



Cours Go

Jean-Baptiste Yunès

yunes.informatique.univ-paris-diderot.fr

Jean-Baptiste.Yunes@u-pariscite.fr

Informations

- Ce cours n'est pas un cours complet/exhaustif de Go
- Certaines constructions seront à acquérir par vous-même
- Consultez le site <https://go.dev/>
 - Installez la version la plus récente (aujourd'hui 1.27.1)
- Installez un IDE
 - Visual Studio Code est gratuit et open source
 - Installez des extensions pour Go (au moins Go)
- Ce support contient des éléments extraits de la documentation Go

Anatomie d'un projet Go

Généralités

- Module
 - Ensemble de packages
 - définit par un fichier `go.mod`
- Package
 - Ensemble de codes sources
- Code source
 - directive `package` en tête

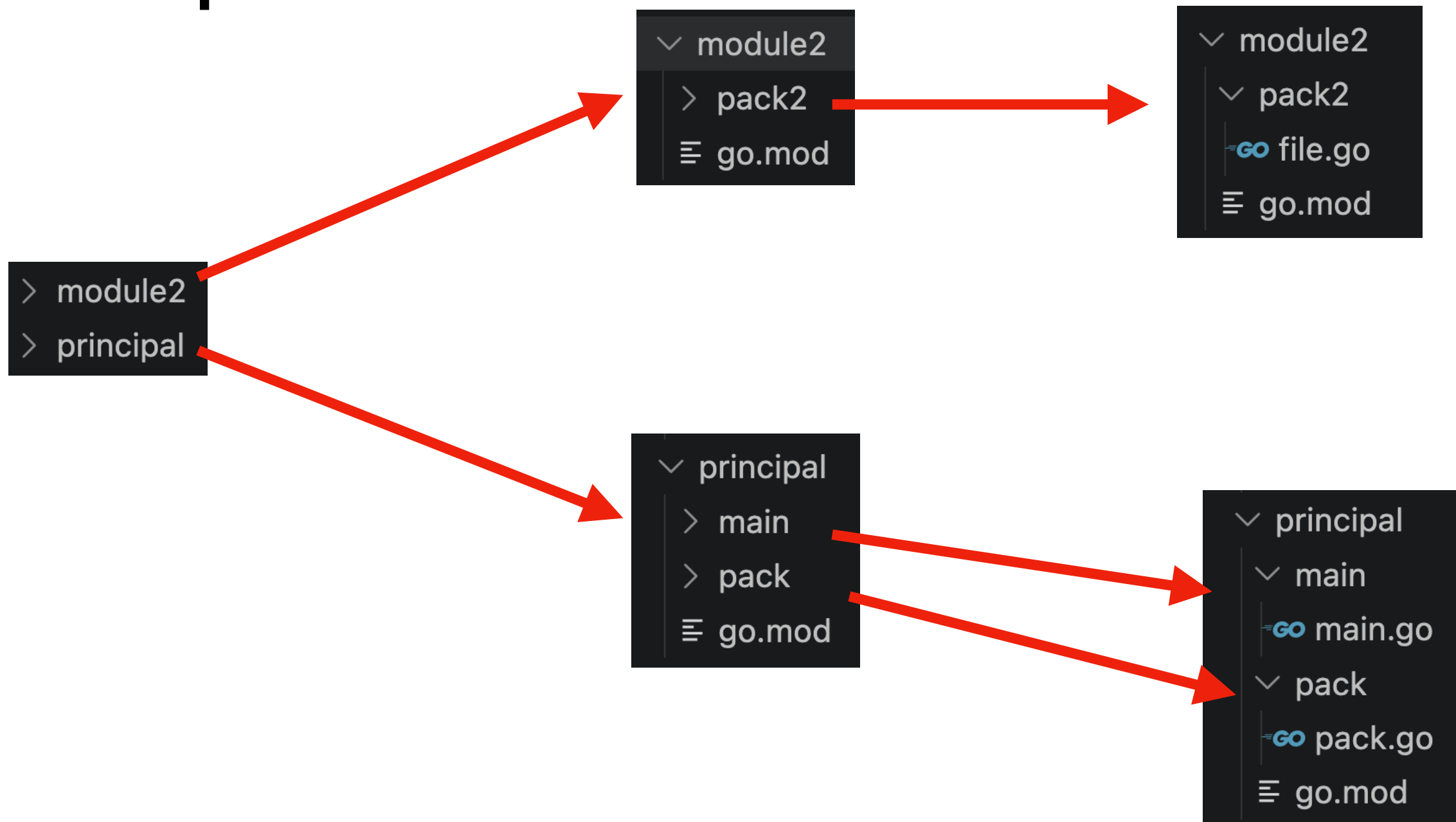
Anatomie d'un projet Go

Généralités

- Un module $\Leftrightarrow$ un répertoire + `go.mod`
- Un package $\Leftrightarrow$ un répertoire à l'intérieur d'un module
- Un ou plusieurs fichiers sources dans un package avec directive `package`
- Il vaut mieux que les modules et packages soient dans des répertoires qui portent leur nom

