



Développement Go

Introduction au langage Go

Sommaire

1. Qui suis-je ?
2. Objectifs
3. Déroulé du cours et notation
4. Introduction
5. Structures de contrôle et gestion des erreurs
6. Fonctions, packages et la bibliothèque standard de Go
7. Développement Web en Go
8. Concurrence en Go
9. Fin - Conclusion
10. Ressources

Qui suis-je ?

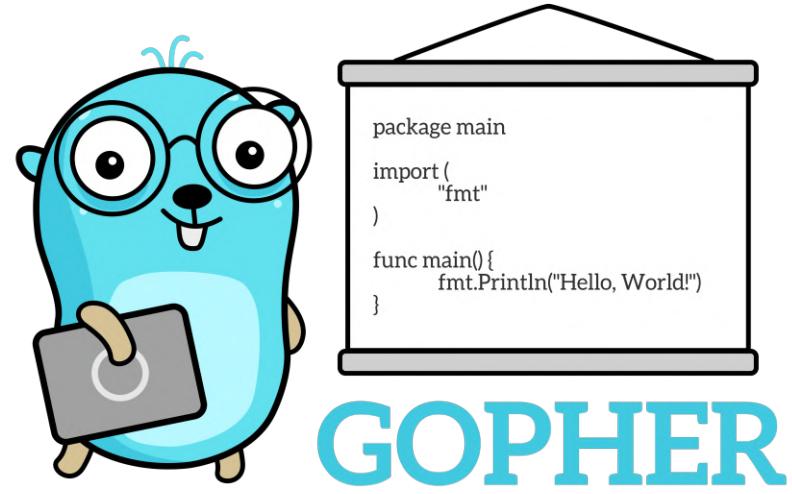
Noé Larrieu-Lacoste

- 27 ans
- Ancien élève de l'ESGI
 - Promo 2023 Architecture Logicielle 🏆
- Ingénieur DevOps chez Scaleway 🎯
- Langage préféré : Go 🐹
- Fan de Docker 🚢 et de Kubernetes ⚙️
- **Contact :**
 - noelarrieulacoste@yahoo.fr
 - linkedin.com/in/noelarrieulacoste



Objectifs

- Comprendre la syntaxe du langage Go
- Connaître les avantages et inconvénients du langage
 - Comprendre les cas d'utilisation du langage
- Savoir utiliser les outils de développement Go
- Savoir développer une application Go
- Utiliser des librairies Go telles que :
 - http
 - os
 - log
 - json
 - etc.

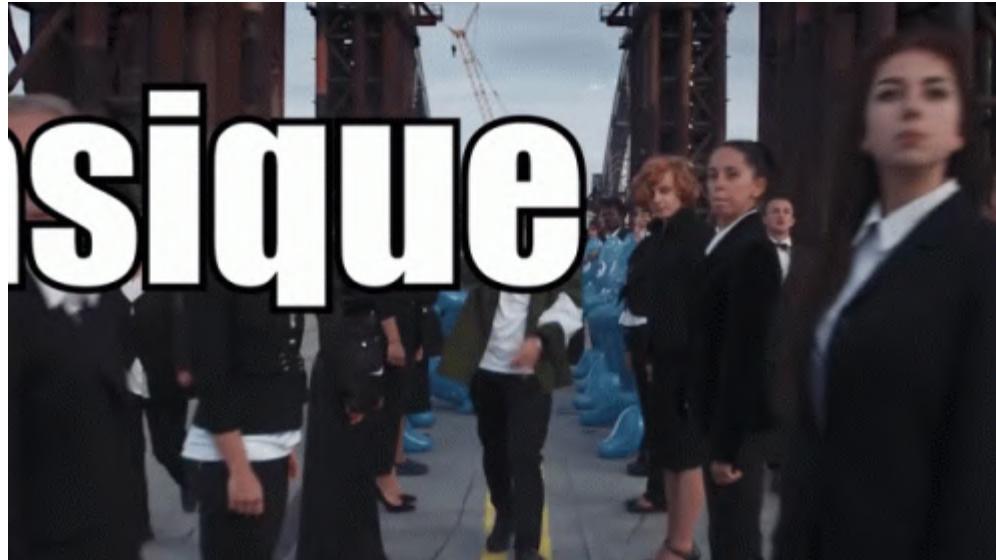


Déroulé du cours et notation

- **~14h de cours, 1 TP de 1h30, 1 projet**
- Présentation du langage Go
- Mise en place de l'environnement de développement
- Structures de contrôle et gestion des erreurs
- Fonctions, packages et bibliothèque standard (utilisation de `log`)
 - Manipulation de fichiers (utilisation de `os` et `io`)
- Structures de données complexes et bibliothèque `json`
- Développement Web (utilisation de `http` et `database/sql`)
- **Bonus** : Concurrence
- **TP noté** : Développement d'une application Go
 - 1h30 de TP, à rendre sur MyGES
- **Projet** : Développement d'une application Go
 - 1 mois pour le réaliser, à rendre sur MyGES
 - Soutenance de 15 minutes

Ne t'inquiète pas, ça va bien se passer

On va commencer par les bases



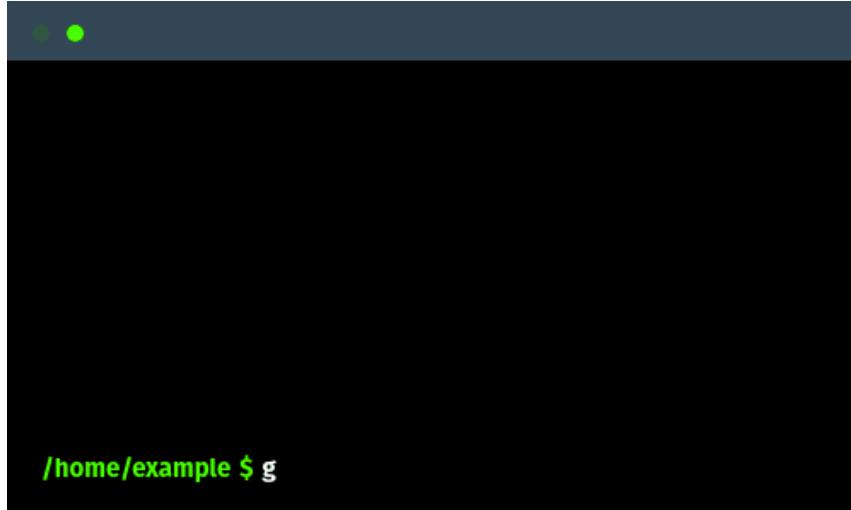
Mais d'abord, un petit test de positionnement...

