLANG, ORIGINE

## Outils ?

Et des outils complets :

- go fmt (formatting automatique du code)
- go mod (vendorizing)
- go doc
- go test
- go install
- ...

Voir ici : <https://golang.org/cmd/go/>

### Golang, origine

Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# INTRODUCTION

Golang, origine

**Les outils intégrés**

La gestion de la mémoire

L'architecture d'un programme

Golang, les Maps

Les bases - TP



# LES OUTILS INTÉGRÉS

## Go fmt

- Formatage automatique du code :
  - Aide à la standardisation de l'apparence du code
  - Évite les commits à "+800" parce que quelqu'un a reformaté dans son IDE
  - Permet aux autres outils go de manipuler le code

Golang, origine  
**Les outils intégrés**  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LES OUTILS INTÉGRÉS

## Go mod

- **Vendoring :**
  - Permet de s'affranchir de l'ancienne architecture (avec le \$GOPATH)
  - Plus besoin de go get
  - Les dépendances sont définies dans un seul fichier "go.mod", et importées dans le code, "par fichier"

Golang, origine  
**Les outils intégrés**  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LES OUTILS INTÉGRÉS

## Go doc

- Documentation :
  - Est-ce que vous documentez vos scripts ?
  - Possibilité de récupérer de la documentation avec :

```
go doc [Symbol]
```

Golang, origine  
**Les outils intégrés**  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LES OUTILS INTÉGRÉS

## Go test

- Les tests :
  - Est-ce que vous testez vos scripts ?
  - Il suffit de placer ses tests dans des fichiers "[nom\_fichier]\*test.go", et de nommer les fonctions de test "Test\*[nom\_fonction]".
  - Lancer les tests avec :

```
go test ./...
```

Golang, origine  
**Les outils intégrés**  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LES OUTILS INTÉGRÉS

## Go install

- Installation des paquets :
  - Même chose qu'un package manager classique type npm, cargo, pip...

```
go install https://github.com/FiloSottile/age
```

Golang, origine  
**Les outils intégrés**  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# INTRODUCTION

Golang, origine

Les outils intégrés

**La gestion de la mémoire**

L'architecture d'un programme

Golang, les Maps

Les bases - TP



# LA GESTION DE LA MÉMOIRE

## Garbage collection / Sécurité mémoire

- Pas de gestion directe des allocations mémoire
- Contrôle plus fin possible :
  - On a accès aux pointeurs, ou valeurs.

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LA GESTION DE LA MÉMOIRE

## Exemple 1 :

```
type Example struct{}

func newExample() Example{
    return Example{}
}

func newExamplePointer() *Example{
    return &Example{}
}
```

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LA GESTION DE LA MÉMOIRE

## Exemple 2 :

```
package main
import "fmt"

func main() {
    message := "Hello Planet"

    // Pointer to string
    var pMessage *string
    ...
```

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LA GESTION DE LA MÉMOIRE

## Exemple 2 (suite) :

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP

```
...
// pMessage points to addr of message
pMessage = &message
fmt.Println("Message = ", *pMessage)
fmt.Println("Message Address = ", pMessage)

// Change message using pointer de-referencing
*pMessage = "Hello Solar System"
fmt.Println("Message = ", *pMessage)
fmt.Println("Message Address = ", pMessage)
}
```



# LA GESTION DE LA MÉMOIRE

## Exemple 2 (suite) :

```
$ go run cosmiclearn.go
Message = Hello Planet
Message Address = 0xc04203e1d0
Message = Hello Solar System
Message Address = 0xc04203e1d0
```

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# LA GESTION DE LA MÉMOIRE

## Garbage collection / Sécurité mémoire

- Pas de gestion directe des allocations mémoire
- Contrôle plus fin possible
- Sécurité mémoire : attention, les nil pointer exceptions (panics en Go) sont possibles !

Golang, origine  
Les outils intégrés  
**La gestion de la mémoire**  
L'architecture d'un programme  
Golang, les Maps  
Les bases - TP



# INTRODUCTION

Golang, origine

Les outils intégrés

La gestion de la mémoire

## L'architecture d'un programme

Golang, les Maps

Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Architecture d'un programme

- Un programme Go s'organise en packages
- Il y a une fonction main, dans le package main, qui sert à lancer le programme
- Un package est un "dossier" dans le code source qui sert à organiser le code

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Packages

- Pour déclarer un package :

```
package main
```

- Pour utiliser un package :

```
import "NomDuModule/main"
```



# L'ARCHITECTURE D'UN PROGRAMME

## Packages

- Visibilité :
  - Publique : commence avec une majuscule

```
func MyFunc(){}
```

- Privée : minuscule

```
func myFunc(){}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Le code

- Les types de variables
- Déclarer une variable
- Définir une fonction
- Définir une méthode
- Les structures de base

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Les types de variables

Principaux types de données en Go:

- **bool** : true ou false
- **int** : entiers signés 32 ou 64 bits
- **float32, float64** : nombres à virgule flottante
- **string** : chaîne de caractères
- **array** : tableau de taille fixe
- **slice** : tableau de taille variable
- **map** : table de hachage
- **struct** : type de données personnalisé

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Déclarer une variable

- Déclaration simple, puis initialisation

```
var maChaine string  
maChaine = "test"
```

- Déclaration et initialisation en une seule ligne

```
maChaine := "test"
```



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Déclarer une fonction

- Mot clé **func**
- Nom de fonction
- Argument Type, séparés par des parenthèses
- Retours

```
func maFunc(arg1 string, arg2 string) (string, err){}
```

Pour les types identiques, on peut aussi écrire :

```
func maFunc(arg1, arg2 string) (string, error){}
```



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Définir une méthode

- Il faut ajouter un champ qu'on appelle "receiver"
- Attention au receiver ! Pointeur, ou valeur !

```
type Example struct{}  
func (ex *Example) maFunc(arg1, arg2 string) (string, error){}
```



# L'ARCHITECTURE D'UN PROGRAMME

## Conditions en Go :

```
if condition {
    // instructions à exécuter si la condition est vraie
} else {
    // instructions à exécuter si la condition est fausse
}
```

```
age := 30
if age >= 18 {
    fmt.Println("Vous êtes majeur")
} else {
    fmt.Println("Vous êtes mineur")
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Boucles en Go : for

```
for initialisation; condition; incrément {  
    // instructions à exécuter à chaque itération  
}
```

```
for i := 0; i < 5; i++ {  
    fmt.Println(i)  
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Boucles en Go : while (presque.)

