Bienvenue à la Sfeir School
GO 100

School

Déroulement de la formation

C’est quand la pause ?
Quand est-ce qu’on mange ?
Feuille de présence…
Tour de table…

Présentation

[sf≡ir]

Olivier FUXET

Developer backend

@ofuxet

Présentation

[sf≡ir]

Yohann FACON

Developer backend

Présentation

[sf≡ir]

Yves DAUTREMAY

Tech lead Java - DevOps

Présentation

[sf≡ir]

Sébastien FRIESS

Developer backend

@sebastienfriess

Présentation

[sfeir]

Vincent DOLEZ

Developer backend / data

@dolez_v

Présentation

[sf≡ir]

Olivier GERARDIN

Architecte Java

@ogerardin

Présentation

[sf≡ir]

Antoine POIVEY

Full Stack Developer

1 Pourquoi le Go ?

Avant de se lancer

Le but de Go

- Rassembler les bonnes idées provenant d'autres langages,
- Abolir les fonctionnalités débouchant sur un code complexe et peu fiable…

…dans le but d'**écrire du code simple, expressif, robuste et efficace**.

Avant de se lancer

Un langage généraliste

- Comme le **C**, il sera adapté dans pratiquement tous les domaines de programmation.
- Idéal pour le cloud.
- Déjà utilisé dans le graphisme, les applications mobiles, le machine learning, WASM, ...

Avant de se lancer

Déjà adopté par des grands

- Google
- Docker
- Kubernetes
- Dropbox
- Spotify
- Hashicorp/Terraform
- SoundCloud
- etc.

Le langage

- Né en 2009 chez Google (après les processeurs multi-coeurs) et OSS
- Binaire compilé autoporteur (début plugin depuis Go 1.8)
- Orienté objet
- Garbage collector (sub millisecond pour 17 Go de heap)
- Pointeurs 😱
- Goroutines
  - Assimilable à un thread
  - Mais ce n'est **PAS** un thread ⇒ **beaucoup plus léger**
- Channels
  - **Do not communicate by sharing memory; share memory by communicating.**
  - Synchronisation
  - Multiplexage (**select**)

Le langage

- Les Mots-Clés :
  - **Dépendances** : import package
  - **Conditionnelles** : if else switch case fallthrough break default goto select
  - **Itérations** : for range continue
  - **Type** : var func interface struct chan const type map make
  - **Misc** : defer go return panic recover

2 Installation

Installation de l'environnement

Installation de Go

- Téléchargez Go pour votre environnement : https://golang.org/dl/
- Suivez les instructions d'installation : https://golang.org/doc/install

- Vérifiez l'installation dans un terminal : `$ go version`
- Ajoutez la variable d'environnement GOPATH pour pointer sur votre workspace. Par exemple :

```
$ GOPATH=$HOME/go
$ GOPATH=%USERPROFILE%\go
```

Installation de l'environnement

Récupération des sources

- Votre workspace de base contient
  - **bin** : binaires de vos appli
  - **pkg** : vos objets à linker
  - **src** : toutes vos sources
  - Ajouter les bin Go à votre path : **PATH=$PATH:$GOPATH/bin**
- Il vous faut cloner :
  - git clone https://github.com/Sfeir/golang-100
  - dans **$GOPATH/src/github.com/Sfeir/**golang-100
  - vous positionner sur la branche master (git checkout master)

Installation de l'environnement

Les commandes de **go**

- **build** compile packages et dépendances
- **fmt** lance gofmt sur les sources
- **test** lance les tests
- **tool** lance les outils spécifiques (tracer, etc...)
- **get** télécharge et installe les packages and dépendances
- **mod** gestion des dépendances

Installation de l’environnement

Installation de Visual Studio Code

- Téléchargez et installez VSC pour votre environnement : https://code.visualstudio.com/
- Installez le plugin Go :  , puis chercher “go”.

Installation de l’environnement

Installation de Visual Studio Code

- En ouvrant un fichier **.go**, le message “Analysis Tools Missing” apparaîtra : cliquez dessus pour compléter l’installation automatiquement.

- Il est conseillé d’activer la sauvegarde automatique.