Installation

<https://go.dev/doc/install>

Installation du SDK Go

<https://go.dev/doc/install>



/home/example \$ g

Documentation > Download and install

Download and install

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- [Managing Go installations](#) -- How to install multiple versions and uninstall.
- [Installing Go from source](#) -- How to check out the sources, build them on your own machine, and run them.

[Download \(1.20.7\)](#)

Don't see your operating system here? Try one of the [other downloads](#).

Go installation

Select the tab for your computer's operating system below, then follow its installation instructions.

Linux Mac Windows

1. Remove any previous Go installation by deleting the /usr/local/go folder (if it exists), then extract the archive you just downloaded into /usr/local, creating a fresh Go tree in /usr/local/go:

```
$ rm -rf /usr/local/go && tar -C /usr/local -xzf go1.20.7.linux-amd64.tar.gz
```

Installation de l'IDE

Visual Studio Code



<https://code.visualstudio.com/download>

<https://code.visualstudio.com/docs/languages/go>

GoLand



<https://www.jetbrains.com/fr-fr/go>

<https://www.jetbrains.com/fr-fr/community/education/#students>

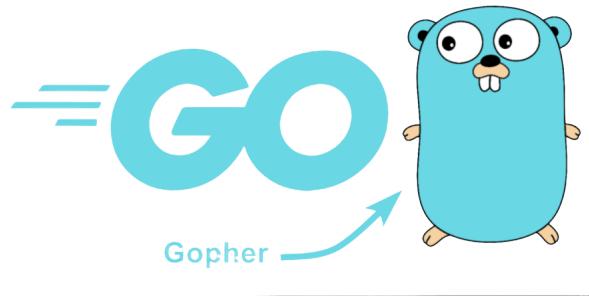
Introduction

Présentation de Go

- 2007 : Développement de Go par Ken Thompson, Rob Pike et Robert Griesemer chez Google
- Trois objectifs :
 - Simplicité d'utilisation
 - Efficacité d'exécution
 - Compilation rapide
- Novembre 2009 : Le langage est publié par Google



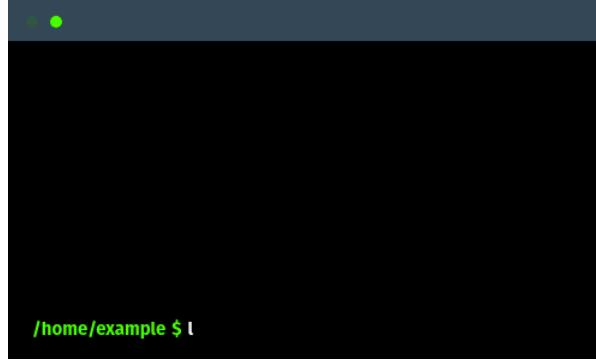
Ceci est un ... Gopher



<https://fr.wikipedia.org/wiki/Geomysidae>

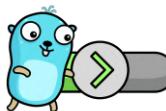
Quelques spécificités

- Langage de programmation compilé
- Fortement typé
- Une gestion de la mémoire automatique
- Compilation très rapide
- Un seul fichier binaire



Une syntaxe similaire au C

```
int main() {
    int x;
    printf("Hello, World!\n");
    x = 5;
    if (x > 6) {
        printf("x is greater than 6: %d\n", x);
    } else {
        printf("x is less than 6: %d\n", x);
    }
    return 0;
}
```



```
func main() {
    var x int
    fmt.Printf("Hello, World!\n")
    x = 5
    if x > 6 {
        fmt.Printf("x is greater than 6: %d\n", x)
    } else {
        fmt.Printf("x is less than 6: %d\n", x)
    }
}
```

Les avantages de Go

- **Garbage Collector** pour une gestion automatique de la mémoire.
- **Support multi-cœurs** pour une utilisation optimale des processeurs.
- **Facilite la maintenance du code.**
- **Typage dynamique** et prise en charge des **mappages clé-valeur** (*dictionnaires*).
- **Bibliothèque standard** complète pour des programmes autonomes.
- **Compilation rapide** et système de build simplifié.
- **Compatibilité avec plusieurs systèmes d'exploitation** (*Windows, Linux, macOS, Android, iOS*).



Domaines d'utilisation de Go

-  Serveurs et APIs.
-  Programmation réseau.
-  Applications microservices.
-  Développement Web avec des frameworks tels que Gin et Echo.
-  Outils de ligne de commande et utilitaires système.

Entreprises / Technologies en Go

-  **Google:** Go a été développé par Google et est largement utilisé en interne.
-  **Docker:** Son moteur principal est écrit en Go.
-  **Kubernetes:** Go est utilisé pour ses principaux composants.
-  **Netflix:** Des parties de l'infrastructure sont en Go.
-  **Uber:** Des services backend d'Uber sont en Go.
-  **Twitch:** Certaines parties de Twitch sont développées en Go.
-  **Prometheus:** Un système de monitoring open-source en Go.
-  **Hugo:** Un générateur de sites statiques en Go.
-  **Golang.org:** Le site officiel de Go est développé en Go.

Pour résumer, le Go c'est...

- Langage **statique et compilé** (Java, C, C++,...)
- Syntaxe proche du langage C
- **Garbage Collector** → Gestion de la mémoire automatique 
- Multi-CPU et parallelisme dans le langage (sans avoir recours à des librairies externes)
- Un seul binaire, presque aucune dépendance !
- **Multi-plateforme** (Windows, macOS, Linux, arm,...)

Premier programme



Hello, World !

Fichier `main.go`

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, World!")
7 }
```

- **Package** `main` : Point de départ de l'exécution.
- **Import** `fmt` : Pour le formatage d'entrée/sortie.
- **Fonction** `main` : Point d'entrée du programme.
- `fmt.Println` : Affiche du texte suivi d'un saut de ligne.

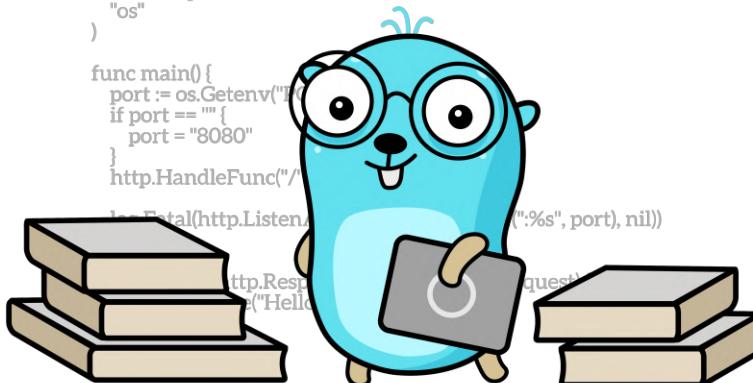
Compiler et exécuter

