

## Programmation en langage Go

- Présentation de Go
- modèle CSP
- Types de base, variables, affectations, opérateurs
- Structures, Tableaux
- Conditions
- Boucles
- Canaux
- Fonctions
- Goroutines
- Synchronisations
- Packages et utilitaires
- Méthodologie de conception

# Présentation de GO

---

- ❑ Créé en 2009 par Robert Griesemer, Rob Pike et Ken Thompson, et développé par Google
- ❑ Qualités souhaitées :
  - ❑ Expressif
  - ❑ Concis
  - ❑ Efficace
  - ❑ Construction de programme flexible et modulaire
- ❑ Langage impératif, instructions séquentielles
- ❑ Mécanismes spécifiques pour exprimer la concurrence et le parallélisme (inspirés du modèle CSP)
  - ❑ Pour multicores
  - ❑ Pour machines en réseau
- ❑ Compilé
- ❑ Typage statique
- ❑ Garbage collector
- ❑ Performant en temps d'exécution et utilisation mémoire

# Présentation de GO

## □ Performances de GO ?

- Tiré de “Energy efficiency across programming languages: how do energy, time, and memory relate?” , Rui Pereira et al. , 10th ACM SIGPLAN International Conference on Software Language Engineering, octobre 2017

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

# Présentation de GO

## □ Performances de GO ?

Table 5. Pareto optimal sets for different combination of objectives.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	Ocaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

# Présentation de GO

---

- Langage impératif séquentiel
- Inspiration
  - C pour la syntaxe, les opérateurs
  - Pascal pour les déclarations et gestions de paquet
  - CSP pour les communications
- Pas de délimiteur de fin de ligne (mais passage ligne obligatoire)

# Modèle CSP

---

- Modèle CSP (Communicating Sequential Processes)
  - Hoare (1974)
  - Théorie mathématique (algèbre de processus) pour spécifier et vérifier des comportements complexes issus d'interactions entre composants concurrents
  
- Caractéristiques
  - Asynchronisme des comportements
  - Synchronisation implicite par message
  - Composition et organisation hiérarchique des processus

# Pourquoi CSP ?

---

- Inclue les principales fonctions de communication
- Suffisamment expressif pour permettre la détection des interblocages
- Robuste et supporté par des outils commerciaux pour la conception logicielle (multi-processus) ou matérielle (systèmes sur puce)
- Neutre au niveau de l'architecture sous-jacente
- Permet de décrire un système à différents niveaux d'abstraction
  - Spécification
  - Conception
  - Implémentation

# Quelques langages et environnements intégrant des primitives CSP



Tiré de [http://www.csp-consortium.org/apps/CSP\\_Updates.pdf](http://www.csp-consortium.org/apps/CSP_Updates.pdf)



# Types de base

---

- ❑ **bool** : booléen de valeur *true* ou *false*
  - ❑ valeur par défaut : *false*
- ❑ **string**
  - ❑ valeur par défaut : ""
- ❑ **int**
  - ❑ valeur par défaut : 0
  - ❑ taille : 32 ou 64 en fonction du système
  - ❑ mais aussi int8, int16, uint, uint8,...
- ❑ Pour le typage des caractères :
  - ❑ **byte**, alias de uint8, pour caractères ASCII
  - ❑ **rune**, alias de int32, pour caractères unicode en UTF8
  - ❑ Littéral : 'A'
- ❑ **float32, float64**
  - ❑ valeur par défaut : 0.0
  - ❑ type par défaut pour déclaration courte : float64
- ❑ Conversion de type avec cast explicite
  - `a = float32(x)`

# Variables et affectations

---

- Toutes les variables sont typées (pas de polymorphisme)
- Affectation : *var = expr*
  - `x = 2`
- Type fixé manuellement
  - Exemple

```
var x int //déclaration d'un entier de nom x
x = 10    //x prend pour valeur 10
var y int = 5 //valeur initiale avec déclaration
```
- Déclaration courte : inférence de type automatique
  - Exemple

```
x := 10 //déclaration et initialisation d'un entier de nom x
```
- Affectations multiples (séparation par virgules)
  - `x,y,z = 3,10,9`