```
for condition {
    // instructions à exécuter tant que la condition est vraie
}
```

```
i := 0
for i < 5 {
    fmt.Println(i)
    i++
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Le code : avancé

- Les types de données
  - Les types "value"
  - Les types "headers"
- Les slices
- Types personnalisés
- Les "zero-value"
- Lecture des arguments passés au programme

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Les types de données

- Go a deux grand "types" de données :
  - Types "value"
  - Types "headers"
- Types "value"
  - Ce sont des types qui désignent directement la valeur qu'ils décrivent.
  - Exemple : les int, rune (équivalent de "char", alias de int32 en go), bytes (alias de int8 en go), bool...

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Les types de données

- Types "headers"
  - Ce sont des types qui comportent des références vers les valeurs décrites.
  - Exemple : Les string. C'est un groupe de deux valeurs : une adresse sur le tas, et une longueur.
  - Autre exemple : Les slices.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Les slices

- Ce sont des "vues", sur des tableaux.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## La slice : un type qui a trois valeurs

- Un pointeur vers le premier élément du tableau
- Une longueur (length)
  - Nombre d'éléments auxquels on a accès depuis le pointeur vers le premier élément. On l'obtient avec `len(maSlice)`
- Une capacité (capacity)
  - Nombre d'éléments existant dans le tableau sous-jacent. On l'obtient avec `cap(maSlice)`

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

```
package main
import (
    "fmt"
)
func main() {
    orig:=[]string{"one", "two", "three", "four"}
    sl1:=test[0:2]
    sl2:=test[1:3]
    modifSlice(sl1, 1)
    fmt.Printf("Ma slice 2: %v", sl2)
}

func modifSlice(sl []string, idx int){
    sl[idx] = "modified"
}
```



# L'ARCHITECTURE D'UN PROGRAMME

## Vrai tableau ?

Est-ce qu'on peut faire de "vrais" tableaux, et pas des slices ?

Oui ! Comme ça :

```
vraiTableau:=[2]int{1,8}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Définir son propre type

- On peut définir des types qui contiennent plusieurs champs, à la manière des classes.
- Attention, les types n'ont (presque toujours) que de la donnée à l'intérieur, pas de méthodes !
- Exemple :

```
type ExempleType struct{
    age int
    nom string
}
```



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Définir son propre type

On peut aussi définir des alias de types:

- types interchangeables :

```
type T = string
```

- nouveaux types :

```
type T string
```



# L'ARCHITECTURE D'UN PROGRAMME

## Les "zero-value"

- En go, l'initialisation des variables est automatique si elle n'est pas explicite
- Chaque type a une "zero-value", qui correspond à la valeur par défaut du type
- Exemples : 0 pour int, false pour bool, etc.
- Fonctionne pour tous types y compris structs, et "types header" !

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Lecture des arguments

- On peut lire les arguments passés à un programme par une commande comme :

```
go run main.go "image.png"
```



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

## Lecture des arguments

- Pour cela, il faut importer le package "os", et récupérer la slice de string "Args" :

```
argsWithProg := os.Args
argsWithoutProg := os.Args[1:]
arg := os.Args[3]
```



# L'ARCHITECTURE D'UN PROGRAMME

## Premier programme

- Dossier du projet

```
mkdir hello && cd hello
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Premier programme

- Initialisation d'un module : création du fichier go.mod

```
go mod init hello
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Premier programme

- Déclaration du package, et fonction main

```
package main
func main(){}

```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Premier programme

- Imports et appel

```
package main
import "fmt"
func main(){
    fmt.Println("Hello, World!")
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## On itère

- Est-ce qu'on pourrait faire exécuter une commande système à notre programme ?
- Essayons avec la commande "date", pour afficher la... date.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## On itère

- Le package os/exec :
  - <https://golang.org/pkg/os/exec/>
- Le type Cmd:
  - Cmd represents an external command being prepared or run.
  - A Cmd cannot be reused after calling its
    - Run
    - Output
    - CombinedOutput
    - methods

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

- La fonction Command :

```
func Command(name string, arg ...string) *Cmd
```

- La méthode Output :

```
func (c *Cmd) Output() ([]byte, error)
```



# L'ARCHITECTURE D'UN PROGRAMME

- Revenons à notre exemple :

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

- On modifie la fonction :

Création de Cmd

```
package main
import "fmt"
func main(){
    out, err := exec.Command("date").Output()
    fmt.Println("Hello, World!")
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP

- Gestion des erreurs :

```
package main
import "fmt"
func main(){
    out, err := exec.Command("date").Output()
    fmt.Println("Hello, World!")
}
```

- Ce code ne compile pas !



# L'ARCHITECTURE D'UN PROGRAMME

- Affichage du message de sortie :

```
package main
import (
    "fmt"
    "log"
    "os/exec"
)

func main(){
    out, err := exec.Command("date").Output()
    if err!=nil {
        log.Fatal(err)
    }
    fmt.Printf("The date is: %s\n", out)
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# L'ARCHITECTURE D'UN PROGRAMME

## Pourquoi ça fonctionne ?

- "date" fonctionne parce qu'elle est disponible sous Windows et sur les systèmes type Unix.
- Comment faire pour une commande qui dépend de l'OS ?
  - Réponse : la cross-compilation, et les instructions de compilation

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
**L'architecture d'un programme**  
Golang, les Maps  
Les bases - TP



# INTRODUCTION

Golang, origine

Les outils intégrés

La gestion de la mémoire

L'architecture d'un programme

**Golang, les Maps**

Les bases - TP



# GOLANG, LES MAPS

## Les maps

- C'est un header type
- Syntaxe :

```
map[Type_des_clés]Type_des_values
```

- Initialiser avec des valeurs :

```
m := map[string]int{"bob": 5}
```

- Initialiser vide :

```
m := make(map[string]int)
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
**Golang, les Maps**  
Les bases - TP



# GOLANG, LES MAPS

## Les maps

- Récuperer des valeurs (attention, si la clé est absente, on récupère la zero-value du type des valeurs !) :

```
valeur := m["bob"]
```

- Bonus : Récupérer une valeur et faire quelque chose si et seulement si la clé était présente :

```
if val, ok := m["bob"]; ok {  
    //do something here  
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
**Golang, les Maps**  
Les bases - TP