```
$ go run main.go
Hello, World!
```

```
$ go build main.go
$ ls
main main.go
$ ./main
Hello, World!
```

Types de données de base en Go

```
// This server can run on App Engine.  
package main  
  
import (  
    "fmt"  
    "log"  
    "net/http"  
    "os"  
)  
  
func main() {  
    port := os.Getenv("PORT")  
    if port == "" {  
        port = "8080"  
    }  
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {  
        log.Printf("Hello, %s! Port: %s", r.RemoteAddr, port)  
        w.Write([]byte("Hello, "+r.RemoteAddr+", "+port))  
    })  
    log.Fatal(http.ListenAndServe(":"+port, nil))  
}
```



go learn()

Types de données de base en Go

- **Type de données** : spécifie la taille et le type de valeurs des variables.
- Go est **typé statiquement** : une fois le type défini, la variable ne peut contenir que des données de ce type.
- Go propose trois catégories de **types de données de base** :
 - `bool` : pour les valeurs booléennes (vrai ou faux).
 - **Types numériques** : englobant les entiers, les nombres à virgule flottante, et les nombres complexes.
 - `int` (alias de `int32`), `byte` (alias de `uint8`), `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `float32`, `float64`, `complex64`, `complex128`, `rune` (alias de `int32`) sont inclus dans cette catégorie.
 - `string` : utilisé pour représenter une séquence de caractères.

Exemple

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var a bool = true      // Boolean
7     var b int = 5         // Integer
8     var c float32 = 3.14 // Floating point number
9     var d string = "Hi!" // String
10
11    fmt.Println("Boolean: ", a)
12    fmt.Println("Integer: ", b)
13    fmt.Println("Float:   ", c)
14    fmt.Println("String:  ", d)
15 }
```

```
Boolean: true
Integer: 5
Float: 3.14
String: Hi!
```

Déclaration de variables

En Go, les variables sont en camelCase et peuvent être déclarées de plusieurs façons :

- Déclaration explicite :

```
`var <nom> <type> = <valeur>`
```

```
var a int = 5  
var b string  
b = "Hello"
```

- Déclaration implicite :

```
`<nom> := <valeur>`
```

```
a := 5 // a est de type int  
b := "Hello" // b est de type string
```

- Déclaration groupée :

```
var x, y int = 5, 6  
var a, b = int  
a = 5  
b = 6
```

- Déclaration multiple :

```
var (  
    x int = 5  
    y int = 6  
)
```

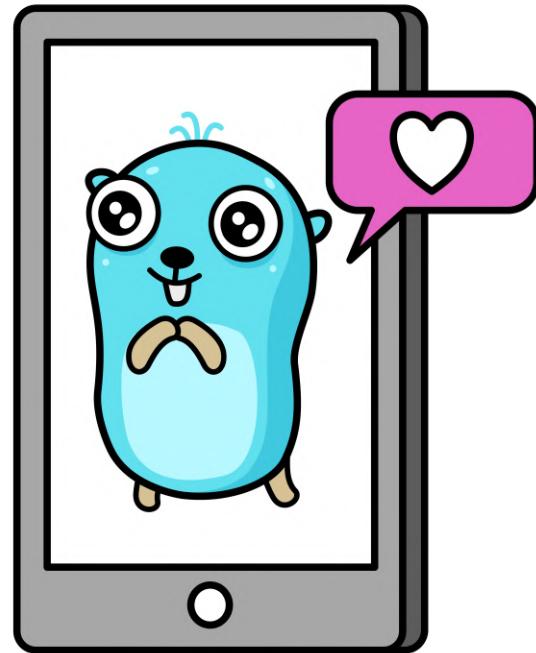
- Déclaration de constantes :

```
const x int = 5  
const pi = 3.14 // pi est de type float64
```

- Cast :

```
var a int = 42  
var b float64 = float64(a) + 0.42
```

Opérateurs en Go



go mobile()

Opérateurs arithmétiques

■ Opérateur `+` :

```
var a = 15 + 25 // a = 40  
var b = a + 5 // b = 45  
var s = "Hello"  
s = s + " World!" // s = "Hello World!"
```

■ Opérateur `*` :

```
var x = 5  
var y = 6  
var res = x * y // res = 30
```

■ Autres opérateurs :

```
var x = 5  
x++ // x = 6  
x-- // x = 5
```

■ Opérateur `-` :

```
var a = 15 - 25 // a = -10  
var b = float64(a) - 5.5 // b = -15.5
```

■ Opérateurs `/` et `%` :

```
var x = 10 / 3 // x = 3  
var y = 10.0 / 4 // y = 2.5  
  
var z = 10 % 3 // z = 1
```

```
var x = 5  
x += 5 // x = 10  
x -= 4 // x = 6  
x *= 2 // x = 12  
x /= 3 // x = 4  
x %= 3 // x = 1
```

Opérateurs de comparaison et logiques

■ Opérateurs de comparaison :

```
var x, y = 5, 6

var res = x == y // Egale à -> res = false
res = x != y // Différent de -> res = true

res = x < y // Inferieur à -> res = true
res = x <= y // Inferieur ou égal à -> res = true

res = x > y // Supérieur à -> res = false
res = x >= y // Supérieur ou égal à -> res = false
```

■ Opérateurs logiques :

```
var x = true
var y = false

var res = x && y // ET -> res = false

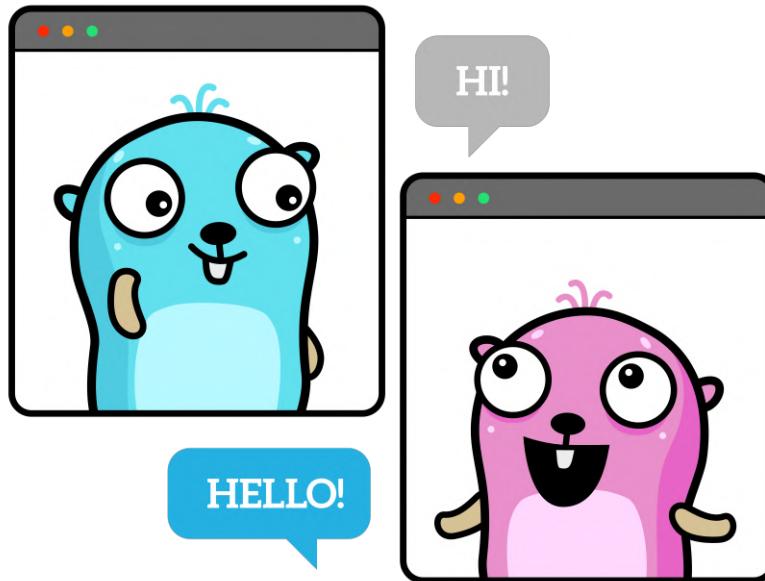
res = x || y // OU -> res = true

res = !x // NON -> res = false
```

Que fait ce programme ?

```
1 func main() {
2     a, b := 5, 10
3     fmt.Println(a < b, a > b, a == b) // ?, ?, ?
4     fmt.Println(a < b && a > b, a < b || a > b) // ?, ?
5     fmt.Println(!(a < b && a > b), !(a < b || a > b)) // ?, ?
6 }
```

Utilisation de `fmt` en Go



Utilisation de `fmt` en Go

Introduction à `fmt`

Le package `fmt` (abréviation de format) est l'un des packages les plus importants en Go.

