



Université Paris Cité

Cours Go

Jean-Baptiste Yunès

yunes.informatique.univ-paris-diderot.fr

Jean-Baptiste.Yunes@u-pariscite.fr

v26.11.10

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 1.27.2)
- 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

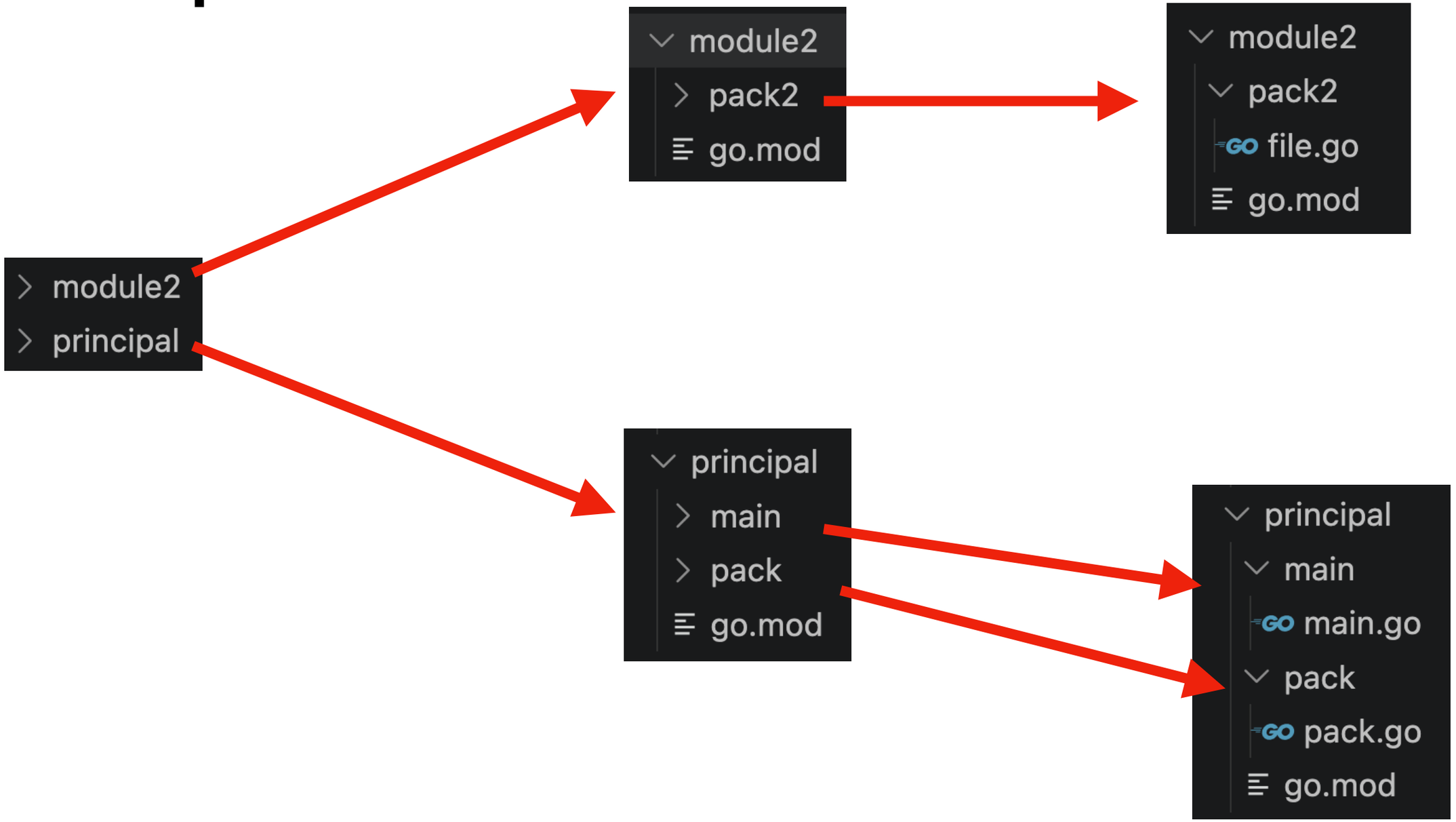
Anatomie d’un projet Go

Généralités

- Un module <=> un répertoire + `go.mod`
- Un package <=> 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` :

```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`
  - un seul fichier `go.work` pour un « projet »
    - à la racine du projet
- 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 élements syntaxiques