# INTRODUCTION

Golang, origine

Les outils intégrés

La gestion de la mémoire

L'architecture d'un programme

Golang, les Maps

**Les bases - TP**



# LES BASES - TP

## Mise en place de l'environnement

- Sous Linux, vous pouvez installer Golang à partir du gestionnaire de paquets.
- Pour les autres systèmes, ou si la détection échoue, vous pouvez vous rendre sur cette URL et choisir le fichier qui vous correspond :
  - <https://golang.org/dl/>
- Vous pouvez également suivre la procédure d'installation de Golang à cette adresse :
  - <https://golang.org/doc/install>
- Une fois que c'est fait, vous pouvez vérifier la version avec la commande :

```
go version
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Enregistrez votre travail

- Utilisez votre compte Gitlab pour y stocker votre code

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Hello World

- Reprenez l'exemple du cours, et écrivez votre propre "Hello, World!" en Go.  
Écrivez-le dans un fichier **main.go**
- Testez votre programme :
  - Soit en le compilant puis en le lançant :

```
go build ./...
./main.go
```

- Soit à la manière d'un "script" :

```
go run main.go
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Installation

- Vous pouvez **installer** votre application sur votre machine avec la commande qui suit, il faut l'exécuter depuis la racine de votre projet :

```
go install
```

- Cela va placer votre binaire dans **\$GOPATH/bin**. Une fois cela fait, vous pouvez exécuter votre programme depuis n'importe où.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Go fmt

- Testez le formateur de code "go fmt" en formatant ce code :

```
package main
import "fmt"

func main() {
    fmt.Printf( "Bad " ) ;
    fmt.Println("formating" )
}
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard

- Nous allons nous servir de quelques packages de la librairie standard, dont la documentation est disponible ici :  
<https://golang.org/pkg/>
- On va réaliser un petit programme qui compare des images et dit si elles sont identiques.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard - Flag

- On va utiliser le package <https://golang.org/pkg/flag/> pour permettre d'afficher la version de notre programme.
- Définissez une constante dans le fichier main.go, comme ceci :

```
const VERSION = "1.0"
```

- Utilisez flag.Bool() et flag.Parse() pour afficher la version du programme lorsque l'on entre :

```
$ go run main.go -version
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard - Crypto

- Téléchargez les images disponibles sur l'espace de cours : image\_1.jpg, image\_2.jpg, image\_3.jpg
- Créez un projet img\_diff, et initialisez-le avec :

```
go mod init img_diff
```

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

- Écrivez une fonction qui permet de lire un fichier complètement, et d'en retourner les octets sous forme de `[]byte`. Utilisez le package <https://golang.org/pkg/io/ioutil/>
- Écrivez une fonction qui permet de hasher un fichier donc vous lui passez le path en argument. Cette fonction retourne un `[]byte`. Utilisez le package <https://golang.org/pkg/crypto/sha256/>
- Comparez les hashes des images, et déterminez celle qui est unique.
- Affichez le résultat (le path de l'image unique tel que passé en argument) dans la console.
- Pourriez-vous imaginer une implémentation plus efficace ?
  - Implémentez-la en utilisant <https://golang.org/pkg/bufio/>

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard - Logs

- On va ajouter du logging pour les erreurs. Pour cela, utilisez :  
<https://golang.org/pkg/log/>
- Créez un nouveau logger, et dirigez son output sur **os.Stderr**  
(<https://golang.org/pkg/os/#pkg-variables>)
- Partagez votre logger dans votre programme, et loguez les erreurs éventuelles.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard - Http

- On va ajouter une possibilité de récupérer une liste d'images sur internet et de les comparer.
- Ajoutez un flag, comme pour la version, qui permet de passer une chaîne de caractères en argument qui est une liste d'URLs séparées par des virgules :  
`"http://url1.tld,http://url2.tld"`
- Ajoutez un parsing de cette chaîne de caractères en utilisant :  
`https://golang.org/pkg/strings/`  
Votre parser d'argument retourne un `[]string`, qui est une slice d'URLs.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

- Utilisez `http.Get()` pour télécharger les images. Vous pouvez lire le contenu téléchargé avec :

```
data, err := ioutil.ReadAll(resp.Body)

if err != nil {
    log.Fatal(err)
}

defer resp.Body.Close()
```

- Ecrivez les images sur le disque avec `os.Create`
- Vérifiez le bon fonctionnement de votre programme

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# LES BASES - TP

## Bibliothèque standard - Time

- Ajoutez des logs qui permettent de savoir combien de temps mettent les images téléchargées à être récupérées depuis internet.  
Utilisez: <https://golang.org/pkg/time/>
- Et utilisez: <https://golang.org/pkg/fmt/> pour les messages. Formattez avec `%v`.

Golang, origine  
Les outils intégrés  
La gestion de la mémoire  
L'architecture d'un programme  
Golang, les Maps  
**Les bases - TP**



# INTRODUCTION

Dans ce chapitre nous avons :

- Les bases du langage Golang.
- Les outils intégrés au langage.
- La gestion de la mémoire.
- L'architecture des programmes Go.
- Découvert les maps en Go.
- Vu les fonctionnalités de base du langage Golang.



CHAPITRE 2

# INTERFACES ET JSON



# INTERFACES ET JSON

- **Golang, rappels** récapitulera certaines notions vues précédemment.
- **Golang, interfaces** nous présentera la gestion des interfaces Go.
- **Golang, JSON** nous présentera l'utilisation du JSON en Go.
- **Les interfaces - TP** nous permettra de tester les interfaces Golang.
- **Le JSON - TP** nous permettra de découvrir l'intérêt du JSON en Golang.



# INTERFACES ET JSON

**Golang, rappels**

Golang, interfaces

Golang, JSON

Les interfaces - TP

Le JSON - TP



# GOLANG, RAPPELS

## Rappels

- On a deux catégories de types en Go : value et headers
  - Un type value décrit directement la valeur correspondante (un bool est bien un boolean, et pas une adresse vers un boolean)
  - Un type header contient des informations sur les valeurs qu'il décrit. On peut bien sûr récupérer ces valeurs.

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

## Rappels

- Exemple de type header : les slices (trois valeurs: pointeur vers le premier élément, longueur, capacité)
- Les zero values : ce sont les valeurs par défaut des types en Go.

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

## Rappels pointeurs

- Les pointeurs sont des adresses mémoire
- Ils désignent un emplacement, où trouver "autre chose"
- Cela peut être une valeur, ou un autre pointeur etc...

Mais alors pourquoi utiliser des pointeurs ?

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

## Qu'affiche ce code ?

```
package main
import (
    "fmt"
)
func main() {
    ex:= Example{age: 10}
    modifAge(ex, 43)
    fmt.Println(ex)
}

type Example struct{
    age int
}

func modifAge (ex Example, age int){
    ex.age=age
}
```

Thomas Saquet - Golang - Interfaces

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

Résultat :

{10}

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

## Pourquoi ?

- Go est "pass by value"
  - C'est à dire que quand on passe une variable à une fonction :
    - une copie est faite
    - c'est cette copie qui est utilisée dans la fonction
- Comment "réparer" l'exemple précédent ?  
=> Avec des pointeurs !