# Variables et affectations

---

## □ Allocation mémoire

- Les types de base qui ne nécessitent pas d'allocation mémoire
  - *bool, int, floatXX, string, array*
- Les types tableaux et canaux ont besoin d'une allocation mémoire avec le mot clé *make* :
  - *slice, map, chan*

## □ Définition de constantes

- mot clé *const*
- Exemples

```
const search int = 1
const Pi float32 = 3.14
const month string = "september"
```

# Opérateurs

---

- Opérateurs de base similaires aux opérateurs C
- Opérateurs de comparaison et logique :
  - Retournent un *bool*
- Opérateurs arithmétique
  - Pas de conversion implicite

# Structures

---

## □ Construction

```
type nomStructure struct {  
    nom_champ1 type1  
    nom_champ2 type2  
}
```

## □ Exemple

```
type maStruct struct{  
    x int  
    s string  
}
```

## □ Déclaration

```
var y maStruct
```

# Structures

---

## □ Affectation

```
m = maStruct {10, "test"}
```

## □ Déclaration courte

```
m := maStruct {10, "test"}
```

## □ Utilisation

```
val = m.x
```

```
fmt.Println(m.s)
```

## □ Valeurs par défaut

- valeurs par défaut des sous-types

# Tableaux : Array et Slice

---

- La 1<sup>ère</sup> case du tableau est d'indice 0

- Affectation

`t[3] = 10`

- Array

- Tableau de taille fixe

- Déclaration Array

`var t[3] int`

- Déclaration/Initialisation d'un array

`t := [3] int {100, 200, 150}`

- Slice

- Tableau de taille modifiable

- Déclaration Slice

`var t[] int`

- Allocation mémoire Slice

`t = make([]int, 5)`

- Déclaration courte Slice avec allocation

`t := make([]int, 5)`

- Ajout dynamique d'une valeur

`t = append(t, 500) // ajout d'une case en fin avec la valeur 500`

# Conditions

---

- if : similaire au C
- mais
  - pas de parenthèses
  - accolades obligatoires
  - respect des passages à la ligne :
    - *if condition* { : sur une même ligne
    - *} else {* : sur une même ligne

## □ Exemple :

```
if x > y {  
    x = x+1  
}else{  
    x = x-1  
}
```



# Conditions

---

- switch : similaire au C
  - mais pas de parenthèses et pas de *break*
- switch avec valeur de test

```
switch x {  
    case 0:  
        ...  
    case 1:  
        ...  
    default:  
        ...  
}
```

- switch sans valeur de test

```
switch {  
    case x == 3 :  
        instructions1  
    case y < b :  
        instructions2  
    default:  
        instructions3  
}
```

# Boucles

---

- Seule structure : boucle *for*
- 3 parties séparées par ;
  - initialisation
    - généralement déclaration courte, variables visibles seulement dans le *for*
  - condition
  - déclaration d'aboutissement exécutée à la fin de chaque itération
- Pas de parenthèses
- Accolades toujours nécessaires
  - même avec un corps de boucle de 1 ligne
- Exemple avec itérateur

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

# Boucles

---

- Déclaration d'initialisation et d'aboutissement facultatives
  - Pour obtenir le comportement d'une boucle while ou infinie

- Exemple : équivalent d'une boucle while

```
sum := 1
for ; sum < 1000; {
    sum += sum
}
```

- Simplification :