3 Let’s Go

Les bases - 01

Hello, 世界

```
package main

import "fmt"

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

Le programme est organisé en packages.
Il doit avoir à minima un package *main*, contenant une fonction *main*.

Go fournit des centaines de packages (fmt, net, http, json, etc.)

La fonction *main* est le point d'entrée du programme.

Les bases - 01

Hello, 世界

Remarque : comme on s'y attend, 世界 signifie "monde", mais ces caractères ne sont pas disponibles en ANSI.

- Par convention, en Go, tous les fichiers sources sont encodés en **UTF8**.
- Les chaînes de caractères sont encodées en **UTF8** par défaut.
- Il est possible de nommer une variable **Δt** par exemple.

Les bases - 01

Compilation, exécution

Placez vous dans le répertoire *go100/01*, puis :

- Compilez le programme : `$ go build main.go`
- Ensuite, exécutez le programme : `$ ./main`
- Ou plus simplement : `$ go run main.go`

Go est un langage compilé. Le compilateur produit un binaire exécutable.

Les bases - 02

Imports

- En Go, il est **interdit** d'importer un package sans l'utiliser.
- Pour importer plusieurs packages, on préférera la notation factorisée.
- Par convention, le nom du package est le même que le dernier élément du chemin d'importation.
  Ex: chemin "math/rand" ⇒ package "rand"

Les bases - 03

Exporter un identifiant

Pour qu'un identifiant (de variable, fonction, type, etc.) soit visible en dehors du package dans lequel il est déclaré, il doit commencer par une majuscule. Il n'y a pas de mot clé "public", "private", ou autre en Go.

⇒ un identifiant est exporté (ie. public) si, et seulement si, il commence par une majuscule.

Les bases - 04

Les fonctions

Une fonction est déclarée à l'aide du mot clé **func** suivi du nom de la fonction. Le corps de la fonction est compris entre { et }. Le caractère '{' doit être obligatoirement sur la même ligne que le mot clé **func**.

```
func maFonction() {
}
```

Les bases - 04

Les fonctions

Une fonction peut prendre zéro ou plusieurs arguments.

```
func maFonction(n int, s string) {
}
```

Notez que le type vient **après** l'identifiant de la variable.

Les bases - 04

Les fonctions

Une fonction peut retourner une valeur.

```
func maFonction(n int, s string) string {
	return s + strconv.Itoa(n)
}
```

Les bases - 04

Les fonctions

En Go, une fonction peut aussi retourner **plusieurs** valeurs.

```
func maFonction(n int, s string) (string, bool) {
	return s + strconv.Itoa(n), n == 0
}
```

Les bases - 04

Les fonctions

Les valeurs de retour peuvent aussi être nommées.

```
func maFonction(n int, s string) (result string, isZero bool) {
	result = s + strconv.Itoa(n)
	isZero = n == 0
	return
}
```

Les bases - 04

Les fonctions

Lorsque deux ou plusieurs paramètres de fonction consécutifs partagent un type, vous pouvez omettre le type de tous sauf le dernier.

```
func maFonction(a bool, b int, c int, d int, e string, f string) { ... }
⇔
func maFonction(a bool, b, c, d int, e, f string) { ... }
```

Les bases - 05

Les variables

On déclare une (ou plusieurs) variable avec l'instruction **var** suivi du nom de la variable, puis du type et de sa valeur initiale.

var nom ***type*** = ***expression***

Le type peut être omis si on donne une valeur initiale, et vice versa.

Les bases - 05

Les variables

Concrètement, voici trois façons de déclarer une même variable :

- **var** nom string = "" déclaration inutilement complète
- **var** nom = "" le type est inféré par la valeur initiale (la chaîne vide)
- **var** nom string la valeur initiale est la valeur-zéro du type

Les bases - 05

Les variables

Il est possible de factoriser la déclaration de plusieurs variables à l'aide de parenthèses :

```
var (
	toto = ""
	titi float32
	tata = true
)
```

Les bases - 05

Les variables

On peut aussi déclarer plusieurs variables sur une même ligne :

```
var a, b, c = "Hello", 42, true
```

Ou si elles sont du même type :

```
var x, y, z float64
```

Les bases - 05

Les variables

A l'intérieur d'une fonction, on peut utiliser la déclaration *courte* :

```
func plop() {
	a := "plop!" // équivalent à : var a = "plop!"
	fmt.Println(a)
}
```

Les bases - 05

Les variables

Attention, gardez à l'esprit que **:=** est une **déclaration+assignation**, tandis que **=** est une **assignation**

Pour cette raison, il est **impossible** d'assigner une valeur à une variable **déjà déclarée** en utilisant **:=**

Les bases - 05

Les variables

A savoir : comme pour la déclaration, on peut assigner plusieurs variables sur une même ligne :

```
x, y = 42, 3.14
```

Très pratique pour faire un swap :