Anatomie d'un projet Go

Exemple



Anatomie d'un projet Go

go.mod

- Peut être créé à la main ou
- par la commande : `go mod init nom_du_module`
- Un exemple de fichier `go.mod` :

```
module module2  
  
go 1.27.0
```

Anatomie d'un projet Go

Un code source

- Un exemple de fichier .go :

```
package main

import (
    "fmt"
    "main/pack"
    "module2/pack2"
)

func main() {
    fmt.Println("cuicui")
    pack.F()
    pack2.F()
}
```

Anatomie d'un projet Go

go.work

- Il est parfois nécessaire d'utiliser un fichier go.work
- On peut le créer à la main ou
- utiliser la commande : `go work init`
- Un exemple :
 - use permet de faire en sorte que la commande go retrouve ses petits

```
go 1.27.0  
  
use principal  
  
use module2
```

Anatomie d'un projet Go

main...

- Un programme Go démarre par l'exécution de la fonction `main` définie dans le package `main`

Exercice

- Créer quelque part un exemple minimal :
 - un fichier `test.go` contenant la fonction `main` du package `main` qui à l'exécution affichera Bonjour
 - Une exécution est pbtenue par la commande
`go run fichier.go`
- Attention, il ne s'agit pas encore de faire un projet modulaire mais de tester si l'installation fonctionne
 - Dans la suite on veillera à toujours utiliser les modules...

La commande Go

- La commande Go est versatile, on l'utilise avec des sous-commandes
 - `go run`
 - `go build`
 - `go mod`
 - `go work`
 - `go clean`
 - `go test`
- Pour l'aide : `go help sous-commande`

Le langage Go

Quelques éléments syntaxiques

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    var val1, val2 float32
    val1, val2 = initialValue()
    affiche(val1, val2)
}

func initialValue() (float32, float32) {
    v1 := rand.Float32()
    v2 := rand.Float32()
    return v1, v2
}

func affiche(v1, v2 float32) {
    fmt.Println(v1, " -- ", v2)
}
```

Le langage Go

Quelques éléments syntaxiques

- Un code **doit** être placé dans un package
- Si on utilise d'autres packages il faut les importer
 - On ne peut importer des packages que l'on utilise pas
 - Erreur de compilation
- Ensuite on définit des types, des fonctions, des variables globales

Le langage Go

Règle de visibilité

- Le découpage en package permet de rendre visible/cacher des éléments
- Un élément d'un package n'est **visible** depuis un autre package que si le nom de cet élément commence par une **majuscule**
- On ne peut utiliser un élément défini dans un autre package que si l'on importe le package correspondant (et si le nom de l'élément commence par une majuscule)
- Attention cette règle ne fonctionne pas pour le «package» `builtin` qui contient des éléments de base

Le langage Go

builtin

- Les mots prédéclarés de Go
- Ils ne sont pas réservés !!!!
 - On peut les redéfinir

```
func append(slice []Type, elems ...Type) []Type
func cap(v Type) int
func clear[T ~[]Type | ~map[Type]Type] T
func close(c chan<- Type)
func complex(r, i FloatType) ComplexType
func copy(dst, src []Type) int
func delete(m map[Type]Type1, key Type)
func imag(c ComplexType) FloatType
func len(v Type) int
func make(t Type, size ...IntegerType) Type
func max[T cmp.Ordered](x T, y T) T
func min[T cmp.Ordered](x T, y T) T
func new(TypeOrExpr) *Type
func panic(v any)
func print(args ...Type)
func println(args ...Type)
func real(c ComplexType) FloatType
func recover() any
```

```
type ComplexType
type FloatType
type IntegerType
type Type
type Type1
type TypeOrExpr
type any
type bool
type byte
type comparable
type complex64
type complex128
type error
type float32
type float64
```

```
true
false
```

```
iota
```

```
type int
type int8
type int16
type int32
type int64
type int64
type rune
type string
type uint
type uint8
type uint16
type uint32
type uint64
type uintptr
```

Le langage Go

Les mots-clés de Go

- Ces mots-clés ne peuvent être employés pour autre chose que ce pourquoi ils sont définis

break	case	chan	const	continue	default
defer	else	fallthrough	for	func	go
goto	if	import	interface	map	package
range	return	select	struct	switch	type
var					

Le langage Go

Les identificateurs

- Un identificateur (de fonction, variable, type, etc) peut contenir n'importe quel caractère alphabétique et numérique Unicode ainsi que le caractère _
 - ඒඒමෆෆ23 est valide (Tamoul+Runique/Chiffres) mais déconseillé
 - si la première lettre est une majuscule (règle de visibilité)
- Convention camel case : ilFaitBeauEtChaud
 - les acronymes sont en majuscules : sendJSON

Le langage Go

Les opérateurs

+	&	+=	&=	&&	==	!=	(	)
-		-=	=		<	<=	[	]
*	^	*=	^=	<-	>	>=	{	}
/	<<	/=	<<=	++	=	::=	,	;
%	>>	%=	>>=	--	!	...	.	:
	&^		&^=		~			

Le langage Go

Les littéraux

- Les littéraux:
 - entier : 666
 - flottant : 3.14
 - complexe : $1+3i$
 - string : "Bonjour"
 - rune : 'A'

Le langage Go

Le formatage

- Le formatage est libre mais :
 - Go rajoute implicitement un ; :
 - après un identificateur ou un littéral en fin de ligne
 - après break, continue, fallthrough, return, ++, --, },),] en fin de ligne
 - après }
- Bizarrerie ???