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

- Boucle infinie :

```
for {
    //traitement sans fin
}
```

# Boucles sur Tableaux

---

- Longueur d'un tableau

- mot clé *len*

- Exemple :

```
for i:=0 ; i<len(tab) ; i ++ {  
    sum = sum + tab[i]  
}
```

# Boucles sur Tableaux

---

## □ Itération avec *range*

### □ Itération sur les indices du tableau :

```
for i := range tab {  
    sum = sum + tab[i]  
}
```

### □ Itération sur les indices et les éléments du tableau :

```
for i,e := range tab {  
    sum = sum + e*i //e = tab[i]  
}
```

### □ Itération sur les éléments du tableau :

```
for _,e := range tab {  
    sum = sum + e  
}
```

# Canaux

---

- Communication entre processus (goroutines) par des canaux synchrones
- Mot clé *chan* et indication du type des messages (non modifiable)
- Déclaration

```
var can1 chan int
```

- Allocation mémoire

```
can1 = make (chan int)
```

- Déclaration/allocation courte

```
can1 := make (chan int)
```

# Canaux

---

## □ Communications :

- Réception (Lecture) dans un canal : `variable = <- canal`
- Envoi (Écriture) dans un canal : `canal <- variable`
- La valeur lue ou écrite doit être compatible avec le canal (même type)
- Cas particulier : réception sans stockage en mémoire
  - Dans le cas d'une synchronisation  
`<- canal`
- Plusieurs émetteur et récepteurs possibles
  - mais pas de diffusion : un message lu est retiré

## □ Types dans les canaux

- types simple : par copie (valeur)
- types tableaux : par référence (adresse)
  - non conseillé
  - peut entrainer des problèmes de mémoire partagée

# Canaux

---

- Deux formats de canaux : avec ou sans buffer (file d'attente)
- Canaux sans buffer : Synchrones
  - canaux *vides* (*pas de contenance*)
  - l'envoi d'un message est bloquant tant que le récepteur n'écoute pas le canal
  - un récepteur en écoute est bloqué tant que le message n'a pas été envoyé
  - Pas de problème d'ordre dans la réception des messages
  - Pas de problème d'excès de lecture ou d'écriture
  - Mais grandes possibilités d'interblocage
- Les canaux synchrone permettent à la fois
  - Des communications de valeurs
  - Des synchronisations, via la notion de rendez-vous bloquant
  - La valeur transmise n'est donc pas forcément *utile*
    - Par exemple si le canal est uniquement utilisé pour synchroniser deux processus



# Canaux

---

- Les canaux avec buffer (asynchrones)
    - contenance paramétrable
    - asynchrones
    - si le buffer intégré au canal n'est pas plein alors l'envoi de message est dissocié de la réception et non bloquant
    - la réception est bloquante si le buffer est vide
  - Allocation mémoire
- ```
can1 = make ( chan int, 5 )
```

# Canaux

---

## □ Exemples

### □ Canal synchrone (sans buffer)

#### □ Création/allocation

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

#### □ Exemple : interblocage

```
canSynch  <- 0  
<- canSynch
```

### □ Canal asynchrone (avec buffer)

#### □ Création /allocation

```
canASynch := make ( chan int, 5 )
```

#### □ Exemple : pas d'interblocage

```
canASynch  <- 0  
<- canASynch
```

# Canaux

---

## □ Tableaux de canaux

### □ Déclaration

```
var tabCan [10] chan int
```

### □ Initialisation du tableau et des cases

```
for i := range tabCan {  
    tabCan [i] = make ( chan int )  
}
```

### □ Exemple d'utilisation séquentielle en envoi

```
for i := range tabCan {  
    tabCan [ i ] <- dataOut [ i ]  
}
```

### □ Exemple d'utilisation séquentielle en réception

```
for i := range tabCan {  
    dataIn [ i ] = <- tabCan [ i ]  
}
```

# Sélection

---

- Attente sur plusieurs opérations de communication
- Syntaxe :

```
select {  
  case canal1 <- x:  
    x ++  
  case y = <- canal2:  
    ...  
  case <- canal3:  
    ...  
}
```