```
x, y = y, x
```

Les bases - 06

Les types de base

- **bool**
- **string**
- **int**, **int8**, **int16**, **int32**, **int64**
- **uint**, **uint8**, **uint16**, **uint32**, **uint64**

int est soit un **int32**, soit un **int64**, dépendamment de la plateforme et du compilateur. Il en va de même pour **uint** qui est soit **uint32** soit **uint64**.

Les bases - 06

Les types de base

- **byte** qui est un alias de int8
- **rune** qui est un alias de int32 et représente un “code point” Unicode
- **uintptr** qui est un entier non signé assez grand pour contenir la valeur d’un pointeur (généralement 32 ou 64 bits).

Les bases - 06

Les types de base

- **float32**, **float64**
- **complex64**, **complex128**

Le type **complex64** est un nombre complexe dont la partie réelle et la partie imaginaire sont des **float32**. Le type **complexe128** utilise des **float64**.

Les bases - 06

Les valeurs zéro

Chaque type a une **valeur zéro**. Une variable déclarée sans valeur initiale (ex: *var* nom *type*) aura pour valeur la **valeur zéro** de son type.

Les valeurs zéro des types de base sont :

- 0 pour les types numériques,
- false pour les types booléen, et
- "" (La chaîne vide) pour les chaînes.

Les bases - 06

Les conversions de type

En Go, toute conversion de type doit être **explicite**.

L'expression T(v) convertit la valeur v au type T.

```
var i int = 42
var f float64 = float64(i)
```

Les bases - 07

Les constantes

- Les constantes sont déclarées comme des variables, mais avec le mot-clé **const**.
- Les constantes peuvent être un caractère, une chaîne, un booléen, ou des valeurs numériques.
- Les constantes ne peuvent pas être déclarées avec la syntaxe := .

Les bases - 07

Les constantes

A savoir : les constantes numériques sont des valeurs de haute précision.

```
const (
	Big   = 1 << 100
	Small = Big >> 99
)
```

Les bases - 08

If

La déclaration **if** en Go s'écrit sans parenthèse, mais les accolades sont obligatoires :

```
if condition {
}
```

Comme pour les fonctions, l'accolade ouvrante doit être placée sur la même ligne que **if**.

Les bases - 08

If

La condition est une expression booléenne.

Nous retrouvons les opérateurs classiques :

==, !=, <, <=, >, >=, &&, ||

Les bases - 08

If

La condition peut être précédée d'une expression suivie d'un **;** (point-virgule)

```
if expression; condition {
}
```

L'expression peut permettre d'initialiser une variable dont la portée sera limité au bloc if/else.

Les bases - 08

If

La clause **else** s'utilise de façon classique et les "**else if**" peuvent être mis à la suite les uns des autres.

```
if condition1 {
} else if condition2 {
} else {
}
```

Les bases - 08 - Exercice

Les fonctions et les if

Éditez le fichier 08-exercice/main.go

Écrire une fonction calculant l'**inverse** d'un nombre décimal (1/x).

Note : pensez au cas où le nombre est nul. Il conviendrait de retourner un booléen à *false* par exemple.

Les bases - 08 - Correction

Les fonctions et les if

Voir le fichier 08-exercice/correction/main.go

Observez la façon dont le cas d’erreur est pris en compte directement dans l’instruction **if** où la fonction est appelée.

Les bases - 09

Les boucles

Go a une seule structure de boucle, la boucle **for**.

Elle a trois composantes séparées par des points-virgules:

- La déclaration d'initialisation : exécutée avant la première itération
- L'expression de condition : évaluée avant chaque itération
- La déclaration d'aboutissement : exécutée à la fin de chaque itération

Les bases - 09

Les boucles

Comme pour l'instruction **if**, il n'y a pas de parenthèses entourant les trois composantes de la déclaration **for** et les accolades { } sont exigées.

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

Les bases - 09

Les boucles

La déclaration d'initialisation sera souvent une déclaration de variable courte et les variables ainsi déclarées ne sont visibles que dans le cadre de la déclaration **for**.

L'itération de la boucle sera arrêtée une fois que la condition booléenne sera évaluée à *false*.

Les bases - 09

Les boucles

La déclaration d'initialisation et d'aboutissement sont facultatifs.

```
for ; n < 1000; {
    n += n
}
```

qui peut s'écrire sans **';'** :

```
for n < 1000 {
    n += n
}
```

Les bases - 09

Les boucles

La condition peut elle aussi être omise.

On obtient ainsi une boucle infinie :

```
for {
}
```

Les bases - 09 - Exercice

Les boucles

Écrire une fonction implémentant l'algorithme de la conjecture de Syracuse qui prend en paramètre :

- un entier de départ
- un nombre max d'itérations à effectuer

Et qui devra retourner :

- un booléen indiquant si le nombre 1 a été atteint
- le nombre d'itérations effectuées.

Les bases - 10

Switch

Le switch fait exactement ce qu'on attend de lui. Comme pour le if ou le for, une expression peut être ajoutée avant la condition.

```
switch initialisation; expression {
case a:
case b:
default:
}
```

Les bases - 10

Switch

En Go, un **case** se termine automatiquement (l'instruction **break** est inutile).

Si, au contraire, nous voulons que la **case** suivante soit exécutée, il faut ajouter le mot clé **fallthrough**.

Les bases - 10

Switch

Si l’on omet la condition, le switch peut permettre d'écrire élégamment de longues chaînes if-then-else.