**Golang, rappels**  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, RAPPELS

[Golang, rappels](#)  
[Golang, interfaces](#)  
[Golang, JSON](#)  
[Les interfaces - TP](#)  
[Le JSON - TP](#)

```
package main
import (
    "fmt"
)
func main() {
    ex:= Example{age:10}
    modifAge(&ex, 43)
    fmt.Println(ex)
}

type Example struct{
    age int
}

func modifAge(ex *Example, age int){
    ex.age=age
}
```



# INTERFACES ET JSON

Golang, rappels

**Golang, interfaces**

Golang, JSON

Les interfaces - TP

Le JSON - TP



# GOLANG, INTERFACES

- C'est quoi une interface ?
- Un contrat :
  - Une liste de méthodes que doit respecter une entité qui l'implémente.
  - Si l'entité implémente ces méthodes, elle est alors aussi du type que cette interface définit.

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

- Une interface permet de grouper des types concrets par fonctionnalité
- Les interfaces peuvent aider pour tester le code
- Une interface est vérifiée implicitement :
  - Si toutes les méthodes d'une interface sont implémentées pour un type, alors ce type vérifie l'interface

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

- Zero value : que se passe-t-il si on execute ce programme ?

```
package main

func main() {
    var ifce IfceTest
    ifce.test()
}

type IfceTest interface {
    test()
}
```

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP

=> Panic



# GOLANG, INTERFACES

- La zero value d'une interface est un pointeur nil
- Un pointeur nil, pour rappel, c'est une absence de pointeur.
- Appeler une méthode sur un pointeur nil, c'est appeler une méthode sur "rien"

=> Panic

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

- Comment initialiser l'interface dans notre exemple précédent ?

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP

```
package main

func main()
{
    var ifce IfceTest
    ifce = Example{}
    ifce.test()
}

type IfceTest interface {
    test()
}

type Example struct{}

func (ex Example)test(){}

```



# GOLANG, INTERFACES

- Si on a besoin de récupérer le type qui se cache sous une interface, au runtime ?

=> C'est possible avec un switch :

```
func printType(i interface{}) {
    switch v := i.(type) {
        case int:
            fmt.Println("The type is int !")
        case string:
            fmt.Println("The type is string !")
        default:
            fmt.Printf("I don't know about type %T!\n", v)
    }
}
```

Golang, rappels  
[Golang, interfaces](#)  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

- Comment vérifier qu'une interface "contient" bien un type (sans switch) ?

```
package main

import
(
    "log"
)

func main()
{
    var ifce Ifce
    ifce = Example{}
    _, ok := ifce.(Example)
    if !ok {
        log.Fatal("Pas le type attendu !")
    }
}

type Ifce interface {
    test()
}

type Example struct {}

func (ex Example)test() {}
```

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# GOLANG, INTERFACES

- En écriture condensée :

```
package main

import "log"

func main() {
    var ifce Ifce
    ifce = Example{}


    if _, ok := ifce.(Example); !ok {
        log.Fatal("Pas le type attendu !")
    }
}

type Ifce interface {
    test()
}

type Example struct {}

func (ex Example)test() {}
```

Golang, rappels  
**Golang, interfaces**  
Golang, JSON  
Les interfaces - TP  
Le JSON - TP



# INTERFACES ET JSON

Golang, rappels

Golang, interfaces

**Golang, JSON**

Les interfaces - TP

Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- JSON est très utilisé dans les APIs de nos jours, et existe aussi dans les logs
- Parsing intégré dans la librairie standard
- Parsing se fait par annotations

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- Exemple de données au format JSON :

```
{  
    "menu":{  
        "id":"file",  
        "value":"File",  
        "popup":{  
            "menuitem": [  
                { "value": "New", "onclick": "CreateNewDoc()" },  
                { "value": "Open", "onclick": "OpenDoc()" },  
                { "value": "Close", "onclick": "CloseDoc()" }  
            ]  
        }  
    }  
}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- Quels avantages ?
  - Simple à manipuler pour un programmeur
  - Lisible par un humain, léger pour les machines
  - "Facile à apprendre" parce que la syntaxe n'est pas extensible

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- Quels inconvénients ?
  - Syntaxe non extensible (contrairement à du XML par exemple)
  - Le typage limité affaiblit la sécurité
  - On ne peut pas toujours commenter du JSON (dépend du parser)

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- Comment générer du JSON ?
  - Créer une struct qui correspond aux données que l'on veut en sortie
  - Utiliser des annotations si nécessaire
  - Utiliser <https://golang.org/pkg/encoding/json/#Marshal>

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

- Qu'affiche ce programme ?

```
package main

import (
    "encoding/json"
    "fmt"
)
func main() {
    group := ColorGroup {
        ID: 1,
        Name: "Reds",
        Colors: []string {"Crimson", "Red", "Ruby", "Maroon"},
    }
    b, err := json.Marshal(group)
    if err != nil {
        fmt.Println("error:", err)
    }
    fmt.Println(string(b))
}

type ColorGroup struct {
    ID int
    Name string
    Colors []string
}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

- Comment déserialiser du JSON ?
  - Créer une struct qui correspond aux données que l'on veut lire
  - Utiliser des annotations si nécessaire
  - Utiliser <https://golang.org/pkg/encoding/json/#Unmarshal>

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP

```
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    var jsonBlob = []byte([
        `[
            {"Name": "Platypus", "Order": "Monotremata"},  

            {"Name": "Quoll", "Order": "Dasyuromorphia"}
        ]`)
    var animals []Animal
    err := json.Unmarshal(jsonBlob, &animals)
    if err !=nil {
        fmt.Println("error:", err)
    }
    fmt.Printf("%+v", animals)
}

type Animal struct{
    Name string
    Order string
}
```



# GOLANG, JSON

## Parsing JSON

Importants : seuls les champs exportés seront serialisés/déserialisés.