```go
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)
}
```

déclaration du package

utilisation de packages

définition d'une fonction

déclaration de variables locales typées

appel de fonction

type de retour (tuple)

renvoi d'un tuple de valeurs

déclaration de paramètres

Le langage Go

Quelques élements 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, etc

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 définit 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 pré-déclarés 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[Typ
func close(c chan<- Type)
func complex(r, i FloatType) Com
func copy(dst, src []Type) int
func delete(m map[Type]Type1,
func imag(c ComplexType) Float
func len(v Type) int
func make(t Type, size ...IntegerT
func max[T cmp.Ordered](x T, y
func min[T cmp.Ordered](x T, y .
func new(TypeOrExpr) *Type
func panic(v any)
func print(args ...Type)
func println(args ...Type)
func real(c ComplexType) FloatTy
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
```

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

```
true
false
```

```
iota
```

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 _
  - 𑁍𑁌𑁍𑁋𑁋23 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 litéraux

- 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 `}`

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


}
```

- Bizarrerie ???

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
  - une variable locale peut cacher une variable de même nom dans le bloc supérieur

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 résolu par les dépendances

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

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

```
var x1, x2, x3 = x3+1, 1, x2+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
- Il existe une forme courte avec `:=` (préférée et conseillée) 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 n°1

- La suite de Syracuse du nombre N est définie ainsi :
  - $S_0 = N$
  - $S_{n+1} = S_n/2$, si S_n est pair
  - $S_{n+1} = 3S_n+1$, si S_n est impair
- On définit :
  - le vol comme la plus petite suite préfixe 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
- Exemple: S = 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1; durée=16; altitude=52

Exercice n°1 (suite)

- 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 et les cas sont examinés dans l'ordre d'apparition
  - 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 (voir un peu plus loin)...

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
}
```

v1 est une variable de «retour»

v1 est utilisée comme n'importe quelle variable

return renvoie les variables de retour

- Il n'y a pas de surchage 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) {
    incr := 1
    g := func(n int) { return n+incr; }
    v1, v2 = g(i), g(i)+34
    return
}
```

capture de la lambda dans une variable

définition d'une lambda

Exercice n°2

- Cette fois on crée un module `syracuse` contenant deux fonctions
  - `CalculVol(n int) (int,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)
```

opérateur de prise d'adresse
(similaire à l'opérateur du C)

Exercice n°3

- Implémenter le jeu consistant deviner un nombre tiré aléatoirement entre 1 et 100. L'utilisateur est invité à saisir un nombre tant qu'il ne trouve pas le bon.

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
  - on ne peut appeler une fonction `init` explicitement
- 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` (|`int`|≥32 bits)
- Entiers non signés : `uint`, `uint8`, `uint16`, `uint32`, `uint64` (|`uint`|≥32 bits)
- 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 !!!
  - Go est très fortement typé
  - 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 OX12
0b10101010_10101010 // binaire

1.5e16 // flottant...


true, false

'A' '😀' '\U0001F600'

"Bonjour"
```

Le langage Go

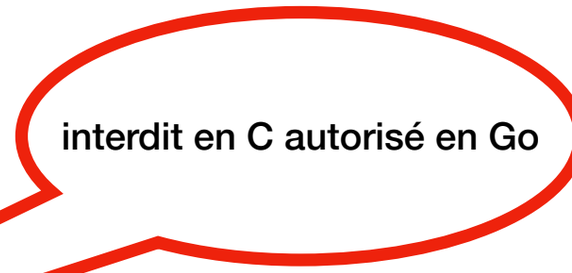
Les pointeurs

- Ils fonctionnent comme en C mais la syntaxe diffère (type derrière)

```
var p *int
```

  - 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 et de réallocation)

```
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 n°4

- 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 (générique) est `map[K]V`
- On crée une map avec

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

- On ajoute/modifie 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 (préféré!)