- Si une seule opération est faisable, elle est exécutée, ainsi que les instructions associées
- Si plusieurs opération sont faisables, exécution « au hasard » (introduction du non-déterminisme pour éviter les famines)
- Sinon le *select* reste bloqué en attente d'une opération faisable

# Fonctions

---

## □ Syntaxe

- mot clé **func**
- **identifiant** (nom de la fonction)
- **paramètres**
  - fusion de types pour paramètres consécutifs de même type
    - x int, y int => x, y int
  - Passage par référence ou par valeur (comme en langage C), usage de & et \*
  - Si pas de paramètre : parenthèses vides obligatoires
- **valeurs de retours**
  - 0 ou plusieurs (séparées par des virgules avec parenthèses)
  - Récupération des valeurs de retour par affectation multiple
  - si pas de valeurs de retour : rien à mettre
- Mot clé return

## □ Exemple

```
func affichage ( nombre int ) ( int , string ) {  
    fmt.Println ( nombre )  
    fmt.Println ( "fin  de la fonction" )  
    return 1 , " fini "  
}  
x,s := affichage(3)
```

# Fonctions

---

## □ Mot clé *defer* :

- Permet d'exécuter le code après le mot clé juste avant le retour de la fonction
- Le code est interprété au niveau du *defer* mais son appel ne se fait qu'à la fin de la fonction.

## □ Exemple

```
func affichage ( nombre int ) ( int , string ) {  
    defer fmt.Println ( " fin  de la fonction" )  
    fmt.Println ( nombre )  
    return 1 , " fini "  
}
```

# Fonctions

---

## □ Fonction anonyme

- Fonction sans identifiant, déclarée et appelée dans un seul bloc

- Exemple

```
func (x int) (int) { // fonction anonyme
    . . .
    return x+10
} (3)
```

## □ Différence pour la visibilité des variables

- fonction anonyme : les variables déclarées en dehors de la fonction sont accessibles dans la fonction

- fonction non anonyme : les variables déclarées en dehors de la fonction ne sont pas accessibles, il faut utiliser les paramètres

# Méthodes

---

## □ Syntaxe

- **func** *receveur* *identifiant* *paramètres* *valeurs de retours*
- *receveur* : nom et type (un seul)

## □ Exemple

```
type MyInt int
func (nombre myInt) affichage () ( int , string ) {
    fmt.Println ( nombre )
    fmt.Println ( "fin de la fonction" )
    return 1 , " fini "
}
var n MyInt = 3
x,s := n.affichage
```

- Le terme « méthode » est utilisé pour distinguer ce cas de fonction utilisée avec un receveur
  - Utile par exemple avec des structures
  - Rappelle les « méthodes » de la programmation orientée objet
  - mais attention : Go n'est pas un LOO



# Goroutine

---

- Goroutine :
  - morceau de code indépendant que l'on peut assimiler à un thread
  - les goroutines d'un processus partagent le même espace d'adressage (processus poids léger)
- Le code d'une goroutine est obligatoirement déclaré dans une fonction
  - fonction non anonyme : déclarée par ailleurs, appel de fonction
  - fonction anonyme : déclaration et appel dans un même bloc de code
- 3 états possibles pour une goroutine:
  - En cours d'exécution
  - En attente de communication (bloquée)
  - Terminée
- Différence entre création et exécution
  - lorsqu'une goroutine est créée, elle n'est pas forcément exécutée immédiatement
  - au contraire, le modèle d'exécution du code d'une fonction est d'exécuter toutes les instructions tant qu'elle n'est pas bloquée. Toutes les goroutines sont créées mais on ne peut pas faire d'hypothèse sur leur moment d'exécution.