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

## Parsing JSON

### Les annotations

- Les annotations permettent de redéfinir les noms des champs entre JSON et struct.
- Exemple :

```
type Animal struct {
    Species string `json:"Name"`
    Order   string `json:"Order"`
}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

- On peut aussi spécifier que l'on ne veut pas que des champs vides soient ajoutés dans les JSON générés :

```
type Animal struct {
    Species string `json:",omitempty"`
    Order   string
}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

Qu'affiche ce programme ?

```
package main
import (
    "encoding/json"
    "fmt"
)

func main() {
    type ColorGroup struct {
        ID int
        Name string
        Colors []string `json:",omitempty"`
    }
    group := ColorGroup {
        ID: 1,
        Name: "Reds",
    }
    b, err := json.Marshal(group)
    if err != nil {
        fmt.Println("error:", err)
    }
    fmt.Println(string(b))
}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# GOLANG, JSON

Résultat :

```
{"ID":1,"Name":"Reds"}
```

Golang, rappels  
Golang, interfaces  
**Golang, JSON**  
Les interfaces - TP  
Le JSON - TP



# INTERFACES ET JSON

Golang, rappels

Golang, interfaces

Golang, JSON

**Les interfaces - TP**

Le JSON - TP



# LES INTERFACES - TP

Reprenez le code suivant :

```
package main

import "fmt"

type IPAddr [4]byte

func main() {
    hosts := map[string]IPAddr{
        "loopback": {127, 0, 0, 1},
        "googleDNS": {8, 8, 8, 8},
    }
    for name, ip := range hosts {
        fmt.Printf("%v: %v\n", name, ip)
    }
}
```

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP



# LES INTERFACES - TP

## Interface Stringer

- Implémentez `fmt.Stringer` pour `IPAddr`, de façon à afficher l'adresse IP comme une suite de nombres séparés de points.  
Autrement dit, la slice `IPAddr{1, 2, 3, 4}` devrait s'afficher de la façon suivante : **"1.2.3.4"**.
- Implémentez ceci de deux façons :
  - En utilisant le package <https://golang.org/pkg/fmt/> et `fmt.Sprintf()`
  - En utilisant le package <https://golang.org/pkg/strconv/>

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP



# LES INTERFACES - TP

## Interface Error (1/3)

- On va implémenter une erreur "custom". Pour cela, remarquez que les erreurs en Go sont simplement des entités qui implémentent l'interface :  
<https://golang.org/pkg/builtin/#error>

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP



# LES INTERFACES - TP

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP

## Interface Error (2/3)

- Définissez une struct "**MyError**" qui contient deux champs :
  - "When" de type `time.Time` (voir <https://golang.org/pkg/time/#Time>)
  - "What" de type `string`
- Implémentez l'interface `Error` pour votre struct "**MyError**".
- Écrivez une fonction "**run()**", dont la signature est :

```
func run() error
```

qui retourne systématiquement une erreur comme celle que vous venez de créer.

- Pour donner une valeur au champ du temps, utilisez "**Now()**", du package `time` de la librairie standard.
- Pour donner une valeur au champ du "**What**", choisissez un texte d'erreur non vide qui vous plaît.



# LES INTERFACES - TP

## Interface Error (3/3)

- Dans la fonction main de votre programme, exécutez la fonction "**run()**" que vous venez de définir, et récupérez l'erreur en l'assignant à une variable, comme ceci par exemple :

```
err := run()
```

- Vérifiez si votre erreur est vide (elle ne devrait jamais l'être) en écrivant un "**if**", et si elle ne l'est pas, affichez-la dans la console. Pour tester si l'erreur est "**vide**", basez-vous sur le fait qu'une erreur est une interface, et comparez la valeur de votre erreur à la zero value d'une interface.



# LES INTERFACES - TP

## Interface vide (1/3)

On va à présent étudier l'interface vide, et la détermination des types sous-jacents :

```
interface{}
```

- Écrivez une fonction qui a la signature suivante, et qui se contente d'exécuter `fmt.Println()` sur l'argument qui lui est passé :

```
func PrintIt(input interface{})
```



# LES INTERFACES - TP

## Interface vide (2/3)

- Essayez d'exécuter cette fonction en passant un entier en argument. (c'est à dire : appelez cette fonction dans votre fonction main et lancez votre programme).  
Que se passe-t-il ?
- Essayez à présent avec une chaîne de caractères. Même question.

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP



# LES INTERFACES - TP

## Interface vide (3/3)

- L'interface vide est un contrat qui n'a aucune méthode. Donc pour l'implémenter, n'avoir aucune méthode sur un type suffit.
- Tous les types en Go ont 0 ou plus méthodes, donc tous les types vérifient l'interface vide.
- Modifiez à présent votre fonction **PrintIt()** pour qu'elle affiche le type de l'argument passé, et non plus sa valeur. Utilisez le switch vu en cours.

Golang, rappels  
Golang, interfaces  
Golang, JSON  
**Les interfaces - TP**  
Le JSON - TP



# INTERFACES ET JSON

Golang, rappels

Golang, interfaces

Golang, JSON

Les interfaces - TP

Le JSON - TP



# LE JSON - TP

## Parsing pur (1/2)

Soit la structure User suivante :

```
type User struct {
    Login string
    Password string
}
```

- Afficher dans le terminal le résultat de la sérialisation d'un utilisateur dont le login est "Paul" et le mot de passe "pass123".
- Remplacer le nom du champ "Login" dans le JSON de sortie par "userName" en utilisant les annotations.



# LE JSON - TP

## Parsing pur (2/2)

- Soit **users.json** un fichier contenant une liste d'utilisateurs au format JSON :

```
[{"userName": "matm", "Password": "123456"}, {"userName": "fake44", "Password": "azerty"}]
```

- Lire puis déserialiser le contenu de ce fichier dans une liste de **User**.
- Que se passe-t-il si vous renommez "Password" en "password" dans votre structure et que vous lancez à nouveau le programme précédent ?



# LE JSON - TP

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**

## Serveur HTTP - Les données (1/2)

On va écrire dans cette partie un petit serveur HTTP qui permet d'obtenir des information sur des utilisateurs, en passant des query. Voir :

[https://en.wikipedia.org/wiki/Query\\_string](https://en.wikipedia.org/wiki/Query_string)

- Ajoutez dans votre fichier users.json un champ "userID" aux utilisateurs de la liste, et donnez lui une valeur différente pour chaque utilisateur.
- Ajoutez dans la struct User définie précédemment un champ permettant de correctement récupérer le user ID lorsque l'on désérialise le JSON du fichier.



## LE JSON - TP

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**

### Serveur HTTP - Les données (2/2)

- Créez une map, sous forme de "variable globale", que vous remplirez dès que la fonction main sera exécutée en lisant votre fichier JSON.
- Le type de cette map est :

`map[string]User`

avec en clé le user ID et en valeur le user correspondant.

- Pour définir une "variable globale", définissez-la en dehors de toute fonction. Elle est alors accessible dans toutes les fonctions de votre fichier **main.go**



# LE JSON - TP

## Serveur HTTP - Fonction Handler (1/3)

- Définissez un **handler** pour votre serveur. Un handler est une fonction qui a cette signature :