```
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) et renvoyer `nil` en cas de défaillance

Le langage Go

L'idiome appliqué à une map

- Si on veut n'ajouter une paire dans une map et non modifier une entrée, il faut donc tester l'existence de la clé

```
if _,e := m[clé]; !e {
    m[clé] = valeur
}
```

ok ou pas ok ?

valeur de retour non utilisée

la clé n'existe pas ?

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 (éléments par éléments) ==, !=

- 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

`range tableau` ou `range slice`	indice	valeur
`range string`	indice	rune
`range map`	clé	valeur

- `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 (mode verbatim) 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
```

- le nombre de bytes est différent du nombre de runes!

Exercice n°5

- 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... (voir plus loin)

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
  - ce qui n'est pas explicitement initialisé est initialisé à la valeur zéro du type (0 pour les nombres, false pour les booléens, etc.)

Le langage Go

Les types structurés et pointeurs

- La syntaxe d'accès aux champs est uniforme (via une variable ou un pointeur)

```
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
  - l'appel d'une méthode m sur l'objet o utilise la syntaxe

```
o.m(params)
```

- un tel appel correspond à un appel de fonction `m(o,params)` si le passage est par valeur sinon `m(&o,params)`

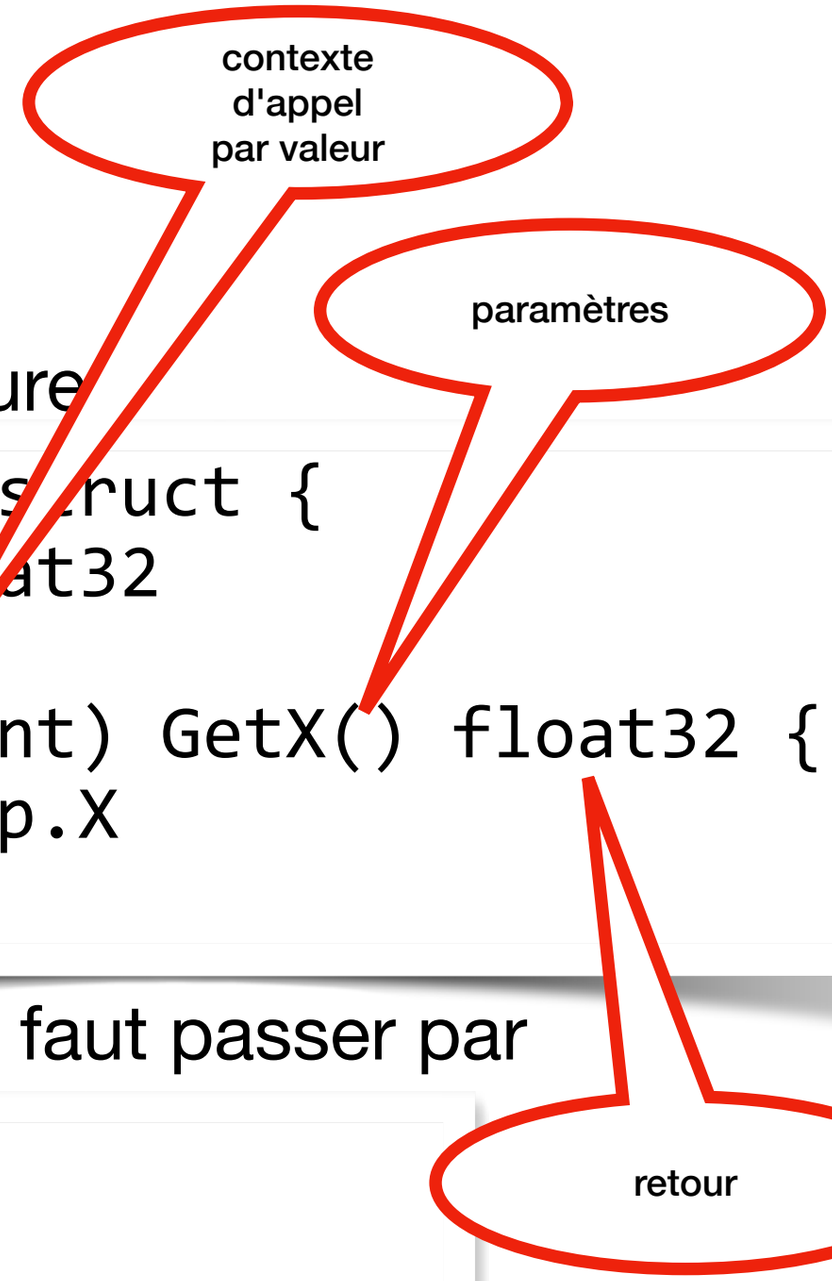


Le langage Go

Structures et objets

- On peut associer des fonctions aux structure
  - 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 confirme à l'interface standard `Stringer` (voir les interfaces plus loin)