```
switch { //équivalent à switch true
case i < 0:
case i == 0:
default:
}
```

Les bases - 11

Defer

Une déclaration **defer** reporte l'exécution d'une fonction jusqu'à ce que la fonction environnante retourne.

Les arguments de l'appel différé sont évalués immédiatement, mais l'appel de fonction n'est pas exécuté jusqu'à ce que la fonction environnante retourne.

Les bases - 11

Defer

```
func main() {
	defer fmt.Print("world\n")
	defer fmt.Print(", ")
	fmt.Print("Hello")
}
```

Les appels de fonction différés sont poussés sur une pile (FILO) et sont effectués même en cas de panique.

4 Let’s Go further

Pour aller plus loin - 12

Déclaration de type

Nous pouvons déclarer un nouveau type à l'aide du mot clé **type**.

Par exemple, il est possible de déclarer un type dont le type sous-jacent est un nombre :

```
type Distance int64
```

C'est exactement ce qui est fait dans le package *time* de Go :

```
type Duration int64
```

Pour aller plus loin - 12

Déclaration de type

Il est possible d'effectuer les même opérations que sur le type sous-jacent, mais seulement entre variables du même type. C'est très utile pour éviter les erreurs d'homogénéité dans une formule.

```
var d Distance = 300
var t Duration = 60
v := d / t // erreur de compilation
v := float64(d) / float64(t)  // ok
```

Pour aller plus loin - 13

Les structures

Une structure est une collection de champs. Déclarer une structure, c'est déclarer un nouveau type. Il est donc logique de déclarer une structure à l'aide du mot clé **type** et du type sous-jacent **struct** :

```
type Vector struct {
    X int
    Y int
}
```

Pour aller plus loin - 13

Les structures

Une structure peut être instanciée littéralement, offrant la possibilité d'initialiser la valeur des champs :

```
v1 := Vector{1, 2}  // X:1 et Y:2
v2 := Vector{X: 1}  // X:1, et Y:0 est implicite
v3 := Vector{}      // X:0 et Y:0
```

Pour aller plus loin - 13

Les structures

Les champs de la structure sont accessibles à l'aide d'un point :

```
func main() {
	v := Vector{1, 2}
	v.X = 4
	fmt.Println(v.X)
}
```

Pour aller plus loin - 14

Les pointeurs

Go comporte des pointeurs. Un pointeur contient l'adresse mémoire d'une valeur.

Le type ***T** est un **pointeur** vers une valeur de type **T**.

```
var p *int
```

Ici, p est un pointeur vers un entier.

Pour aller plus loin - 14

Les pointeurs

La valeur zéro d'un pointeur est **nil**.

L'opérateur **&** génère un pointeur vers son opérande.

```
var p *int  // p vaut nil
i := 42
p = &i  // p est un pointeur vers i
```

Pour aller plus loin - 14

Les pointeurs

L'opérateur * dénote la valeur sous-jacente du pointeur.

```
i := 3
p := &i
fmt.Println(*p)   // lire i (=3) par le pointeur p
*p = 21           // définir i par le pointeur p
```

Pour aller plus loin - 14

Les pointeurs vers les structures

Les champs struct peuvent être accessibles via un pointeur de struct.

L'indirection via le pointeur est transparente.

```
func main() {
	v := Vector{1, 2}
	p := &v
	p.X = 7
	fmt.Println(v)  // { X:7, Y:2 }
}
```

Si vous appréciez la formation, Envoyez un Tweet !

#sfeirschool #golang
@Sfeirlille @sfeir
@sebastienfriess

Si vous appréciez la formation, Envoyez un Tweet !

#sfeirschool #golang
@sfeirstrbg @sfeir
@sebastienfriess

Si vous appréciez la formation, Envoyez un Tweet !

#sfeirschool #golang
@SFEIRLux @sfeir
@ogerardin

Pour aller plus loin - 14 - exercice

Les pointeurs et les structures

Répondez aux questions en remplaçant **REPLACEME** par **true** ou **false**.

Pour aller plus loin - 15

Les tableaux

Le type **[n]T** est un tableau de **n** valeurs de type **T**.

L'expression :