```
func(http.ResponseWriter, *http.Request)
```

avec l'interface:

<https://golang.org/pkg/net/http/#ResponseWriter> et la struct :

<https://golang.org/pkg/net/http/#Request>

comme arguments. Laissez le corps de la fonction vide pour l'instant.

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**



## LE JSON - TP

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**

### Serveur HTTP - Fonction Handler (2/3)

- Définissez un handler dans votre fonction main, en utilisant `http.HandleFunc` de :  
<https://golang.org/pkg/net/http/#HandleFunc>
- Utilisez le pattern "/" dans votre appel à cette fonction, et passez lui comme second argument la fonction définie au point précédent.
- Écrivez l'appel qui permettra de lancer votre serveur en utilisant :  
<https://golang.org/pkg/net/http/#ListenAndServe>



# LE JSON - TP

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**

## Serveur HTTP - Fonction Handler (3/3)

- Lorsque vous faites une requête http au serveur que nous sommes en train d'écrire, le serveur appelle notre fonction handler. On a accès à la requête envoyée par le client par l'objet **\*http.Request** passé en argument de votre fonction de handler, et on peut écrire notre réponse sur le **http.ResponseWriter**.
- Écrivez du code qui vérifie que le user passé dans la requête est bien dans notre map, définie en variable globale. Pour cela, utilisez :

```
id := r.FormValue("id")
```

qui permet de récupérer le champ de query "**id**" dans une requête HTTP de type Get à l'adresse :  
**http://localhost:8000/?id=id1**



## LE JSON - TP

### Serveur HTTP - Sérialisation (1/3)

- Si l'id est trouvé, sérialisez le User correspondant en JSON.  
Autrement, ne faites rien.
- Écrivez le header de votre réponse avec :

```
w.Header().Set(  
    "Content-Type",  
    "application/json; charset=utf-8",  
)
```

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**



## LE JSON - TP

### Serveur HTTP - Sérialisation (2/3)

- Ajoutez ensuite le code de la réponse avec :

```
w.WriteHeader(http.StatusOK)
```

lorsque le user existe et :

```
w.WriteHeader(http.StatusNotFound)
```

lorsqu'il est introuvable.

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**



## LE JSON - TP

### Serveur HTTP - Sérialisation (3/3)

- Enfin, écrivez le JSON du User (si votre code réponse n'est pas "Not Found") dans le ResponseWriter avec :

```
Write([]byte) (int, error)
```

de:

<https://golang.org/pkg/net/http/#ResponseWriter>

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**



## LE JSON - TP

### Serveur HTTP - Requête

- Lancez votre serveur, et testez-le avec des requêtes curl comme :

```
curl -i http://localhost:8000/?id="id1"
```

Golang, rappels  
Golang, interfaces  
Golang, JSON  
Les interfaces - TP  
**Le JSON - TP**



# INTERFACES ET JSON

Dans ce chapitre nous avons :

- Rappelé certaines notions déjà abordées.
- Compris la gestion des interfaces Go.
- Etudié comment utiliser le JSON en Go.
- Testé les interfaces dans le langage Golang.
- Appris à utiliser le JSON en Golang.



# CHAPITRE 3

# CONCURRENCE



# CONCURRENCE

- **Concurrence, les threads** nous présentera la gestion de la concurrence en Go.
- **Concurrence, les channels** nous présentera les channels en Go.
- **La cross compilation** nous expliquera la cross compilation.
- **Concurrence - TP** nous permettra de tester la concurrence en Go.



# CONCURRENCE

Concurrence, les threads

Concurrence, les channels

La cross compilation

Concurrence - TP



# CONCURRENCE, LES THREADS

## Concurrence et parallélisme

- Quelle différence ?
  - Concurrence: effectuer plusieurs tâches sans attendre d'en finir une pour en commencer une autre.  
Ex : écrire deux textes différents sur deux feuilles de papier, en écrivant une ligne de chaque texte à la fois.

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

- Quelle différence ?
- Parallélisme: effectuer plusieurs tâches simultanément.  
Ex : écrire les deux textes en même temps, avec un stylo dans chaque main.

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Rappels sur les threads

- Un thread est un "fil d'exécution" de votre programme.
- Il existe deux types de threads : les "threads lourds", ou "threads système", et les green-threads, ou "threads légers"

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Rappels sur les threads

- Un thread lourd est un thread géré par l'OS. Il est dit lourd parce que généralement plus coûteux en ressources
  - Il faut communiquer avec l'OS pour le manipuler.
- Un green thread est plus léger, il est géré par le runtime du programme.

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Pourquoi les threads ?

- Pour les performances
- Pour les opérations bloquantes

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Threads, inconvénients

- La programmation d'un code multithreaded est plus difficile.
- Il y a des risques qui n'existent pas dans la programmation non-threaded. Ex : deadlock, data races...

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Threads, mémoire

- La mémoire n'est pas automatiquement synchronisée entre les threads.
- Si on partage un objet entre plusieurs threads, cela peut causer des problèmes (data races).

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

**Concurrence, les threads**  
 Concurrence, les channels  
 La cross compilation  
 Concurrence - TP

## Threads, mémoire

- Comment est gérée la mémoire ?
- Exemple pour illustrer les data races, sur l'opération : `compteur = compteur+1`

Thread 1	Thread 2	Integer value
		0
read value	←	0
increase value		0
write back	→	1
	read value	1
	increase value	1
	write back	2

Pas de data race

Thread 1	Thread 2	Integer value
		0
read value	←	0
	read value	0
increase value		0
	increase value	0
write back	→	1
	write back	1

Data race



# CONCURRENCE, LES THREADS

## Threads, mémoire

- Comment synchroniser ?
- Avec des mutexes
- Avec des channels => En Go on fait plutôt ça.  
=> "Partagez de la mémoire en communiquant, et non pas, partagez de la mémoire pour communiquer."

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Mutex

- Ce sont des objets qui permettent de réguler l'accès aux données.
  - Ils permettent par exemple qu'un seul thread y accède à la fois.
- Penser à un verrou. Lorsqu'un thread accède à une donnée, elle ferme le verrou et garde la clé.
  - Lorsque les opérations sont finies pour la donnée, ne pas oublier de libérer le verrou !
- Deadlock ?

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP

## Threads en go

- Ils sont appelés "goroutines".
- Ce sont des green threads : le runtime se charge de tout pour nous.  
Lorsqu'un thread lourd est nécessaire pour l'exécution du programme, le runtime se charge de le créer pour nous.
- Attention quand le thread de votre fonction **main** se termine, le programme se termine aussi...
  - ... que les autres threads soient finis ou non !



# CONCURRENCE, LES THREADS

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP

## Threads en go : goroutine

- Comment les lancer ?
- Mot clé "go"