Exercice n°6

- 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 n°7

- Introduire dans le package individus un type `PersonneRâleuse`
  - qui à l'affichage produit `"Pfff, moi c'est Kévin Trottin et j'ai 25 ans"`

- Bien entendu, les méthodes d'accès aux champs seront aussi disponibles...
- Créer une usine adéquate
- Tester

Le langage Go

Interfaces et polymorphisme structurel

- Le polymorphisme structurel (une forme de *raw 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 définit en Go une interface

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

- Une collection de prototypes de méthodes (pas de code, juste la signature)

- 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 n°8

- 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
- On ne peut pas définir de méthode sur un alias...

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 similaire 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

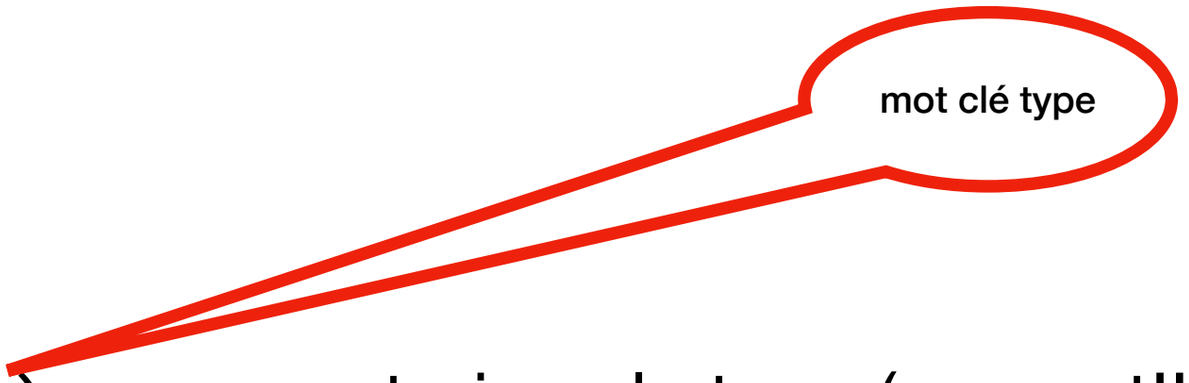
- La conversion de type à partir d'une interface est interdite
- Mais on peut utiliser l'assertion de type

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

- Si le type choisi n'est pas le bon, e vaut `false`, sinon `true`
  - 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 n°9