```
var a [10]int
```

déclare une variable **a** comme un tableau de dix entiers.

Pour aller plus loin - 15

Les tableaux

La longueur d'un tableau fait partie de son type.

⇒ les tableaux ne peuvent pas être redimensionnés.

Pour aller plus loin - 16

Les slices

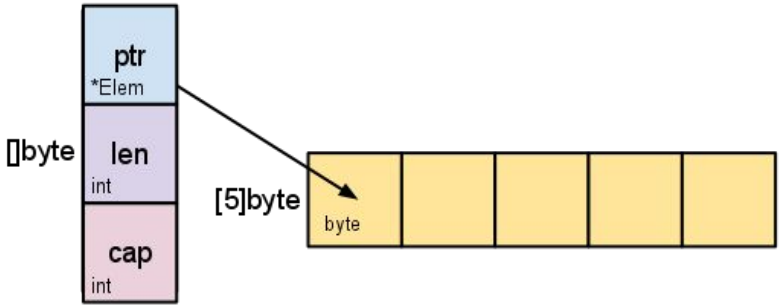
Un **slice** pointe vers un tableau de valeurs et possède une longueur.

[]T est un slice avec des éléments de type **T**.

len(s) retourne la longueur du slice **s**.

cap(s) retourne la capacité du slice **s**.

La valeur zéro d'un slice est **nil**.



Pour aller plus loin - 16

Les slices

Les slices peuvent être créés avec la fonction **make**. Cela alloue un tableau vide et retourne un slice qui référence ce tableau :

```
a := make([]int, 5)  // len(a)=5
```

Pour aller plus loin - 16

Les slices

Les slices peuvent être redécoupés, créant ainsi un nouveau slice qui pointe vers le même tableau.

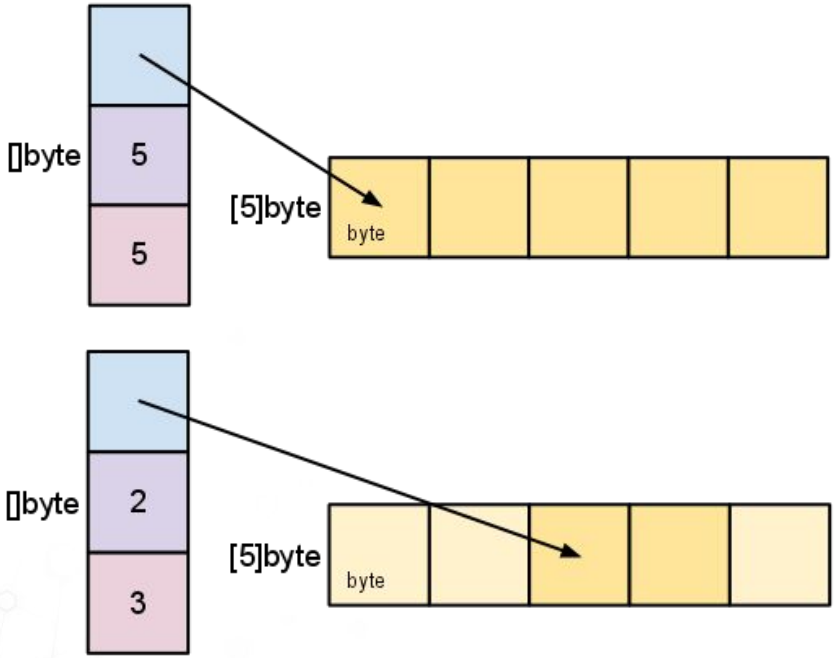
L’expression :

```
s[low:high]
```

récupère les éléments du slice de **low** à **high-1** inclus (ou **high** exclus).

Pour aller plus loin - 16

Les slices




```
var a = make([]byte, 5)
len(a) == 5
cap(a) == 5
```

```
var b = a[2:4]
len(b) == 2
cap(b) == 3
```

Pour aller plus loin - 16

Les slices

Pour spécifier une capacité, passer un troisième argument à make :

```
b := make([]int, 0, 5) // len(b)=0, cap(b)=5
b = b[:cap(b)]         // len(b)=5, cap(b)=5
b = b[1:]              // len(b)=4, cap(b)=4
```

Pour aller plus loin - 16

Les slices

Il est courant d'ajouter de nouveaux éléments à un slice, et ainsi Go offre une fonction intégrée **append**.

func append(s []T, vs ...T) []T

Le premier paramètre s de append est un slice de type T, et les suivants sont des valeurs T à ajouter au slice.

Pour aller plus loin - 17

For ... range

La forme **range** de la boucle **for** permet d'itérer sur tous les éléments d'un slice, d'un tableau, ou d'une map.

Deux valeurs sont renvoyées pour chaque itération :

- l'indice
- une **copie** de l'élément à cet indice.

Pour aller plus loin - 17

For ... range

```
for i, v := range s {
}
```

Est équivalent à :

```
for i := 0; i < len(s); i++ {
	v := s[i] // s[i] est ici modifiable
}
```

Pour aller plus loin - 17

For … range

On peut omettre l'indice en le remplaçant par "_" :

```
for _, v := range s { }
```

On peut aussi omettre la valeur.