```
func main(){
    go maFunc() // Lance une goroutine qui exécute "maFunc()"
}

func maFunc(){
    fmt.Println("je ne fais rien !")
}
```



# CONCURRENCE, LES THREADS

## Threads en go

Pourquoi est-ce que le programme précédent n'affiche quasiment jamais rien ?

- La goroutine qui doit print est lancée, mais le thread de main() se termine immédiatement, et donc le programme aussi.
- La goroutine qui doit print est tuée avant d'avoir pu afficher quoi que ce soit.

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES THREADS

## Threads en go

On va parler des channels, mais pas des mutexes en Go.

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
Concurrence - TP



# CONCURRENCE

Concurrence, les threads

**Concurrence, les channels**

La cross compilation

Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Threads en go : channels

- C'est un type ! Un type header.
- Sa zero value est "nil"
- Comme les maps, on les crée avec "make"
  - Rappel :

```
make(map[string]string)
```

- Pour les channels :

```
make(chan [TYPE QUI PASSE DANS LE CHANNEL])
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Threads en go : channels

- Il existe deux types de channels :
  - Buffered channels
  - Non-buffered channels

Référence :

[https://golang.org/doc/effective\\_go.html#channels](https://golang.org/doc/effective_go.html#channels)

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP

## Channels : non-buffered

- Penser à un tuyau, dans lequel on peut envoyer de l'information.
- Cette information est typée à l'avance :

```
ch := make(chan int)
```

Dans ce channel, on ne peut envoyer que des int.



# CONCURRENCE, LES CHANNELS

## Channels : non-buffered

- Comment écrire de l'information dans un channel ?

```
information := 0
ch := make(chan int)

ch <- information
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Channels : non-buffered

- Comment lire de l'information dans un channel ?

```
information := 0
ch := make(chan int)

ch <- information

informationLue := <-ch
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Channels : non-buffered

- L'écriture et la lecture sont bloquantes.
- Quand on écrit une valeur sur le channel, on ne peut rien faire avec tant que la valeur n'a pas été lue à l'autre bout.
- Même type de raisonnement en lecture

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP

## Channels : buffered

- Tout pareil que les non buffered channels, mais on a une file d'attente à l'intérieur du channel

```
// On peut mettre jusqu'à 10 int dedans sans bloquer. Leur ordre est préservé.  
ch := make(chan int, 10)
```



# CONCURRENCE, LES CHANNELS

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP

## Select statement

- Comment signaler qu'une goroutine a terminé son travail ?

```
func main() {
    ch := make(chan bool)
    go maFonction(ch)
    select {
        case <-ch:
            fmt.Println("j'ai fini !")
        case <-time.After(time.Second):
            fmt.Println("j'ai pas fini, mais tant pis !")
    }
}

func maFonction(ch chan bool) {
    time.Sleep(3*time.Second)
    ch<-true
}
```



# CONCURRENCE, LES CHANNELS

## wait groups

- Un wait group est une struct, donc un type value (pas un type header !)
- Il sert à attendre qu'un groupe de goroutines aient fini leurs tâches avant d'avancer dans le programme.
  - Il bloque donc le programme.

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## wait groups

- On commence par le déclarer :

```
var wg sync.WaitGroup // La zero value suffit
```

- Puis quand on lance une goroutine (juste avant),
  - on ajoute un incrément au WaitGroup
  - et on le passe en argument de la fonction :

```
wg.Add(1)  
go maFunc(&wg)
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## wait groups

- Il faut décrémenter ce compteur lorsque la goroutine a fini son execution :

```
func maFunc(wg *WaitGroup){  
    defer wg.Done()  
}
```

- Enfin pour bloquer le programme jusqu'à la fin de l'exécution des goroutines, placer un Wait() à l'endroit choisi :

```
func main(){  
    // Tout le reste ici  
    wg.Wait() // Cette ligne fait attendre que les goroutines soient achevées  
    // Suite des instructions  
}
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## wait groups

- Référence: <https://golang.org/pkg/sync/#WaitGroup>

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Closures

- Ce sont des fonctions "anonymes". Autrement dit elles n'ont pas de nom.
- Elles peuvent accéder à des variables définies en dehors de leur body :

```
func main() {
    l := 20
    b := 30
    func() { // C'est une closure
        var area int
        area = l - b
        fmt.Println(area)
    }()
}
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE, LES CHANNELS

## Closures

On peut directement les lancer dans des goroutines comme cela :

```
func main() {
    l := 20
    b := 30
    go func() { // C'est une closure lancée dans une goroutine
        var area int
        area = l - b
        fmt.Println(area)
    }()
}
```

Concurrence, les threads  
**Concurrence, les channels**  
La cross compilation  
Concurrence - TP



# CONCURRENCE

Concurrence, les threads

Concurrence, les channels

**La cross compilation**

Concurrence - TP



# LA CROSS COMPIRATION

## Cross compilation & static linking

- Possibilité de compiler pour "n'importe quelle" plateforme. Lister les targets possibles :

```
go tool dist list
```

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# LA CROSS COMPIRATION

## Cross compilation & static linking

- Les binaires sont "self contained", "statically linked" : aucune dépendance au runtime
- Pour choisir une plateforme, il suffit de définir une variable d'environnement :
  - GOARCH pour l'architecture
  - GOOS pour la plateforme

```
GOARCH=amd64 GOOS=linux go build ./...
```



# LA CROSS COMPIRATION

Comment spécifier au compilateur qu'on aimeraît compiler tel code pour tel OS ?

- En C par exemple, on peut mettre des headers.
- En Go, on a plusieurs méthodes : noms de fichiers et annotations.

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# LA CROSS COMPIRATION

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP

## Instructions compilateur et noms de fichiers

- Il suffit de nommer ses fichiers avec un suffixe qui est le target de la plateforme pour laquelle on veut compiler.
- On a la liste des plateformes avec la commande donnée précédemment.