```
var i = 3
if i <
3 {

}
```

Le langage Go

La portée des identificateurs

- Les portées définies par Go sont :
 - la portée globale
 - première lettre **majuscule** => **visible** de l'extérieur
 - la portée locale
- Go autorise le masquage

Le langage Go

Inférence de type, initialisation

- Go utilise l'inférence de type à l'initialisation
- Une variable non initialisée reçoit une valeur par défaut
 - `0` pour les numériques
 - `false` pour les booléens
 - `nil` pour les pointeurs

Le langage Go

Variable globale

- Une variable globale est nécessairement définie avec le mot-clé `var`
- L'opérateur d'initialisation est `=`
- L'initialisation peut-être faite sous la forme de tuple
- L'ordre d'initialisation est de la gauche vers la droite

```
var v1=2, v2=333*v1, v3=(v1==v2)
```

```
var x, y = 12, 24
```

```
var x1, x2, x3 = x1+x2+1, 0, 1
```

Le langage Go

Variable globale

- Pour typer une variable, le type doit suivre l'identificateur

```
var v int = 32
```

Le langage Go

Variable locale

- On peut utiliser la même syntaxe (var) pour une variable locale
- Mais il existe une forme courte avec := (préférée) mais qui ne permet pas de typer la variable et oblige son initialisation (qui elle est typée)
- Son typage est obtenu par inférence

```
v := 12  
z := float64(13)
```

Le langage Go

Variable locale

- L'initialisation tuple fonctionne aussi

```
v, w := 12, 34
```

- Elle sert aussi d'affectation s'il y a au moins une variable nouvellement définie

```
v := 12  
w, v := v, 34
```

- Attention, il n'est pas possible de faire référence aux variables en cours d'initialisation

```
x1, x2 := x2, 0 // interdit
```

Le langage Go

Affectation

- L'opérateur d'affectation (=) est tuple

```
v, w = 12, 34
```

- Les valeurs en partie droite sont calculées avant l'affectation

```
v, w := 1, 2  
v, w = v+1, w+v+1
```

Le langage Go

Le for

- La boucle for a 4 formes
 - `for definitions; test de continuation; instruction { ... }`
 - `for test de continuation { ... }`
 - `for { ... }`
- Le for d'itération avec range (voir plus loin)
- Attention, on ne doit pas parenthéser
 - `for i:=0; i<10; i++ { ... }`

Le langage Go

Le for

- Les définitions sont nécessairement des formes courtes (`:=`)
- Les variables définies sont dans la portée de la boucle

Le langage Go

Le if

- Deux formes de base
 - `if definitions; test { ... }`
 - `if test { ... }`
- puis les variantes avec `else`
- La portée des variables définies est celle du `if` et du `else...`

```
if i:=666; j<i {  
} else {  
}
```

Le langage Go

label, break, continue

- On peut étiqueter n'importe quel bloc d'instructions

```
unBloc : { ... }
```

- On peut sortir d'un for étiqueté ou non

```
unBloc : for ... { ... break unBloc; ... }
```

- On peut forcer la continuation d'un for étiqueté ou non

```
unBloc : for ... { ... continue unBloc; ... }
```

Le langage Go

label, goto

- On peut forcer à sauter vers un bloc
 - mais le saut ne doit pas comporter de définition de variable !
 - et on ne peut sauter à l'intérieur d'un bloc...

```
unBloc : { ... }  
...  
{ ... goto unBloc; ... }
```

Exercice

- La suite de Syracuse du nombre N est définie ainsi :
 - $S_0 = N$
 - $S_{n+1} = S_n/2$, si n est pair
 - $S_{n+1} = 3S_n+1$, si n est impair
- On définit
 - le vol comme la plus petite sous-suite qui termine par le nombre 1
 - la durée de vol comme la longueur de la sous-suite
 - l'altitude maximale comme le plus grand élément de la sous-suite

Exercice

- Écrire un programme qui pour une valeur de N donnée affiche:
 - le vol
 - la durée
 - l'altitude maximale
- Modifier le programme de sorte que ces données soient affichées pour tout $1 \leq N \leq 100$

Le langage Go

Le switch

- Il a deux formes possibles