```
for i := range s { }
```

Pour aller plus loin - 17 - exercice

Les slices et le range

Écrire les fonctions **indexOf** et **split**.

Pour aller plus loin - 18

Maps

Une map est une liste de clés-valeurs. Une map se déclare :

map[typeClé]typeValeur

Les maps doivent être créées avec make. Leur valeur-zéro est nil.

m = make(**map**[string]Vector)

Pour aller plus loin - 18

Maps

Comme les structs, une map peut être instanciée littéralement. Il faut cependant indiquer la valeur de chaque clé :

```
var m = map[string]Vector{
    "Bell Labs": Vector{40.68433, -74.39967},
    "Google": Vector{37.42202, -122.08408},
}
```

Pour aller plus loin - 18

Maps

- Affectation : m[key] = elem
- Suppression : delete(m, key)
- Récupération :
  - elem = m[key] *//elem prend la valeur-zéro de son type si la clé n'est pas présente.*
  - elem, ok = m[key] *//ok est true si la clé est présente, sinon false.*

Pour aller plus loin - 19

Un peu plus sur les fonctions

En Go, les fonctions sont aussi des valeurs. Elles peuvent être passés comme toute autre valeur.

Les valeurs de fonction peuvent être utilisées comme arguments de fonctions ou valeurs de retour. On dit que les fonctions sont ***first class citizen*** en Go

Pour aller plus loin - 19

Closures

Les fonctions de Go peuvent être des **closures**. Une **closure** (fermeture) est une valeur de la fonction qui fait référence à des variables à partir de l'extérieur de son corps.

La fonction peut accéder et assigner les variables référencées, dans ce sens, la fonction est «liée» aux variables.

Pour aller plus loin - 19 - exercice

Closures

Implémentez la fonction **multiplicateurPar** pour qu'elle retourne une fonction qui multiplie son argument par un nombre donné.

```
//fonction qui retourne une fonction qui retourne un int.
func multiplicateurPar(x int) func(int) int {
}
```

Pour aller plus loin - 20

Déclarations multiples

Il est possible en Go de déclarer plusieurs variables sur une même ligne.

```
a, b, c := 1, "toto", 8
```

Pour aller plus loin - 20

Assignations multiples

De même, il est possible d'assigner plusieurs variables sur une même ligne.

C'est très pratique pour effectuer un *swap* :

```
x, y = y, x
```

5 Orienté objet

Orienté Objet - 21

Les méthodes

Go ne dispose pas de **classes**.

Cependant, on peut définir des méthodes sur les types.

Orienté Objet - 21

Les méthodes

Une **méthode** est une **fonction** avec un ***récepteur***, indiqué entre le mot clé **func** et le **nom** de la méthode.

```
func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

La méthode **Abs** possède un récepteur de type **Vertex** nommé **v**.

Orienté Objet - 21

Les méthodes

Une **méthode** est donc juste une **fonction** avec un **récepteur**.

Voici Abs écrit comme une fonction régulière sans changement de fonctionnalité :

```
func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Orienté Objet - 21

Les méthodes

La différence se fait dans la façon d'appeler la fonction/méthode.

Méthode :

```
v := Vertex{1, 3}
a := v.Abs()
```

Fonction :

```
v := Vertex{1, 3}
a := Abs(v)
```

Orienté Objet - 21

Les méthodes

On peut aussi déclarer une méthode sur des types non-struct.

```
type MyFloat float64

func (f MyFloat) IsPositive() bool {
	return f >= 0
}
```

Orienté Objet - 21

Les méthodes

En revanche, le **type du récepteur** doit impérativement être déclaré dans le même **package** que la méthode.

Il est donc impossible d'écrire :

```
func (f float64) IsPositive() bool {
	return f >= 0
}
```

puisque float64 est un type défini dans un autre package.

Orienté Objet - 21

Les méthodes, récepteur de pointeur

On peut déclarer des méthodes avec des récepteurs de **pointeur**.

Cela permet à la méthode de modifier la valeur vers laquelle le récepteur pointe :

```
func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}
```

Orienté Objet - 22

Les méthodes, récepteur de pointeur

Un autre avantage des récepteurs de pointeurs est de pouvoir profiter de la transparence de l'indirection de Go :

v.Scale(5) compilera si **v** est de type ***Vertex** ou **Vertex**.

alors que :

Si on a **func** ScaleFunc(v *Vertex, f float64)

ScaleFunc(v, 10) ne compilera que si **v** est de type ***Vertex**.

Orienté Objet - 22

Choix d'un récepteur de valeur ou pointeur

Il y a deux raisons d'utiliser un récepteur de pointeur :

- le cas ou la méthode doit modifier la valeur du récepteur pointé.
- éviter la copie de la valeur sur chaque appel de méthode.

⇒ à l’usage on utilisera le plus souvent des récepteurs de pointeur.

Orienté Objet - 23

Le récepteur nil

En Go, il est possible d'appeler une méthode sur un pointeur **nil** sans générer d'exception :