```
go tool dist list
```



# LA CROSS COMPIRATION

## Exemple de structure

```
└── cross_compil_names
    ├── getCommand_linux.go
    ├── getCommand_windows.go
    ├── go.mod
    └── main.go
```

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# LA CROSS COMPIRATION

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP

## Fichier Linux (getCommand\_linux.go) :

```
func GetCommand() []string{
    return []string{"ifconfig", "-a"}
}
```

## Fichier Windows (getCommand\_windows.go) :

```
func GetCommand() []string{
    return []string{"ipconfig", "/all"}
}
```



# LA CROSS COMPIRATION

Fichier main (tronqué) :

```
cmd := GetCommand()
out,err := exec.Command(cmd[0], cmd[1:]...).Output()
```

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# LA CROSS COMPILATION

- Instructions compilateur et annotations :

- Il est également possible d'utiliser des annotations pour les instructions.
- C'est plus flexible, et plus "propre".

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# LA CROSS COMPILATION

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP

## Instructions compilateur et annotations :

- Supposons que l'on ait 4 targets différents à couvrir :  
MacOS, Windows (32bits), Openbsd, Linux.
- On pourrait faire 4 fichiers différents... Mais.



# LA CROSS COMPIRATION

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP

Plutôt que de faire 4 fichiers, on factorise et on utilise des annotations comme ceci :

- Fichier Unix (getCommand\_unix.go) :

```
// +build linux darwin openbsd
func GetCommand() []string{
    return []string{"ifconfig", "-a"}
}
```

- Fichier Windows (getCommand\_windows.go):

```
// +build windows,386
func GetCommand() []string{
    return []string{"ipconfig", "/all"}
}
```

- Fichier main : inchangé.



# LA CROSS COMPIRATION

Référence :

[https://golang.org/cmd/go/##hdr-Build\\_constraints](https://golang.org/cmd/go/##hdr-Build_constraints)

Concurrence, les threads  
Concurrence, les channels  
**La cross compilation**  
Concurrence - TP



# CONCURRENCE

Concurrence, les threads

Concurrence, les channels

La cross compilation

**Concurrence - TP**



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Concurrence simple et interfaces

- Considérez le code suivant :

```
package main

import (
    "fmt"
)

func main() {
    go maFonction()
    fmt.Println("Fin du programme")
}

func maFonction() {
    fmt.Println("j'ai fini !")
}
```



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Concurrence simple et interfaces - WaitGroups

- Modifiez-le pour que "**maFonction()**" ait toujours le temps de s'exécuter complètement avant la fin du programme.
- Pour cela, utilisez un WaitGroup :  
<https://golang.org/pkg/sync/#WaitGroup>
- Vous pouvez modifier la signature de la fonction "**maFonction()**" (c'est même encouragé).



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Channels, select et HTTP

- On va utiliser dans cette partie les concepts vus en cours aujourd'hui de channels et de select.
- On va également reprendre ce qu'on a vu dans le TP concernant le HTTP, mais en se plaçant cette fois du côté d'un client, et non plus d'un serveur.
- Définissez un type "**Reponse**", qui est une struct comprenant deux champs : **respText** de type string, et **err** de type error.
- Créez dans votre fonction **main()**, deux channels qui transportent des types "**Reponse**".



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Le principe du Callserver

- Écrivez une fonction "**callServer**" qui prend deux arguments :
  - une adresse, sous forme de chaîne de caractères
  - un channel qui transporte des "**Reponse**"



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## La fonction Get (1/3)

- Dans votre fonction callServer, réalisez une requête HTTP vers le serveur dont l'adresse a été passée en argument. Vous utiliserez pour cela la fonction "**Get**" du package http.
- Toujours dans **callServer**, gérez les réponses possibles du serveur :
  - Si l'appel à Get retourne une erreur, créez un objet "**Reponse**" et remplissez le champ "**err**". Envoyez cet objet dans le channel passé en argument et arrêtez l'exécution de la fonction avec "**return**".
  - Si la réponse que vous obtenez présente un status code HTTP qui est différent de 200, créez à nouveau un objet Reponse, et peuplez le champ "**err**" de celui-ci avec (c'est à mettre sur une seule ligne) :

```
errors.New("Le code retourné par le serveur indique une erreur: " + strconv.Itoa(resp.StatusCode))
```



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## La fonction Get (2/3)

- Lisez le corps de la réponse HTTP en utilisant le code suivant (adaptez avec les noms de variables que vous aurez utilisés) :

```
body, err := ioutil.ReadAll(resp.Body)
```



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## La fonction Get (3/3)

- Pensez à fermer le body de la réponse http avec un "**defer**".
- Si la lecture du code de la réponse retourne une erreur, créez un objet `Reponse`, mettez cette erreur dans le champ `err` et passez la `Reponse` dans le channel.
- Si tout s'est passé comme prévu, envoyez dans le channel un `Reponse` en peuplant le champ "**respText**" avec le contenu du body de la réponse du serveur. Pensez bien que `ioutil.ReadAll` qui vous a servi à lire le body retourne `[]byte` et non pas `string`. Il faut faire un "**cast**".



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Bientôt fini !

- On retourne à présent dans notre fonction main. On va faire les appels au serveur. Autrement dit, `main()` va utiliser votre fonction `callServer`.
- À la suite des déclarations des channels faites précédemment, ajoutez deux appels à `callServer` qui respectivement :
  - appelle l'adresse: "`http://localhost:8000/?id=id1`" et prend le channel 1 en argument
  - appelle l'adresse: "`http://localhost:8000/?id=id3`" et prend le channel 2 en argument
- À présent, lancez vos appels au serveur dans une goroutine chacun, avec :

```
go callServer(...) // Remplissez avec les bons arguments
```

Que se passe-t-il si vous lancez votre programme comme cela ? Pourquoi ?



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Channels (1/2)

- Pour attendre que les fonctions terminent sans utiliser de **waitgroup**, on va lire sur les channels.
- À la suite de vos appels au serveur, lisez les informations qui sont sur vos channels en utilisant :

```
<-ch1  
<-ch2
```



# CONCURRENCE - TP

Concurrence, les threads  
Concurrence, les channels  
La cross compilation  
**Concurrence - TP**

## Channels (2/2)

- Écrivez maintenant un select, qui lit pour chaque "case" sur un channel différent.  
Par ce biais, vous pourrez voir quel endpoint est le plus rapide à répondre puisque select va prendre le premier résultat mis à sa disposition.
- Quel est l'endpoint le plus rapide ?
- Quelle est la réponse de cet endpoint ?



# CONCURRENCE

Dans ce chapitre nous avons :

- Vu la gestion de la concurrence Golang.
- Etudié les channels en Go.
- La cross compilation en Go.
- Testé les différents aspects de la gestion de la concurrence dans le langage Golang.



**FIN**

Merci pour votre attention !