# Goroutine

---

□ Mot clé *go*, suivi de l'appel de fonction

□ Exemple :

```
func main ( ) {  
    . . .  
    var a int  
    go f ( a )  
    . . .  
    go func ( ) { // fonction anonyme  
        . . .  
    } ( )  
}
```

# Goroutine

---

- ❑ Cas des lancements de goroutine dans une boucle avec fonction anonyme

- ❑ Exemple

```
func main ( ) {  
    . . .  
    for i := 0 ; i < 5 ; i ++{  
        go func() {  
            x :=i  
        } ( )  
    }  
}
```

- ❑ Dans ce cas, x vaut 5 dans tous les fonctions

- ❑ le go va créer les goroutines, mais leur exécution n'a lieu qu'après que la boucle *for* soit terminée.

- ❑ Pour régler le problème : utilisation d'un paramètre pour garder la valeur de i en mémoire

```
func main ( ) {  
    . . .  
    for i := 0 ; i < 5 ; i ++{  
        go func ( i int ) {  
            x :=i  
        } ( i )  
    }  
}
```

# Synchronisations

---

- Nécessité de synchroniser les goroutines avec la fonction les ayant créées
  - Les goroutines sont lancées de façon asynchrone
  - La fonction origine peut se terminer avant la fin des goroutines
  - Dans ce cas, les goroutines sont également stoppées
  - Dans le cas du `main`, risque de fin du programme avant la fin de tous les traitements
- Deux méthodes pour la synchronisation
  - En utilisant des canaux synchrones
  - En utilisant les fonctions du paquet *sync*

# Synchronisations

---

- Synchronisation par canal (synchrone !)

- Avec une goroutine

```
c := make (chan int)
go func ( ) {
    . . .
    c <- 1
} ( )
<- c
```

- Avec plusieurs goroutines :

- Mettre une boucle de lecture

- Avec un seul canal

- ou plusieurs canaux

- ou un tableau de canaux

# Synchronisations

---

- ❑ Synchronisation avec les fonctions du package *sync*
  - ❑ création d'un *WaitGroup*
  - ❑ définition du nombre de « jetons » d'attente
  - ❑ dans chaque goroutine : libération d'un « jeton » en fin (utiliser *defer*)
  - ❑ dans la fonction origine : blocage en attente de la libération de tous les jetons

- ❑ Exemple avec fonction anonyme ou non :

```
var wg sync.WaitGroup
&wg.Add(1)
go func ( ) {
    ...
    defer &wg.Done ( )
} ( )
&wg.Add(1)
go uneFonction (&wg) //le wg.Done est interne
&wg.Wait ( )
```

# Structuration des programmes

---

- Un programme est composé de paquets (packages)
- Les programmes commencent l'exécution dans le paquet main
- On importe les paquets nécessaires en début de programme
- Exemple

```
package main
import "fmt"
func main() {
    fmt.Println("Hello world")
}
```

# Paquets

---

- Un package est défini par
  - des structures de données
  - des fonctions
- Les fonctions peuvent être
  - publiques (exportés)
    - callable en externe
    - identifiant commençant par une majuscule
  - privées
    - appelables seulement en interne dans le paquet
    - identifiant commençant par une minuscule
- Pour utiliser un package

```
import (  
    "fmt"  
    "time"  
)
```



# Utilitaires

---

- package fmt
  - `fmt.Println("Hello")`
  - `fmt.Println("Le résultat est", res)`
  - `fmt.Println("Valeurs du tableau", tab)`
  - `fmt.Printf("%c %d\n", a,b)`
  - `fmt.Print(valeur)`
- package time
  - `start = time.Now()`
  - `time.Sleep (10 * time.Second)`
  - `latence = time.Since (start)`
- package math/rand
  - `var r rand.Source = rand.New(rand.newSource(seed))`
  - `r.Intn(10)`
    - retourne la même suite de nombres

# Méthodologie de conception pour le cours d'ASD

---

- Canaux
  - Usage de canaux synchrone
  - Tableaux de canaux uniformes
- Threads (“Processus”)
  - définis par des fonctions
  - créés par des goroutines
- Composition hiérarchique de processus
  - Processus structurants
  - Processus terminaux effectuant une action