```
var p *T = nil
p.SomeMethod()
```

Orienté Objet - 23

Le récepteur nil

Il faut malgré tout que la méthode soit prévue pour fonctionner avec un récepteur ayant la valeur **nil** :

```
func (pf *MyFloat) IsPositive() bool {
        return pf != nil && *pf >= 0
}
var p *MyFloat = nil
ok := p.IsPositive() // pas d'erreur. ok = false
```

Orienté Objet - 23 - exercice

Les méthodes et le récepteur nil

Créez un type **person** qui contiendra le prénom et le nom d'une personne.

Ajoutez une méthode **fullname()** sur le type ***person** qui retourne la concaténation du prénom et du nom.

La méthode devra gérer le cas où le récepteur est nil.

Orienté Objet - 24

Les interfaces

En Go, il n’y a pas de notion d’héritage entre les **struct**, mais la notion d’**interface** est bien présente.

Orienté Objet - 24

Les interfaces

Comme on s’y attend, un type d'interface est définie comme un ensemble de signatures de méthode.

```
type Oiseau interface {
	Vole(direction Vector)
	Mange() (int, bool)
}
```

Orienté Objet - 25

Les interfaces

Particularité de Go : les **interfaces** sont implémentées **implicitement**.

Un type implémente “automatiquement” une interface en mettant en œuvre ses méthodes.

⇒ Il n'y a aucune déclaration explicite d'intention, pas de mot-clé «*implements*».

Orienté Objet - 25

Les interfaces

Si on a :

```
type I interface {
    M()
}

type T struct { }
```

Alors il suffit de déclarer la méthode :

```
func (t T) M() {
    /*...*/
}
```

pour que le type **T** implémente **I**.

Orienté Objet - 25

Les interfaces

Si on a :

```
type Aigle struct {}
func (a Aigle) Mange() { /*... */ }
func (a Aigle) Vole() { /*... */ }
```

Et :

```
type Oiseau interface {
    Mange()
    Vole()
}
type Volant interface { Vole() }
```

⇒ *Quelle(s) interface(s) Aigle implémente-t-il ?*

Orienté Objet - 25

Les interfaces

Si on a :

```
type Aigle struct {}
func (a Aigle) Mange() { /*... */ }
func (a Aigle) Vole() { /*... */ }
```

Et :

```
type Oiseau interface {
        Mange()
        Vole()
}
type Volant interface { Vole() }
```

⇒ **Aigle** implémente à la fois l'interface **Oiseau** et l'interface **Volant**.

Orienté Objet - 25

Les interfaces

En revanche, si on a :

```
type A380 struct {}
func (a A380) Vole() { /*...*/ }
```

Et :

```
type Oiseau interface {
        Mange()
        Vole()
}
type Volant interface { Vole() }
```

⇒ *Quelle(s) interface(s) A380 implémente-t-il ?*

Orienté Objet - 25

Les interfaces

En revanche, si on a :

```
type A380 struct {}
func (a A380) Vole() { /*...*/ }
```

Et :

```
type Oiseau interface {
        Mange()
        Vole()
}
type Volant interface { Vole() }
```

⇒ **A380** implémente seulement l'interface **Volant**.

Orienté Objet - 26

Les interfaces

Attention,

il suffit de déclarer la méthode :

```
func (t T) M() {
      fmt.Println(t.S)
}
```

pour que le type **T** implémente **I**.

mais si on déclare :

```
func (t *T) M() {
      fmt.Println(t.S)
}
```

c'est le type ***T** qui implémente **I**.

Orienté Objet - 26

Les interfaces

⇒ En général, toutes les méthodes sur un type donné **T** doivent avoir soit un récepteur de valeur (T) soit un récepteur de pointeur (*T), mais pas un mélange des deux.

Sinon, **T** implémenterait des interfaces tandis que ***T** en implémenterait d'autres, ce qui deviendrait pénible à utiliser.

Orienté Objet - 27

L'interface vide

L'interface vide ne définit, comme son nom l'indique, aucune méthode :

```
interface{}
```

Tous les types implémentent l'interface vide puisqu'un type a toujours au moins zéro méthode. On utilise donc l'interface vide pour gérer des valeurs de type inconnu.

Ex: *fmt.Print* prend des arguments de type **interface**{}

Orienté Objet - 28

Assertion de type

Une assertion de type donne accès à la valeur concrète (sous-jacente) d'une valeur d'interface.