- Il fournit des fonctions pour formater et afficher du texte.
- `fmt` est souvent utilisé pour l'entrée et la sortie standard (E/S) de données.

Liens utiles :

- <https://golang.org/pkg/fmt/>
- <https://gobyexample.com/string-formatting>
- <https://cheatography.com/fenistil/cheat-sheets/go-fmt-formatting/>
- <https://yourbasic.org/golang/fmt printf reference cheat sheet/>

Utilisation de `fmt` en Go

Affichage de texte

Pour afficher du texte en Go, la fonction `fmt.Println()` est la plus couramment utilisée.

Elle affiche une ligne de texte suivie d'un saut de ligne.

Exemple :

```
package main

import "fmt"

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

Utilisation de `fmt` en Go

Formatage de texte avec `fmt.Sprintf()`

Le package `fmt` en Go offre de puissantes fonctionnalités de formatage de texte pour afficher des données de manière structurée.

- La fonction `fmt.Sprintf` est l'une des fonctions les plus couramment utilisées dans le package fmt.
- Elle permet de formater et d'afficher du texte en utilisant des spécificateurs de format.

Exemple :

```
fmt.Sprintf("Bonjour, %s ! Vous avez %d ans.\n", "Alice", 30)
```

- `%s` pour insérer une chaîne de caractères, et `%d` pour insérer un nombre entier.
- Les spécificateurs de format commencent par `%` et sont suivis du type de donnée.
- `\n` est un caractère spécial qui indique un saut de ligne.

Utilisation de `fmt` en Go

Les Principaux Spécificateurs de Format

Voici quelques spécificateurs de format couramment utilisés avec `fmt.Printf` :

- `'%d'` : Affiche un nombre entier.
- `'%f'` : Affiche un nombre à virgule flottante.
- `'%s'` : Affiche une chaîne de caractères.
- `'%t'` : Affiche un booléen.
- `'%c'` : Affiche un caractère Unicode.
- `'%v'` : Affiche la valeur de la variable. Il choisit le format approprié en fonction du type de données

Exemple :

```
nombre := 42
texte := "exemple"
boolValue := true

fmt.Printf("%d, %f, %s, %t, %c\n", nombre, 3.1415, texte, boolValue, 'A') // 42, 3.141500, exemple, true, A
fmt.Printf("%v, %v, %v, %v, %v\n", nombre, 3.1415, texte, boolValue, 'A') // 42, 3.1415, exemple, true, 65
```

Utilisation de `fmt` en Go

Alignment et largeur de champ

On peut contrôler l'alignment et la largeur des valeurs affichées

Par exemple, `%5d` affiche un nombre entier sur 5 caractères, aligné à droite :

```
fmt.Printf("Nombre : %5d\n", 42) // Nombre :    42
```

Précision pour les nombres à virgule flottante

Pour les nombres à virgule flottante, on peut spécifier la précision en utilisant `%.2f` pour afficher deux décimales :

```
pi := 3.14159265359
fmt.Printf("Pi : %.2f\n", pi) // Pi : 3.14
```

Arguments de la ligne de commande



Arguments de la ligne de commande

Introduction

Les arguments de ligne de commande en Go sont utilisés pour personnaliser le comportement d'un programme lors de son exécution.

Exemple :

```
$ go run main.go 1 2 3
```

Dans cet exemple, `1`, `2` et `3` sont des arguments de la ligne de commande.

Arguments de la ligne de commande

Lire les arguments de la ligne de commande

- Utilisez la variable `os.Args` pour récupérer les arguments en ligne de commande.
- Elle contient une liste de chaînes de caractères représentant les arguments passés au programme.

Exemple :

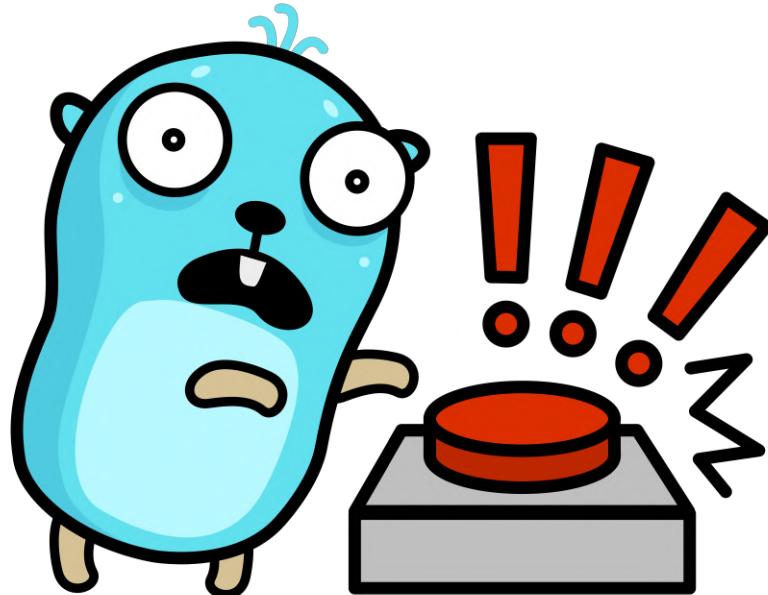
```
func main() {
    // Récupération des arguments en ligne de commande
    arguments := os.Args

    // Affichage des arguments
    fmt.Println("Arguments en ligne de commande :")
    fmt.Println(arguments)

    // Affichage du premier argument
    fmt.Println("Premier argument :", arguments[0])

    // Affichage du nombre d'arguments
    fmt.Println("Nombre d'arguments :", len(arguments))
}
```

Exercices



Ressources

Avantages de Go

- yourbasic.org/advantages-over-java-python
- yourbasic.org/go-vs-java

Syntaxe et Formatage

- yourbasic.org/fmt-printf-reference-cheat-sheet
- gobyexample.com/string-formatting

Types et Système de Types

- go101.org/article/type-system-overview.html

Développement en Go

- gobyexample.com/command-line-arguments
- gobyexample.com/command-line-flags
- devopssec.fr/article/configurer-environnement-golang
- devopssec.fr/article/variables-golang

Structures de contrôle et gestion des erreurs



Conditions

Condition de base