- Définir un type de liste chaînée (*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
```

Exercice n°10

- Implémenter un clone de « Akinator ». Pour cela il faut implémenter une structure d'arbre binaire de recherche: les nœuds sont les questions et les feuilles les réponses. Le jeu est donc un parcours d'arbre binaire de recherche guidé par l'utilisateur. On ne se posera pas le problème du stockage de l'arbre, tout fonctionnera en mémoire.
- Le fonctionnement est le suivant:
    - L'utilisateur se voit poser une question fermée (les seules réponses possibles sont oui ou non) : « Est-ce que c'est vert ?" réponse « oui ».
    - Selon la réponse Akinator pose une autre question : « Est-ce que c'est gros ?» réponse « non »
    - La fin est atteinte lorqu'il n'y a plus de question dans la branche, la dernière est par exemple : « Est-ce que c'est un petit pois ? »
        - Si la réponse est « oui », c'est terminé Akinator a gagné.
        - Si c'est « non », Akinator demande « À quoi pensais-tu ? »
            - L'utilisateur donne une réponse « Une sauterelle »,
            - Akinator demande « Propose-moi une question qui permet de différencier une sauterelle d'un petit pois ».
            - L'utilisateur propose « Est-ce que c'est un animal ? [oui, sauterelle] [non, petit pois] »
            - L'arbre est alors modifié en conséquence et on retourne au début.
- La version originale est disponible sur https://fr.akinator.com/. Attention, l'implémentation demandée ne contient que des choix de type oui/non.

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` en remontant la pile d'appels
- Pour récupérer une panique il faut utiliser la fonction `recover` (nécessairement placée 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 sychroniser 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

- TP n°1 (goroutines, Threads Java)

Le langage Go

L'atomicité

- On dira qu'une opération (une série d'instructions) est atomique si elle est indivisible (du point de vue de la concurrence)
    - Une opération est atomique si du point de vue du reste (ce qui est en dehors de cette opération) elle s'exécute d'un seul bloc:
        - soit elle s'exécute entièrement;
        - soit elle ne s'exécute pas.
- Un exemple typique est la notion de transaction dans les bases de données.

Le langage Go

Le package `sync.atomic`

- Le package `sync.atomic` contient de nombreuses constructions (types, fonctions, méthodes) permettant d'obtenir des opérations atomiques.
- Le type `sync.atomic.Int32` correspond à un type `int32` rendu atomique
  - On peut par exemple incrémenter une telle variable en utilisant la méthode :
    `func (i *Int32) Add(d int32) (int32)`
    - Attention aux types dans la signature. La valeur renvoyée est la valeur du sync.atomic.`Int32` après l'ajout de la valeur `d`; l'effet de bord est de modifier le `Int32`...

Le langage Go

Le package `sync.atomic`

- On trouve aussi la fonction :
  `func AddInt32(i *int32, d int32) (int32)`
  - Cette fois, on ne passe pas par le type `Int32`, mais on utilise une fonction qui modifie de façon atomique un `int32` passé par pointeur.
- Les deux écritures sont équivalentes pour la variable `r` :

```
var i sync.atomic.Int32
var r := i.Add(int32(5))
```

```
var i int32
var r := AddInt32(&i, int32(5))
```

Le langage Go

Le package sync et le verrouillage

- Une atomicité plus explicite utilise un verrou (*lock*).
  - Le motif correspond à celui de l'acquisition d'une ressource (ici l'acquisition du verrou):

    obtention du verrou
    opération
    libération du verrou

- L'obtention du verrou est une opération bloquante si une goroutine l'a déjà acquis
- La libération libère une goroutine qui serait en attente du verrou

Le langage Go

Le package sync et le verrouillage

- Le type `sync.Mutex` correspond à un verrou (mutex : mutual exclusion)
  - Il propose 3 méthodes:
    - `Lock()` pour fermer le verrou
    - `UnLock()` pour ouvrir le verrou (ne doit pas être employé si le verrou n'est pas fermé)
    - `TryLock()` pour essayer de fermer le verrou en mode non-bloquant (ne pas employer si possible)
- Il existe l'interface `sync.Locker` qui correspond à tout objet réalisant la contrat `Lock`/`UnLock` (peut donc être employé par exemple pour créer son propre verrou, voir la partie théorique du cours)

Le langage Go

Le package sync et le verrouillage

- On l'emploi souvent de la façon suivante :

```
var l sync.Mutex // partagé par des goroutines

l.Lock()
// instructions
l.UnLock()
```

- Quand utiliser un `atomic` ou un verrou ?
  - Sur un type de base, il est préférable et plus efficace d'utiliser un `atomic` correspondant
  - Lorsque l'opération correspond à une suite d'instructions plus complexes qu'une opération élémentaire, il vaut mieux un verrou

Et en Java ???
`volatile`

- Attention si en Java des mécanismes similaires existent, il y a une petite subtilité supplémentaire :
    - Il existe le mot-clé `volatile` qui permet de qualifier une variable
        - Cette qualification garantit que toute modification par un Thread d'une telle variable est immédiatement visible de tous les autres Thread
            - Pas de mise en cache de cette variable
- Attention `volatile` ne garantit aucune synchronicité !!!
- En Go on en a pas besoin car l'utilisation de toute primitive de synchronisation garantit cette propriété de visibilité immédiate

Le langage Go

Les channels

- Le modèle CSP (Communicating Sequential Processes) de Sir Charles Antony Richard Hoare est disponible en Go
  - Ce modèle correspond à la philosophie suivante
    Do not communicate by sharing memory; instead, share memory by communicating
- Il repose sur les deux concepts suivants de Go:
  - les goroutines
  - les channels
- Les goroutines s'échangent des données *via* les channels plutôt que de les partager et les protéger par des verrous

Le langage Go

Les channels

- Un channel est donc un objet de communication et de synchronisation
  - C'est comme un tube (*pipe*) Unix, ou une socket réseau, etc.
    - On y envoie des données d'un côté
      - On y réceptionne les données de l'autre, les lectures sont destructrices
        - L'ordre est FIFO
- En Go les channels sont:
  - typés
  - peuvent avoir une capacité

Le langage Go

Les channels

- Il y a deux types de channels
  - Les channels sans capacité et qui constituent donc un point de rendez-vous
  - Les channels avec capacité et qui correspondent au modèle producteur/consommateur
    - Il y a découplage temporel entre les lecteurs et les écrivains

Le langage Go

Création d'un channel

- Le mot-réservé chan correspond au type générique des channels
- Pour créer un channel sans capacité :