```
t := i.(T)
```

Cette instruction affirme que la valeur d'interface **i** contient le type concret **T** et assigne la valeur sous-jacente à la variable **t**.

Orienté Objet - 28

Assertion de type

Si **i** ne détient pas **T**, la déclaration va déclencher une **panique**.

Mais l'assertion de type peut aussi s'effectuer sous la forme :

```
t, ok := i.(T)
```

Si **i** ne détient pas **T**, aucune panique ne sera déclenchée, mais **ok** aura la valeur ***false***.

Orienté Objet - 29

Switch de type

Un switch de type permet de gérer les différents types concrets qu'une interface pourrait avoir :

```
switch v := i.(type) {
case T:    // ici v est de type T
case S:   // ici v est de type S
default: // ici v a le même type que i
}
```

Orienté Objet - 29 - exercice

Les interfaces

1. Créer un type **FmtLogger** qui implémente l'interface **Logger**.
2. Finir l'implémentation de la fonction **recoverPanic** en utilisant le **logger** pour écrire ce que contient la panique **r**.
3. Instancier le **FmtLogger** et appeler la fonction **recoverPanic** dans la fonction **main**.

6 Programmation concurrente

Concurrence - 30

Goroutine

Une **goroutine** est un **processus léger** géré par le « Go runtime ».

```
go f(x, y, z)
```

démarre une nouvelle **goroutine** exécutant

```
f(x, y, z)
```

L'évaluation de **f**, **x**, **y** et **z** est effectuée dans la goroutine actuelle et l'exécution de **f** est effectuée dans la nouvelle goroutine.

Concurrence - 31

Channels (canaux)

Les **channels** sont des conduits typés à travers lesquels vous pouvez envoyer et recevoir des valeurs avec l'opérateur de canal, **<-**

```
ch <- v    // Envoyer v sur le canal ch.
v := <- ch  // Recevoir de ch, et attribuer la valeur à v.
```

(Le flux de données vas dans le sens de la flèche.)

Concurrence - 31

Channels (canaux)

Les **channels** doivent être créés avant l'utilisation :

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

Par défaut, l'envoi et la réception bloquent jusqu'à ce que l'autre côté soit prêt. Cela permet de synchroniser les goroutines sans verrous explicites ou variables de condition.

Concurrence - 32

Channels avec buffer

Les **channels** peuvent avoir un **buffer**. Il faut indiquer la taille du buffer en second argument de make pour initialiser un channel avec buffer :

```
ch := make(chan int, 100)
```

L'envoi à un channel avec un buffer **bloque** uniquement **lorsque le buffer est plein**. La réception **bloque lorsque le buffer est vide**.

Concurrence - 33

Fermeture de channels

Un expéditeur peut fermer un canal pour indiquer que plus aucune valeur ne sera envoyée :

```
close(ch)
```

Les récepteurs peuvent vérifier si un canal a été fermé :

```
v, ok := <-ch
```

ok est *false* si le canal est fermé et qu'il n'y a plus de valeur à recevoir.

Concurrence - 33

Fermeture de channels

Attention :

- **Seul l'expéditeur doit fermer un canal**, jamais le récepteur. L'envoi sur un canal fermé provoque une panique.
- Les canaux ne sont pas comme les fichiers, vous n'avez généralement **pas besoin de les fermer**. La fermeture n'est nécessaire que lorsque le récepteur doit être informé qu'il n'y a plus de valeurs à venir, comme mettre fin à la boucle d'un range.

Concurrence - 34

For range sur les channels

La boucle **for i := range c** reçoit les valeurs du canal jusqu'à ce qu'il soit fermé.

```
for i := range c {
	fmt.Println(i)
}
```

Concurrence - 34 - exercice

Les goroutines

Modifiez le programme pour que les requêtes HTTP soient faites simultanément.

Concurrence - 35

Select

La déclaration **select** permet à une goroutine d'attendre sur plusieurs opérations de communication simultanément.

```
select {
case c <- x: // attend de pouvoir envoyer x sur 'c'
   x, y = y, x+y
case <- quit: // attend de recevoir depuis 'quit'
   fmt.Println("quit")
}
```

Concurrence - 35

Select

Un **select** bloque jusqu'à ce que l'un de ses cas puisse s'exécuter, puis il l'exécute. Il en choisit un au hasard si plusieurs sont prêts.

Concurrence - 36

Select

Le cas **default** dans un **select** est exécuté si aucun autre cas n'est prêt et permet donc d'éviter le blocage du select.

```
select {
case i := <-c:
   // utiliser i
default:
   // exécuté si rien ne peut être lu de 'c'
}
```

Concurrence - 36 - exercice

Les goroutines et select

Modifiez le programme pour que les erreurs soient envoyées sur un canal spécifique (chan error) et récupérées via un select dans la fonction main.

7 Et maintenant ?

Et maintenant ?

Pour aller plus loin

- golang.org (documentation, wiki, référence des packages, etc.)
- blog frenchgo.fr (news & infos sur le Go)
- livre blanc Comprendre Go de SFEIR
- livre The Go Programming Language
- chaîne youtube : The Go Programming Language
- chaîne youtube et blog : Just for func

Merci !