```
age := 19
if age >= 18 {
    fmt.Println("Vous êtes majeur")
}
```

```
age := 19
if age >= 18 && age < 21 {
    fmt.Println("Vous êtes majeur " +
    "mais pas encore autorisé à boire aux USA")
}
```

Condition alternative

```
var number = 9
if number < 0 {
    fmt.Println(number, "est négatif")
} else if number < 10 {
    fmt.Println(number, "a 1 chiffre")
} else {
    fmt.Println(number, "a plusieurs chiffres")
}
```

Condition avec initialisation

```
age := 16
if minAge := 18; age >= minAge {
    fmt.Println("Vous avez l'âge requis")
} else {
    fmt.Println("Vous êtes trop jeune")
    fmt.Println("Venez dans", minAge-age, "ans")
}
```

Switch

Le switch est une structure de contrôle permettant de diriger l'exécution du programme vers des branches de code différentes en fonction de la valeur d'une expression donnée.

Switch basique

```
package main

import "fmt"

func main() {
    i := 2
    fmt.Println("Le chiffre ", i, " s'écrit ")

    switch i { // Début du switch case
        case 1: // Si 'i' est égal à 1, affiche "un"
            fmt.Println("un")
        case 2: // Si 'i' est égal à 2, affiche "deux"
            fmt.Println("deux")
        case 3: // Si 'i' est égal à 3, affiche "trois"
            fmt.Println("trois")
    }
    // Fin du switch case
}
```

Switch avec plusieurs valeurs

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var dayOfWeek = time.Now().Weekday()
    fmt.Println("Jour de la semaine :", dayOfWeek)

    switch dayOfWeek {
        case time.Saturday, time.Sunday:
            fmt.Println("C'est le week-end !")
        default:
            fmt.Println("C'est la semaine !")
    }
}
```

Switch

Les cas du switch sont évalués de haut en bas, s'arrêtant lorsqu'un cas réussit.

```
func main() {
    fmt.Println("C'est quand le samedi?")
    today := time.Now().Weekday()
    switch time.Saturday {
        case today + 0:
            fmt.Println("Aujourd'hui.")
        case today + 1:
            fmt.Println("Demain.")
        case today + 2:
            fmt.Println("Dans deux jours.")
        default:
            fmt.Println("C'est trop loin.")
    }
}
```

Si aucune expression n'est indiquée dans le switch, les différents cas sont évalués en se basant sur la véracité des expressions associées.

```
func main() {
    currentHour := time.Now().Hour()
    fmt.Printf("Il est %dh\n", currentHour)

    switch {
        case currentHour < 12:
            fmt.Println("C'est le matin !")
        case currentHour >= 12 && currentHour < 18:
            fmt.Println("C'est l'après-midi !")
        default:
            fmt.Println("C'est le soir !")
    }
}
```

Switch

Que fait ce code ?

```
func main() {
    var number = 42

    switch {
    case number < 0:
        fmt.Println("Number is negative")
    case number > 0:
        fmt.Println("Number is positive")
    case number == 42:
        fmt.Println("Number is 42")
    default:
        fmt.Println("Number is 0")
    }
}
```

```
$ go run main.go
Number is positive
```

Boucle while

En Go, il n'y a pas de boucle `while` 😠

Boucle for



Boucle for

Une boucle "for" (ou "for loop") est une structure de contrôle qui permet de répéter l'exécution d'un bloc de code en spécifiant le nombre de répétitions ou en définissant une condition de sortie.

Les boucles "for" sont utiles pour automatiser des tâches répétitives.

```
for initialisation; condition; itération {
    // Bloc de code à répéter
}
```

```
sum := 0
for i := 1; i < 5; i++ {
    sum += i
}
fmt.Println(sum)
```

```
$ go run main.go
10
```

Boucle for

Boucle for avec une seule condition

```
for condition {  
    // Bloc de code à répéter  
}
```

```
n := 1  
for n < 5 {  
    n *= 2  
}  
fmt.Println(n) // 8 (1*2*2*2)
```

Boucle for infinie

```
for {  
    // Bloc de code à répéter  
}
```

```
for {  
    fmt.Println("Je suis une boucle infinie")  
}
```

Boucle for

Break

```
for i := 1; i < 10; i++ {  
    if i > 5 {  
        break  
    }  
    fmt.Println(i)  
}
```

Continue

```
for i := 1; i < 10; i++ {  
    if i%2 == 0 {  
        continue  
    }  
    fmt.Println(i)  
}
```

```
$ go run main.go  
1  
2  
3  
4  
5
```

```
$ go run main.go  
1  
3  
5  
7  
9
```

Boucle for

Range

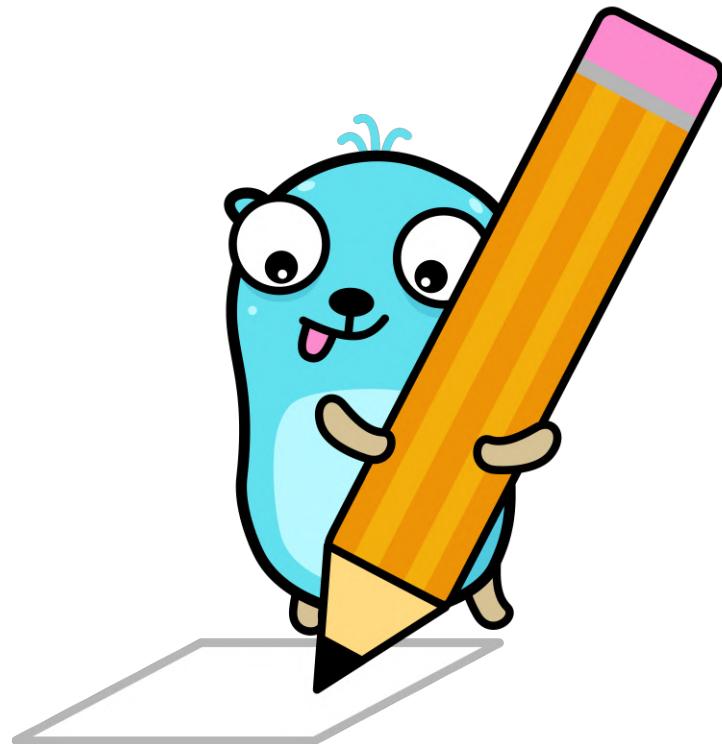
Range existait déjà avant, mais c'est depuis Go 1.22 que l'on peut utiliser `range` pour itérer sur une valeur numérique.

```
package main

import "fmt"

func main() {
    for i := range 10 {
        fmt.Println(10 - i)
    }
    fmt.Println("go1.22 has lift-off!")
}
```