```
switch définitions; sélecteur {  
    case expression: ...  
    ...  
}
```

```
switch { // blank switch  
    case expression: ...  
    ...  
}
```

- default peut apparaitre n'importe où
- les cas sont exclusifs, sinon il faut utiliser fallthrough

Le langage Go

Le switch

- Le switch est dynamique
 - La valeur des cas est calculée à l'exécution

```
switch v:=10; i {  
    case 2*v+1 : ...  
    ...  
}
```

- On peut y appeler des fonctions... (voir suite)
- Le blank switch...

Le langage Go

Les fonctions

- Une fonction est introduite avec le mot-clé `func`
 - elle peut avoir autant d'arguments que l'on souhaite
 - elle peut renvoyer autant d'arguments que l'on souhaite
- Le dernier argument peut être variadique (voir slices plus loin)

```
func f(i int) (int, int) {  
    return i+1, i+2  
}  
  
func g() { ... }
```

Le langage Go

Les fonctions

- On peut nommer les arguments de sortie

```
func f(i int) (v1 int, v2 int) {  
    v1, v2 = i+1, i+2  
    return  
}
```

- Il n'y a pas de surcharge en Go...
- Les arguments sont passés par copie
- On peut définir des arguments non nommés (`_`)

Le langage Go

Les fonctions

- On peut définir des lambdas et faire de la programmation fonctionnelle

```
func f(i int) (v1 int, v2 int) {  
    incroyable := 1  
    g := func(n int) { return n+incr }  
    v1, v2 = g(i), g(g(i))  
    return  
}
```

Exercice

- Cette fois on crée un module `syracuse` contenant deux fonctions
 - `CalculVol(n int)` qui renvoie la durée et l'altitude maximale
 - `AfficheVol(d int, a int)` qui affiche la durée et l'altitude maximale
- Le `main` appellera ces deux fonctions pour les valeurs $1 \leq N \leq 100$

Le langage Go

Quelques fonctions utiles...

- Pour tirer un nombre aléatoire entre 0 et N

```
import "math/rand"  
  
rand.Intn(N)
```

- Pour lire au clavier

```
import "fmt"  
  
var essai int  
  
fmt.Scan(&essai)
```

Exercice

- Implémenter le jeu consistant à deviner un nombre tiré aléatoirement entre 1 et 100.

Le langage Go

Retour sur le switch

- Il existe une forme de switch appelée le blank switch ou switch sans condition
 - Il ne contient pas de clause de test
 - Il évalue les cas dans l'ordre et s'arrête au premier qui renvoie true

```
switch {  
    case f(1,2): ...  
    case rand.Int()<17: ...  
    default: ...  
}
```

- Il remplace agréablement une série de if-else

Le langage Go

La fonction `init`

- Toute fonction `init` définie dans un package sera automatiquement exécutée une et une seule fois préalablement à l'exécution du `main`
- Il peut y avoir autant de fonctions `init` que l'on souhaite dans un fichier/package.
 - dans un même fichier, elles seront exécutées dans l'ordre d'apparition
 - dans plusieurs fichiers (même package) c'est normalement l'ordre lexicographique des noms de fichiers (ordre par défaut des outils)

Le langage Go

La fonction `init`

- Toute fonction `init` définie dans un package sera automatiquement exécutée une et une seule fois préalablement à l'exécution du `main`
- Il peut y avoir autant de fonctions `init` que l'on souhaite dans un fichier/package.
 - dans un même fichier, elles seront exécutées dans l'ordre d'apparition
 - dans plusieurs fichiers (même package) c'est normalement l'ordre lexicographique des noms de fichiers (ordre par défaut des outils)

Le langage Go

Les constantes

- Le mot `const` permet de déclarer des constantes (globales ou locales)

```
const (  
    PI = 3.141  
    E = 2.718  
)
```

Le langage Go

iota

- L'identificateur «magique» `iota` sert à simplifier la définition des constantes
 - Pour chaque utilisation de `const`, `iota` est initialisé à 0
 - À chaque utilisation de `iota` elle est incrémentée de 1

```
const (  
    LUNDI = iota  
    MARDI = iota  
    MERCREDI = iota  
)
```

```
const (  
    JANVIER = iota  
    FEVRIER = iota  
    MARS = iota  
)
```

```
const (  
    _ = iota  
    UN = iota  
    DEUX = iota  
    TROIS = iota  
    CINQ = iota+1  
    X = 2*iota  
)
```

Le langage Go

Les types de base

- Entiers : `int`, `int8`, `int16`, `int32`, `int64` ($|\text{int}| \geq 32$ bits)
- Entiers non signés : `uint`, `uint8`, `uint16`, `uint32`, `uint64`
- Flottants : `float32`, `float64` (IEEE 754)
- Complexes : `complex64`, `complex128` (ex.: `5+3i`)
- Booléens : `bool`
- Caractère : `rune`
- Chaîne de caractères : `string`
- Autres : `byte`, `error`

Le langage Go

Les types de base

- Aucune conversion implicite n'existe entre types de base !!!
- Il faut utiliser une conversion explicite

```
var i int = 666  
var i32 int32  
  
i32 = int32(i)
```

Le langage Go

Les types de base et littéraux

```
12 // décimal  
012 // octal  
0x12 // hexadécimal ou 0X12  
0b10101010_10101010 // binaire
```

```
1.5e16 // flottant...
```

```
true, false
```

```
'A' '😊' '\U0001F600'
```

```
"Bonjour"
```

Le langage Go

Les pointeurs

- Ils fonctionnent comme en C mais la syntaxe diffère

```
var p *int
```

- Mais l'échappement est possible !