```
var canal = make(chan int)
```

- Attention ! Avec un tel channel, une écriture est bloquante jusqu'à ce qu'une lecture soit tentée et vice-versa
    - C'est donc un point de rendez-vous

Le langage Go

Création d'un channel

- Le mot-réservé `chan` correspond au type générique des channels
- Pour créer un channel avec capacité :

```
var canal = make(chan int, 10)
```

- Avec un tel channel, une écriture est immédiate tant que le canal n'est pas plein (et réciproquement pour les lectures)
- Lorsque le canal est plein, une écriture est bloquante
- Lorsque le canal est vide, une lecture est bloquante

Le langage Go

Création d'un channel

- Un channel peut être typé par une interface :

```
var canal = make(chan any, 10)
```

- On peut restreindre un type channel :

```
var canal = make(chan int, 10)
var lecture <-chan int = canal
var ecriture chan<- int = canal
```

- `lecture` est le côté du canal d'origine où l'on ne peut effectuer que des lectures, son type est `<-chan int`

- `ecriture` est le côté du canal d'origine où l'on ne peut effectuer que des écritures, son type est `chan<- int`

Le langage Go

Création d'un channel

- Les canaux restreints sont utilisés pour assurer la sémantiques en cas de passage de paramètre :

```
func fonctionDeLecture(c <-chan int) {
  // on ne peut que lire...
}
func fonctionDEcriture(c chan<- int) {
  // on ne peut qu'écrire...
}
var canal = make(chan int, 10)
fonctionDeLecture(canal)
fonctionDEcriture(canal)
```

Le langage Go

Fermeture d'un channel

- La fermeture d'un channel s'effectue avec la fonction `close` :

```
var canal = make(chan int, 10)
close(canal)
```

Le langage Go

Écriture sur channel

- Une écriture sur channel s'effectue avec l'opérateur <-
- On écrit, par exemple, l'entier 666 dans le canal ainsi :

```
var canal chan int = ...
canal <- 666
```

- Attention ! L'écriture sur un channel fermé provoque une panic

Le langage Go

Lecture sur channel

- Une lecture sur channel s'effectue avec l'opérateur `<-`
- On extrait un `int` d'un canal ainsi :

```
var canal chan int = ...
data := <-canal // data est donc un int
```

- Les lectures sur un canal fermé :
  - Si le canal contient encore des données, elles sont extraites
  - Si le canal est vide le «zéro» du type est renvoyé
    - pour plus sûreté on emploiera l'idiome standard

```
var canal chan int = ...
data, ok := <-canal // data est donc un int
```

Le langage Go

Sélection sur channel

- Supposons qu'une goroutine désire, par exemple, faire des lectures sur deux channels
  - On se retrouve devant une difficulté car il faut bien choisir un ordre :

```
var canal1, canal2 chan int = ...
var data int

for {
  data = <-canal1
  ...
  data = <-canal2
  ...
}
```

- Si le canal1 est vide, on reste bloqué un temps non défini et le canal2 peut se remplir...
  - L'ordre inverse ne fonctionnera pas mieux

Le langage Go

Sélection sur channel

- L'opérateur `select` permet de demander si des opérations voulues peuvent être réalisées
  - Parmi celle qui peuvent être réalisées l'une d'entre elles sera choisie (au pseudo-hasard)

```
var canal1, canal2 chan int = ...
var data int

for {
  select {
    case data = <-canal1:
      fmt.Println(data)
    case data = <-canal2:
      fmt.Println(data)
  }
}
```

Le langage Go

Sélection sur channel

- On peut sélectionner des lectures et des écritures...