Exercices

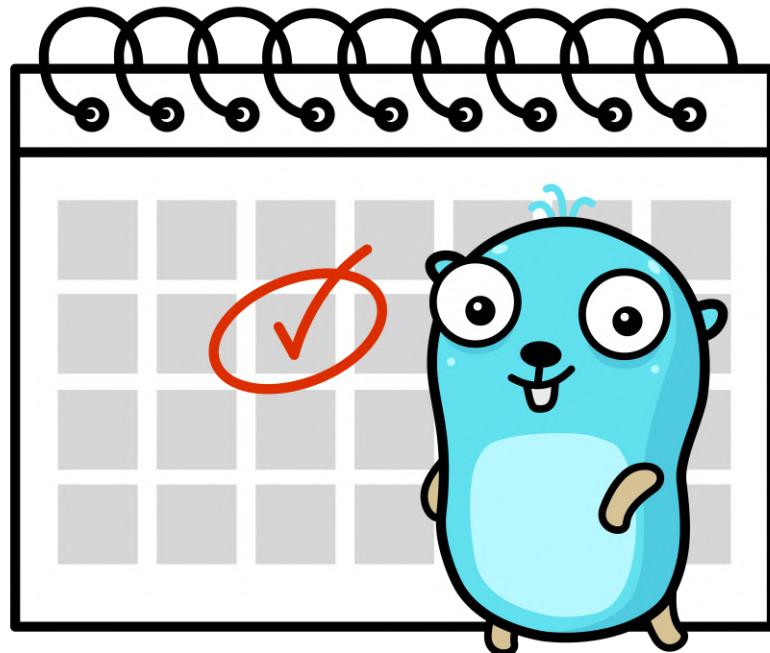


Ressources

Structures de Contrôle

- [if-else](#)
- [switch](#)
- [for](#)
- [if-else \(yourbasic.org\)](#)
- [switch \(yourbasic.org\)](#)
- [for \(yourbasic.org\)](#)
- [Conditions en Go \(devopssec.fr\)](#)
- [Boucles en Go \(devopssec.fr\)](#)

Tableaux et tranches



Tableaux et tranches

Un tableau (array) est une séquence d'éléments de taille fixe d'un type donné.

- Éléments contigus en mémoire.
- Indexation à partir de 0.
- Taille immuable. Pour l'agrandir, créez un nouveau tableau ou utilisez une autre approche.
- Éléments initialisés par défaut (ex. : 0 ou "").

Un tableau est déclaré comme suit :

```
var nomTableau [taille]type
```

Exemples :

```
var tableau [5]int
fmt.Println(tableau) // [0 0 0 0 0]

tableau[0] = 1
fmt.Println(tableau) // [1 0 0 0 0]
```

```
var tableau [5]int{1, 2, 3, 4, 5}

fmt.Println(len(tableau)) // 5
fmt.Println(tableau) // [1 2 3 4 5]
fmt.Println(tableau[3]) // 4
```

Tableaux et tranches

Une tranche (slice) est une structure de données dynamique basée sur un tableau sous-jacent (tableau à taille dynamique).

- Segments variables de tableaux en mémoire.
- Gestion de collections de données de taille variable.
- Manipulation, agrandissement, réduction aisés.

Une tranche est déclarée comme suit :

```
var nomTranche []type
```

Exemples :

```
var maTranche []int
fmt.Println(maTranche) // []
maTranche = append(maTranche, 1)
fmt.Println(maTranche) // [1]
maTranche = append(maTranche, 2, 3, 4)
fmt.Println(maTranche) // [1 2 3 4]
```

```
maTranche := []int{1, 2, 3, 4, 5}

fmt.Println(maTranche) // [1 2 3 4 5]
fmt.Println(len(maTranche)) // 5
fmt.Println(maTranche[3]) // 4
```

Tableaux et tranches

Initialisation avec `make`

`make` est couramment utilisé pour créer des tranches avec une capacité initiale spécifiée.

- Crée une tranche avec taille et capacité spécifiées.
- Capacité importante pour éviter des réallocations fréquentes.
- La tranche est initialisée avec des valeurs par défaut appropriées au type (ex. : 0, "").

Exemple :

```
maTranche := make([]int, 5, 10)
fmt.Println(maTranche) // [0 0 0 0]
fmt.Println(len(maTranche)) // 5
fmt.Println(cap(maTranche)) // 10
```

Tableaux et tranches

Capacité d'une tranche

La capacité d'une tranche représente la taille du tableau sous-jacent par rapport à la taille de la tranche.

- Une tranche peut contenir des éléments jusqu'à sa capacité maximale.
- Atteindre la capacité déclenche une réaffectation mémoire coûteuse lors de l'ajout d'éléments.
- Augmentez la capacité avec `append` si nécessaire.

```
maTranche := make([]int, 5)
fmt.Println(maTranche) // [0 0 0 0 0]
fmt.Println(len(maTranche)) // 5
fmt.Println(cap(maTranche)) // 5
maTranche = append(maTranche, 1)
fmt.Println(maTranche) // [0 0 0 0 1]
fmt.Println(len(maTranche)) // 6
fmt.Println(cap(maTranche)) // 10
```

Tableaux et tranches

Réallocation mémoire

Lorsque la capacité d'une tranche est dépassée, une nouvelle allocation de mémoire est nécessaire.

- Réaffectation possible pour une capacité accrue.
- Copie des éléments existants dans la nouvelle tranche.
- Opérations d'ajout coûteuses si réaffectation fréquente.

```
s := make([]int, 0, 3)
fmt.Printf("len=%d cap=%d, %p\n", len(s), cap(s), s) // len=0 cap=3, 0xc0000a4000

s = append(s, 1) // len=1, cap=3, 0xc0000a4000
s = append(s, 2) // len=2, cap=3, 0xc0000a4000
s = append(s, 3) // len=3, cap=3, 0xc0000a4000
s = append(s, 4) // len=? cap=?, ?

fmt.Printf("len=%d cap=%d, %p\n", len(s), cap(s), s) // len=4 cap=6, 0xc0000c0000
```

Tableaux et tranches

Trancher (slicing) une tranche

Trancher (slicer) une tranche crée une nouvelle tranche.

- Un "tranchage" génère une vue sur une partie de la tranche d'origine.
- Aucune copie de données n'est effectuée, partage la mémoire.
- Pratique pour manipuler des sections de grandes tranches.