```
func f() *int {  
    i := 666  
    return &i  
}
```

- la variable sera créée dans le tas et non la pile
- Opérateur new (voir plus loin)

Le langage Go

Les tableaux

- Ils sont nécessairement dimensionnés (attention, si pas de taille ce sont des slices!!!! voir plus loin)

```
var t = [50]int {}  
var t2 = [12]int { 1, 2, 3 }  
var t3 = [...]int { 1, 2, 3 }  
var t4 = [10]int { 5:22, 4:21 }
```

- La taille est obtenue par la fonction `len()`

```
lg := len(t2)
```

- Opérateurs `[]`, `==`, `!=`, `&`

Le langage Go

Les slices

- Un slice est un « tableau » dynamique (taille dynamique + capacité)

- Le type d'un slice

```
var s []int
```

- La création d'un slice de longueur 20

```
s = make([]int, 20)
```

- La taille est obtenue par la fonction `len()`

```
lg := len(s)
```

- L'opérateur `[]` ne fonctionne que sur les éléments existants !
- On peut aussi créer un slice initialisé

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

Le langage Go

Les slices

- Pour ajouter un(des) élément(s)

```
s := append(s, 1, 2, 3)
```

- Toujours récupérer le retour de append
- Le type slice se manipule comme une sorte de pointeur sur un tableau qui contient les éléments et append peut faire de la réallocation pour agrandir le tableau...

Le langage Go

Les slices

- On peut créer un slice à partir d'un tableau

```
t := [...]int { 1, 2, 3, 4, 5 }  
s := t[2:4]
```

- Attention, le slice utilise le tableau comme stockage sous-jacent (sauf en cas d'utilisation de `append`)

```
s[0] = 111  
fmt.Println(t, s)  
s = append(s, 9, 8)  
fmt.Println(t, s)  
s[0] = 222  
fmt.Println(t, s)
```

Le langage Go

Les slices

- Pour supprimer un élément on peut écrire

```
s = append(s[:i], s[i+1:]...)
```

- Il existe le package `slices` contenant différentes fonctions utiles de manipulation des slices
 - dont une fonction pour supprimer des éléments...

Exercice

- Reprendre «syracuse»
- Cette fois on modifie la suite de sorte que étant donné K impair on calcule
 - $S_{n+1} = S_n/2$, si n est pair
 - $S_{n+1} = 3S_n+K$, si n est impair
- La suite « boucle » mais on ne sait pas exactement comment (pas toujours sur 4, 2, 1)
 - Il faut donc détecter les cycles en employant un slice des valeurs atteintes au fur et à mesure...

Le langage Go

Les maps

- Le type est `map[K]V`
- On la crée avec

```
m := make(map[int]string)
```
- On ajoute une paire avec

```
m[1] = "un"
```
- On supprime une entrée avec

```
delete(m, 1)
```
- On récupère une valeur avec

```
v := m[1]
```

 - si la clé n'existe pas => valeur nulle du type
- ou avec

```
v, e := m[1]
```

 - où e indique si l'opération a fonctionné ou pas : true ou false

Le langage Go

Note sur un idiome courant de Go

- En ce qui concerne la gestion des erreurs, l'idiome courant est de renvoyer un tuple

```
v, e := uneFonction(...)
if e { ...traitement erreur... }
else { ...v... }
```

- e peut-être un booléen et la recommandation est que si `e==true`, la valeur `v` ne doit pas contenir quelque chose d'intéressant, si `e==false`, la valeur de retour est à consulter dans `v`
- Si on souhaite qu'une erreur soit plus complexe qu'un simple booléen, il faudra utiliser les interfaces (voir plus loin)

Le langage Go

Les maps

- On peut créer une map en l'initialisant

```
m := map[int]string { 1:"un", 2:"deux" }
```

- On peut comparer des maps ==, !=
- On peut écrire

```
m := map[int]string { 1:"un", 2:"deux" }  
m[3] += "trois"
```

- même si la clé n'existe pas, si la clé est absente la valeur nulle du type est utilisée

Le langage Go

L'itérateur range

- L'opérateur range est défini pour les tableaux, les slices, les maps, les strings, les channels...
- Il retourne en général un tuple de deux résultats

<code>range tableau ou range slice</code>	<code>indice</code>	<code>valeur</code>
<code>range string</code>	<code>indice</code>	<code>rune</code>
<code>range map</code>	<code>clé</code>	<code>valeur</code>

- range utilise une copie de l'élément

Le langage Go

L'itérateur range

- Exemple