```
var canal1, canal2 chan int = ...
var data int

for {
  select {
    case data = <-canal1:
      fmt.Println(data)
    case data = <-canal2:
      fmt.Println(data)
  }
}
```

- Si le `select` contient un `default` la clause est vérifiée si aucune opération n'est possible

Le langage Go

Sélection sur channel

- Attention aux fermetures des channels dans les sélections !!!
  - Lorsqu'on détecte une fermeture, il veut mieux positionner le channel correspondant à `nil`, de la sorte les clauses le concernant sont ignorées

```
var canal1, canal2 chan int = ...
var data int

for {
  select {
    case data, ok = <-canal1:
      if !ok { canal1 = nil; continue }
      fmt.Println(data)
    case data, ok = <-canal2:
      if !ok { canal2 = nil; continue }
      fmt.Println(data)
  }
}
```

Le langage Go

Les variables de condition

- Le vérrouillage Lock/Unlock n'est pas toujours suffisant
- Exemple, la gestion d'une file (lecteurs à un bout, écrivains à l'autre). Le modèle est connu sous le nom du problème des producteurs/ consommateurs.
- La modification de la file (lecture ou écriture):
  - nécessite l'atomicité pour les opérations d'insertion ou suppression.
- mais ces opérations sont conditionnées:
  - un lecteur ne peut lire que si la file n'est pas vide.
  - un écrivain ne peut écrire que si la file n'est pas pleine.

Le langage Go

Les variables de condition

- En utilisant Lock/Unlock pour régler cette question on écrit quelque chose comme (en pseudo-code) :

```
// Lecture
Lock()
while (file_vide()) {
  Unlock()
  Lock()
}
remove_first()
Unlock()
```

```
// Écriture
Lock()
while (file_plein()) {
  Unlock()
  Lock()
}
add_last()
Unlock()
```

- Il y a de nombreux inconvénients.

Le langage Go

Les variables de condition

```
// Lecture
Lock()
while (file_vide()) {
  Unlock()
  Lock()
}
remove_first()
Unlock()
```

- Il y a de nombreux inconvénients dont :
  - attente active : une activité consomme du CPU pour rien s'il n'y a rien et symétriquement
  - absence d'équité : une activité pourrait se faire doubler sans fin par d'autres activités (cela conduit à la famine)
  - inversion de priorité : *on ne traitera pas cette question ici*

Le langage Go

Les variables de condition

- Pour régler cette question, on utilise des variables de condition :
- Une variable de condition est attachée à un verrou :
  - Wait() : On peut attendre sur une variable de condition (sous réserve d'avoir acquis le verrou).
    - Cette opération libère le verrou, suspend l'activité qui est placée dans une file d'attente
  - Notify()/NotifyAll() : permet à une activité de libérer une (ou toutes les activités en attente sur la condition.
    - Cette opération permet aux activités de reprendre leur travail
      - les activités elle-mêmes bloquées sur le Lock()
      - les activités bloquées sur la condition

Le langage Go

Les variables de condition

- En Go on utilise alors l'idiome suivant :

```
mutex sync.Mutex
notFull  *sync.Cond = sync.NewCond(&mutex)
notEmpty *sync.Cond = sync.NewCond(&mutex)
```

```
// Lecteur
mutex.Lock()
defer mutex.Unlock()
for isQueueEmpty() {
  notEmpty.Wait()
}
e := removeFirstFromQueue()
notFull.Broadcast()
```

```
// Écrivain
mutex.Lock()
defer mutex.Unlock()
for isQueueFull() {
  notFull.Wait()
}
addLastToQueue(e)
notEmpty.Broadcast()
```

Le langage Go

Les variables de condition

- Attention ! Cet idiome supprime l'attente active, mais ne garantit pas l'absence de famine
  - En pratique, elle ne se produit que rarement.
- Régler la famine nécessite des constructions plus complexes (voir cours théorique pour cette question)