Exemple :

```
elements := []string{"chat", "chien", "oiseau", "poisson", "souris"}  
  
sousTranche := elements[1:4] // Contient "chien", "oiseau", "poisson"  
  
sousTranche = elements[:3] // Contient "chat", "chien", "oiseau"  
  
sousTranche = elements[2:] // Contient "oiseau", "poisson", "souris"  
  
sousTranche = elements[:] // Contient "chat", "chien", "oiseau", "poisson", "souris"
```

Tableaux et tranches

Copier une tranche

Pour copier une tranche en Go, utilisez `copy`.

- Copie le contenu d'une tranche source dans une tranche de destination.
- Capacités différentes sont acceptées.
- Crée une copie indépendante, sans affecter la source.

Exemple :

```
source := []int{1, 2, 3}
destination := make([]int, len(source))

copy(destination, source)

fmt.Println(destination) // [1 2 3]
```

Tableaux et tranches

Parcourir une tranche

Parcourir une tranche avec la boucle `for`.

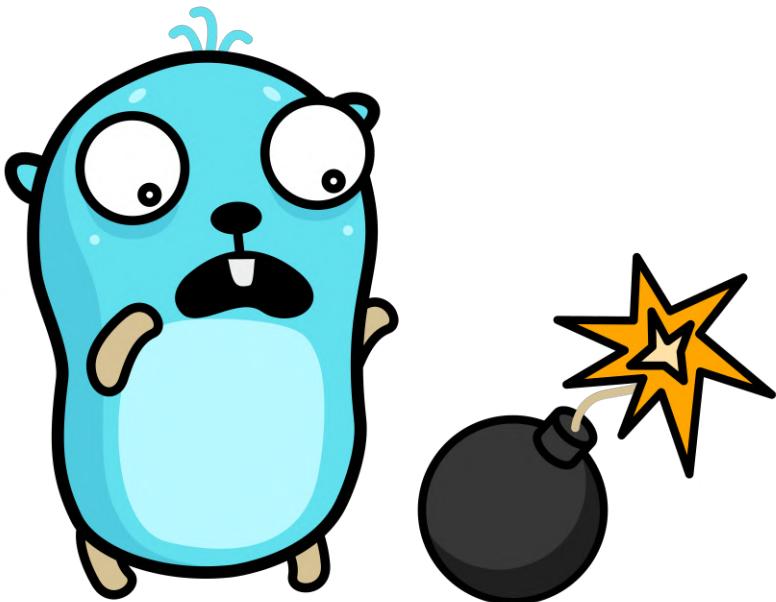
- La boucle `for` parcourt chaque élément de la tranche.
- Utilisez l'index et la valeur pour des opérations spécifiques.
- Des raccourcis comme `range` simplifient le parcours des tranches.

Exemple :

```
elements := []string{"chat", "chien", "oiseau", "poisson", "souris"}  
  
for i := 0; i < len(elements); i++ {  
    fmt.Println(i, elements[i])  
}
```

```
// Utilisation de la boucle "range"  
for i, element := range elements {  
    fmt.Println(i, element)  
}
```

Exercices



Ressources

Tutoriels sur les Tableaux (Arrays) et Tranches (Slices)

- gobyexample.com/slices
- gobyexample.com/arrays

Utilisation de la Boucle `range` en Go

- gobyexample.com/range

Comparaison et Explication des Tranches (Slices) en Go

- yourbasic.org/golang/compare-slices/
- yourbasic.org/golang/slices-explained/

Compréhension approfondie des Conteneurs en Go

- go101.org/article/container.html

Articles sur les Tableaux en Go (en français)

- devopssec.fr/article/tableaux-golang
- devopssec.fr/article/slices-tableaux-dynamiques-golang

Dictionnaires (maps)



Dictionnaires (maps)

Un dictionnaire (map) est une collection de paires clé-valeur.

- Stocke des données sous forme de paires clé-valeur avec clés uniques.
- Utiles pour représenter des relations et des associations de données.
- Dynamiques, la taille s'ajuste au besoin.

Déclaration :

```
var nomMap map[clé]valeur
```

Exemple :

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

```
ages["Alice"] = 30  
ages["Bob"] = 25
```

```
ages := map[string]int{
```

```
    "Alice": 30,  
    "Bob": 25,  
}
```

Dictionnaires (maps)

Accéder aux éléments d'un dictionnaire

Pour accéder aux valeurs dans une map, utilisez la clé correspondante.

- Accédez à une valeur en utilisant la clé entre crochets.
- En cas d'absence de clé, renvoie la valeur par défaut du type (0 pour les entiers, "" pour les chaînes, etc.).
- Vérifiez l'existence de la clé avec une deuxième valeur de retour.

Exemple :

```
ages := make(map[string]int)
ages["Alice"] = 30
ages["Bob"] = 25

ageAlice := ages["Alice"] // Renvoie 30
existe, ageCharlie := ages["Charlie"] // Renvoie 0, false (Charlie n'existe pas)
```

Dictionnaires (maps)

Supprimer un élément d'un dictionnaire

Supprimez un élément d'une map avec l'opérateur `delete`.

- Utilisez `delete` avec la clé pour supprimer un élément de la map.
- Pas d'erreur si la clé n'existe pas.
- Aucun effet si la clé n'existe pas.

Exemple :

```
ages := make(map[string]int)
ages["Alice"] = 30
ages["Bob"] = 25

delete(ages, "Alice") // Supprime la clé "Alice"

ageAlice, existe := ages["Alice"] // Renvoie 0, false (Alice n'existe pas)
fmt.Println(ages) // map[Bob:25]
```

Dictionnaires (maps)

Parcourir un dictionnaire

Vous pouvez parcourir les éléments d'une map en utilisant une boucle `for` ou la boucle `range`.

- La boucle `for` itère sur chaque paire clé-valeur de la map.
- L'instruction `range` permet d'accéder aux clés et aux valeurs individuellement.

Exemple :

```
ages := map[string]int{
    "Alice": 30,
    "Bob":   25,
}

// Parcours la map : clé, valeur
for nom, age := range ages {
    fmt.Printf("%s a %d ans\n", nom, age)
}
```

```
// Parcours la map : clé
for nom := range ages {
    fmt.Println(nom)
}
```

```
// Parcours la map : valeur
for _, age := range ages {
    fmt.Println(age)
}
```

Dictionnaires (maps)

Taille d'un dictionnaire

La taille d'une map et son allocation mémoire sont dynamiques.

- La taille représente le nombre d'éléments.
- Augmente lors de l'ajout, diminue lors de la suppression.
- Agrandissement automatique selon la taille des données.
- Obtenez la taille avec `len` .