```
m := map[int]string { 1:"un", 2:"deux" }  
  
for k, v := range m {  
    fmt.Println(k, v)  
}
```

Le langage Go

Le type string

- Les littéraux sont

```
"Bonjour\tAu revoir"
```

- ou

```
`Bonjour\tAu revoir  
À bientôt`
```

- en ce cas il n'y a aucun échappement et les retours à la ligne sont conservés

Le langage Go

Le type string

- len permet d'obtenir la longueur **en bytes**

```
s := "Bonjour 🚒"  
fmt.Println(s) // 12 bytes
```

- Les opérateurs +, ==, !=, >, >=, <, <=, []
- Sous-chaîne

```
fmt.Println(s[:3])
```

- Le package strings

Le langage Go

Le package strconv

- Ce package contient de nombreuses fonctions de conversion entre le type `string` et des types de base

```
s := "1234"  
i := strconv.ParseInt(s, 10, 32)  
i = strconv.Atoi(s)
```

```
i := int64(1234)  
  
s := strconv.FormatInt(i, 16)
```

Le langage Go

Le type rune

- C'est le type des caractères
 - codage Unicode UTF-8 vs octets utilisés pour le codage

```
s := "Bonjour 🚒"  
rs := []rune(s)  
s = string(rs)
```

```
s := "Bonjour 🚒"  
bs := []byte(s)  
s = string(bs)
```

- Le package unicode/utf8

```
s := "Bonjour 🚒"  
l := utf8.RuneCountInString(s) // 9
```

Exercice

- Rechercher les anagrammes dans une liste de chaînes de caractères
 - apéro, coucou, gare, opéra, rage, petit, manoir, minora, romain, poisson, 😎 (je suis cool), je suis 😎 (cool)
- Indice : construire pour un mot donné, la map qui à une rune associe le nombre de fois qu'elle apparaît dans la chaîne... Comparer.

Le langage Go

Les types structurés

- C'est le type des structures (agrégation de champs)

```
type identificateur struct {  
    champs // identificateur type  
}
```

```
type identificateur = struct {  
    champs // identificateur type  
}
```

```
type Point struct {  
    X, Y float32  
}
```

```
type Point = struct {  
    X, Y float32  
}
```

Le langage Go

Les types structurés

- Attention, la forme

```
type identificateur = struct {  
    champs // identificateur type  
}
```

- créé un alias de type... (peut-être le verra t'on plus loin, qui sait ?)

Le langage Go

Les types structurés

- Les opérateurs de comparaison `==` et `!=` comparent champ par champ
- L'opérateur `.` permet d'accéder aux champs

Le langage Go

Les types structurés et l'initialisation

```
type Point struct {  
    X, Y float32  
}  
  
p := Point{10.0, 54.5}  
  
p2 := Point{X:10.0, Y:54.5} // recommandé
```

- L'initialisation peut être partielle

Le langage Go

Les types structurés et pointeurs

- La syntaxe d'accès aux champs est uniforme

```
type Point struct {  
    X, Y float32  
}  
  
p := Point{10.0, 54.5}  
pp := &p  
p.X = 12.7  
pp.Y = 66.6
```

- Attention à l'accessibilité des champs (première lettre)

Le langage Go

Structures et objets

- On peut associer des fonctions aux structures

- donc des méthodes

```
type Point struct {  
    X, Y float32  
}  
func (p Point) GetX() float32 {  
    return p.X  
}
```

- Attention, si on veut des modificateurs il faut passer par pointeur

```
type Point struct {  
    X, Y float32  
}  
func (p *Point) SetX(x float32) {  
    p.X = x  
}
```

Le langage Go

Structures et objets

- La définition d'une méthode `String() string` permet de contrôler l'affichage

```
type Point struct {  
    X, Y float32  
}  
func (p Point) String() string {  
    return "...";  
}
```

- Cela en fait un type conforme à l'interface `Stringer` (voir les interfaces plus loin)

Exercice

- Créer un module avec un package individus avec un type Personne contenant les champs: nom, prénom, âge.
 - Lui associer des méthodes permettant d'accéder aux champs en lecture
 - Lui associer une méthode pour modifier l'âge (uniquement croissant)
 - Créer dans un autre module un package main utilisant Personne pour en créer une et l'afficher agréablement (ex.: "Je m'appelle Sandra Klingon et j'ai 12 ans")
- Modifier le code de sorte qu'on ne puisse accéder directement aux champs
 - Que faut-il ajouter au package individus pour pouvoir en créer ?
Indice : une usine...

Le langage Go

Structures imbriquées

```
type Point struct {  
    X, Y float32  
}  
type Segment struct {  
    Origine, Extrémité Point  
}  
  
s := Segment{ Origine:Point{X:1,Y:2},  
              Extrémité:Point{X:3,Y:4} }
```

Le langage Go

Composition de structures

- On peut composer des structures

```
type Point struct {  
    X, Y float32  
}  
type PointColoré struct {  
    Point Point  
    Couleur int  
}
```

Le langage Go

Composition de structures

- On peut composer des structures par aplatissement (c'est une sorte d'héritage)
- Attention cette composition ne fournit pas de polymorphisme par sous-typage. Il y a bien des objets en Go, mais ce n'est pas un langage orienté objet.

```
type Point struct {  
    X, Y float32  
}  
type PointColoré struct {  
    Point // type  
    Couleur int  
}
```

- Les champs de Point sont directement visibles dans PointColoré

Le langage Go

Composition de structures

- Les méthodes de Point sont applicables aux PointColoré (c'est aussi une forme d'héritage)

```
type Point struct {  
    X, Y float32  
}  
func (p Point) GetX() int {...}  
type PointColoré struct {  
    Point  
    Couleur int  
}  
pc := PointColoré{X: 10, Y:20, Couleur:999}  
pc.GetX()
```

Exercice

- Introduire dans le package individus un type `PersonneRâleuse`
 - qui à l'affichage produit `"Pffff, moi c'est Kévin Trottin et j'ai 25 ans"`
 - Bien entendu, les méthode d'accès aux champs seront aussi disponibles...
 - Créer une usine adéquate
 - Tester

Le langage Go

Interfaces et polymorphisme structurel

- Le polymorphisme structurel (un *row polymorphism*) vérifie qu'un type est conforme à un contrat (en général un ensemble de méthodes) sans que le type ne le déclare explicitement.
- Cette vérification est statique (réalisée à la compilation)
- Le contrat est explicité et en Go il prend la forme d'une interface
- Attention, c'est différent du *duck typing* de Python par exemple qui lui effectue la vérification dynamiquement à l'exécution et ne nécessite pas de définition du contrat (il est implicite dans le code)

Le langage Go

Interfaces et polymorphisme structurel

- On peut donc définir en Go une interface

```
type Named interface {  
    GetName() string  
}
```

- Une collection de prototypes de méthodes
- C'est le « contrat »...

Le langage Go

Interfaces et polymorphisme structurel

- N'importe quel type qui remplit (implicitement) le contrat peut être utilisé

```
type Named interface {  
    GetName() string  
}  
func Print(o Named) {  
    fmt.Println(o.GetName())  
}
```

```
type Person struct {  
}  
func (p Person) GetName() string {  
    return "Joe";  
}
```

```
p = Person{}  
Print(p)
```

- Il n'y a pas de lien explicite entre Named et Person !!!!

Le langage Go

Interfaces et polymorphisme structurel

- Une interface peut typer une variable...

```
type Named interface {  
    GetName() string  
}  
func Print(o Named) {  
    fmt.Println(o.GetName())  
}
```

```
type Person struct {  
}  
func (p Person) GetName() string {  
    return "Joe";  
}
```

```
var p Named = Person{}  
Print(p)
```

- ou *via* un pointeur

```
var p Named = &Person{}  
Print(p)
```

Exercice

- Implémenter une fonction qui permet d'afficher un Point ou un PointColoré
- Que faut-il créer en plus ?

Le langage Go

Interfaces et polymorphisme structurel

- Comme les structures, on peut composer les interfaces
 - Cela induit un sous-typage (implicite)

```
type Named interface {  
    GetName() string  
}  
type NamedAndAged interface {  
    Named  
    GetAge() int  
}
```

Le langage Go

Interfaces et polymorphisme structurel

- Il existe de nombreuses interfaces dans la bibliothèque standard
 - Reader, Writer, Stringer
- Il existe aussi any qui est l'interface vide, tout type est conforme à cette interface !

```
type any interface {}
```

Le langage Go

Aliasing de type

- On peut définir un alias de type
- Par exemple :

```
type Color = package.Color
```

- Le type `Color` est exactement le même que `package.Color`
- Cette construction sert surtout au refactoring

Le langage Go

Type avec sous-jacent

- On peut définir un type utilisant une représentation sous-jacente
- Par exemple :

```
type Celsius float64
```

- Le type Celsius est **différent** de float64
 - On ne ne peut les *mélanger* sans conversion explicite

```
t := Celsius(100)  
v := 10  
t = v // interdit
```

- Le type défini possède les opérations du type sous-jacent (mais pas les méthodes) : `[]`, `+`, etc.

Le langage Go

Type avec sous-jacent

- Les types avec représentation sous-jacente sont très utiles pour typer des données en assurant une cohérence équivalente à celle de l'analyse dimensionnelle en physique
 - On additionne pas des mètres et des secondes...
- D'un type avec représentation sous-jacente on peut extraire la valeur typée par le sous-jacent

```
var t Celsius := 100  
var v int := 10  
v := int(t)
```

- et l'inverse

```
var t Celsius := 100  
var v int := 10  
t := Celsius(v)
```

Le langage Go

Type avec sous-jacent

- Il n'y a pas d'héritage de méthode avec les sous-jacents mais on peut définir des méthodes sur le type défini

```
type Celsius float64
func (t Celsius) String() string {
    return fmt.Sprintf("%.2f°C", t)
}
```

```
var t Celsius := -273.15
fmt.Println(t) // -273.15°C
```

Le langage Go

Assertion de type

- La conversion de type à partir d'une interface est interdite
- Mais on peut utiliser l'assertion de type

```
var o any = ...  
i, e := o.(int)  
if !e { // la conversion a fonctionné  
}
```

- Si le type choisi n'est pas le bon, e vaut faux, sinon vrai
 - si l'assertion est vraie, i vaut la valeur désignée par l'interface, sinon i vaut 0 (le zéro du type).

Le langage Go

Assertion de type

- On peut aussi utiliser (type) pour « extraire » le type (concret!!!)

```
func doSomething(o any) {  
    switch o.(type) {  
        case int32:  
            v := int(o.(int32))  
            fmt.Println("Nombre: "+strconv.Itoa(v))  
        case string:  
            fmt.Println("String: "+strconv.Itoa(len(o.(string))))  
        default:  
            fmt.Println("????")  
    }  
}
```

```
doSomething(32)  
doSomething(int32(32))  
doSomething("Hihi")
```

Le langage Go

La généricité

- Elle permet de paramétrer une fonction à l'aide de types

```
func MyPrint[T any](v T) {  
    fmt.Println("J'affiche", v)  
}
```

- il existe la possibilité de contraindre la généricité

```
func AddAll[K comparable, V int|float](m map[K]V) V {  
    var s V  
    for _, v := range m {  
        s += v  
    }  
    return s  
}
```

Le langage Go

La généricité

- Elle permet de paramétrer une structure à l'aide de types

```
type Wrapper[T] struct {  
    value T  
}
```

Exercice

- Définir un type de liste (linked list) générique
 - une usine de création de liste vide `CreateList()`
 - avec quelques méthodes `AddToHead(v)`, `AddToTail(v)`
 - une méthode de test `IsIn(v, comparator)`
 - définir la méthode `String()`
 - définir la méthode `Remove(v, comparator)`
- Tester

```
l := CreateList[int32]()
fmt.Println(l)
l.AddToHead(int32(45))
fmt.Println(l)
l.AddToHead(int32(55))
fmt.Println(l)
l.AddToTail(int32(777))
fmt.Println(l)

comparator := func(v1, v2 int32) bool { return v1 == v2 }

fmt.Println(l.IsIn(int32(666), comparator))
fmt.Println(l.IsIn(int32(45), comparator))
```

```
[ 45 ]
[ 55, 45 ]
[ 55, 45, 777 ]
[ 55, 45, 777 ]
false
true
```

Le langage Go

new

- `new(type)` permet d'allouer dynamiquement de la mémoire pour le type structuré correspondant
 - les champs sont initialisés à «zéro»
 - il faut donc initialiser les champs «à la main»
 - la valeur renvoyée est un pointeur du type
- L'environnement d'exécution de Go possède un garbage collecteur

Le langage Go

defer

- la directive `defer` *appel de fonction* permet d'obtenir l'exécution de l'appel à la sortie de la fonction qui contient l'instruction
- elle sert principalement à la gestion des ressources en permettant leur libération quoiqu'il arrive

```
func f() {  
    file, err := os.Open("fichier.txt")  
    if erreur == nil { ...; return }  
    defer file.Close()  
    ....  
}
```

- `defer` est exécuté après l'évaluation des valeurs de retour
- les `defer` sont exécutés dans l'ordre inverse de leur apparition

Le langage Go

error

- Il existe un type error permettant d'indiquer une cause d'erreur
 - petit idiome Go : $v, e = F()$
 - voir le package errors

Le langage Go

Les paniques

- Il s'agit d'un système primitif d'exception
 - la fonction `panic(any)` permet d'arrêter l'exécution courante et d'exécuter tous les `defer`
- Pour récupérer une panique il faut utiliser la fonction `recover` (dans un `defer` donc)
 - la fonction renvoie ce qui a été envoyé par l'appel à `panic` et `nil` sinon
 - si une panique est captée, le fonctionnement reprend alors normalement

Le langage Go

Les goroutines

- La directive `go` *appel de fonction* permet d'exécuter en concurrence la fonction
 - lorsque la fonction termine, la goroutine (ou le thread) s'arrête
- Attention, l'exécution s'arrête lorsqu'on sort du `main`
- L'ordonnanceur est préemptif depuis 1.14

Le langage Go

Les WaitGroup

- Si on souhaite se synchroniser avec le terminaison de goroutines
 - il faut utiliser un WaitGroup
 - C'est une sorte de compteur
 - On peut l'incrémenter `Add(n int)`
 - On peut le décrémenter `Done()`
 - On peut attendre qu'il soit nul `Wait()`
- Une goroutine peut aussi être lancée *via* `Go(fonction)`

- TP n°1 (goroutines, Threads Java)