Exemple :

```
ages := make(map[string]int)
ages["Alice"] = 30      // Taille : 1
ages["Bob"] = 25       // Taille : 2
delete(ages, "Alice") // Taille : 1 (élément supprimé)

fmt.Println(len(ages)) // 1
```

Dictionnaires (maps)

Allocation mémoire d'un dictionnaire

Les maps sont dynamiques, elles allouent automatiquement la mémoire.

- Initialement, les maps allouent peu de mémoire.
- Elles réallouent automatiquement plus de mémoire en cas d'augmentation de taille.
- Prévoir la taille est recommandé pour éviter des réallocations coûteuses.

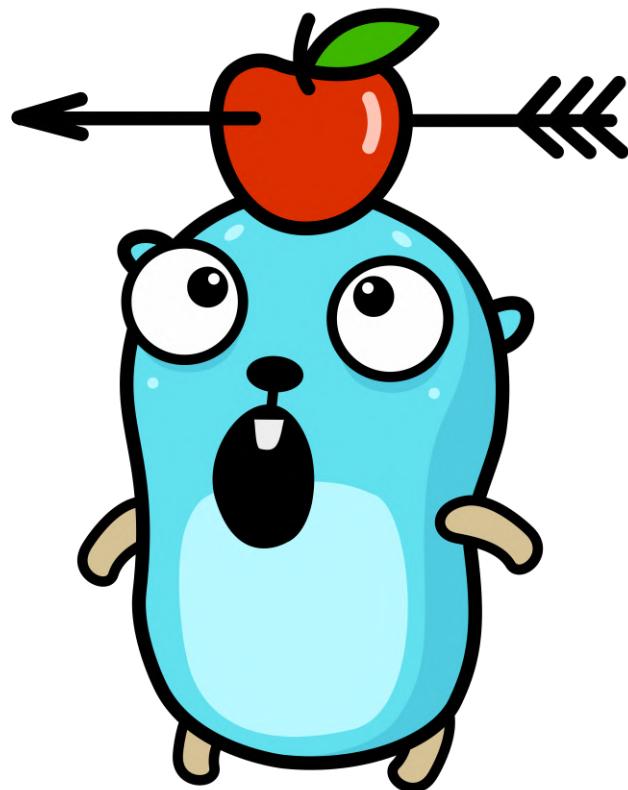
Exemple :

```
elements := make(map[string]string, 1000) // Alloue de la mémoire pour 1000 éléments
fmt.Println(len(elements)) // 0

elements["chat"] = "meow"
fmt.Println(len(elements)) // 1

elements["chien"] = "woof" // Réallocation de mémoire si nécessaire
fmt.Println(len(elements)) // 2
```

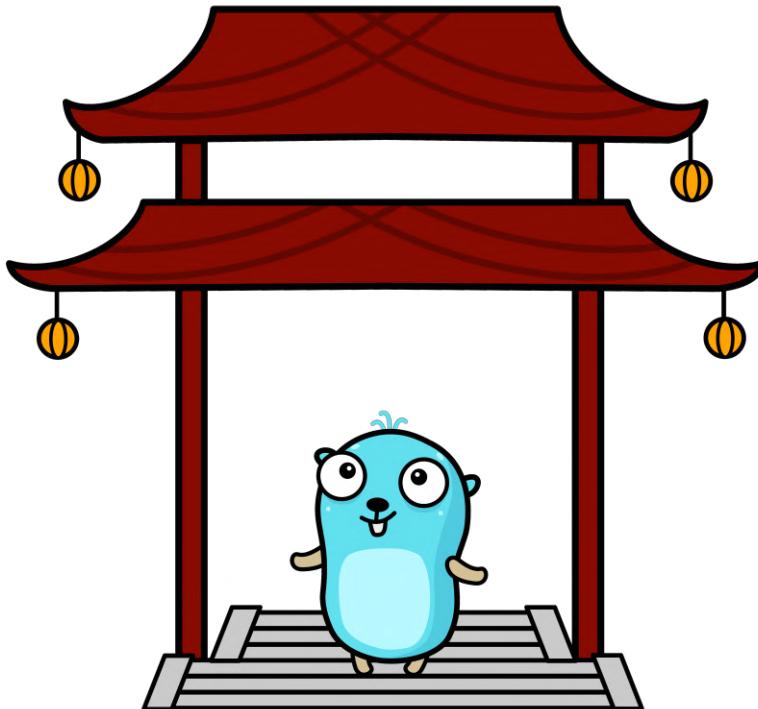
Exercices



Ressources

- gobyexample.com/maps
- devopssec.fr/article/maps-golang

Structures



Structures

Une structure (struct) est un type de données composite pour regrouper diverses valeurs de types différents.

- Représentent des objets complexes avec plusieurs propriétés.
- Chaque propriété est un champ nommé avec un type de données.
- Communément employées pour créer des types de données personnalisés.

Exemple :

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

Structures

Déclaration d'une structure

Déclarer une structure en spécifiant les noms et les types de ses champs.

- Utilisez `type` pour déclarer la structure avec un nom.
- À l'intérieur des accolades, définissez les noms et les types des champs.
- Crée un nouveau type de données pour la structure.

Exemple :

```
type Personne struct {
    nom string
    age int
    adresse string
    téléphones []string
}
```

Structures

Instanciation d'une structure

Pour créer une instance d'une structure, utilisez la déclaration de la structure suivie de valeurs pour ses champs.

```
variable := NomStructure{champ1: valeur1, champ2: valeur2, ...}  
variable := NomStructure{valeur1, valeur2, ...}
```

- Utilisez la notation littérale avec des valeurs pour chaque champ.
- L'ordre des champs est crucial, mais les noms des champs peuvent être spécifiés pour plus de clarté.

Exemple :

```
type Personne struct {  
    nom string  
    age int  
}  
alice := Personne{nom: "Alice", age: 30}  
bob := Personne{"Bob", 25}
```

Structures

Affichage d'une structure

Pour afficher une structure en Go, utilisez le package `fmt` avec différentes options de formatage.

- `'%v'` : Affiche les valeurs des champs en format simple.
- `'%+v'` : Affiche noms de champs avec leurs valeurs.
- `'%#v'` : Affiche la structure avec une syntaxe littérale.

Exemple :

```
alice := Personne{nom: "Alice", age: 30}

fmt.Printf("Format simple : %v\n", alice)      // {Alice 30}
fmt.Printf("Avec noms de champs : %+v\n", alice) // {nom:Alice age:30}
fmt.Printf("Syntaxe littérale : %#v\n", alice)   // main.Personne{nom:"Alice", age:30}
```

Structures

Accéder aux champs d'une structure

Accès aux champs d'une structure pour lecture ou modification.

- **Lecture** : `nomStructure.champ`.
- **Modification** : Utilisez la même notation pour assigner une nouvelle valeur.

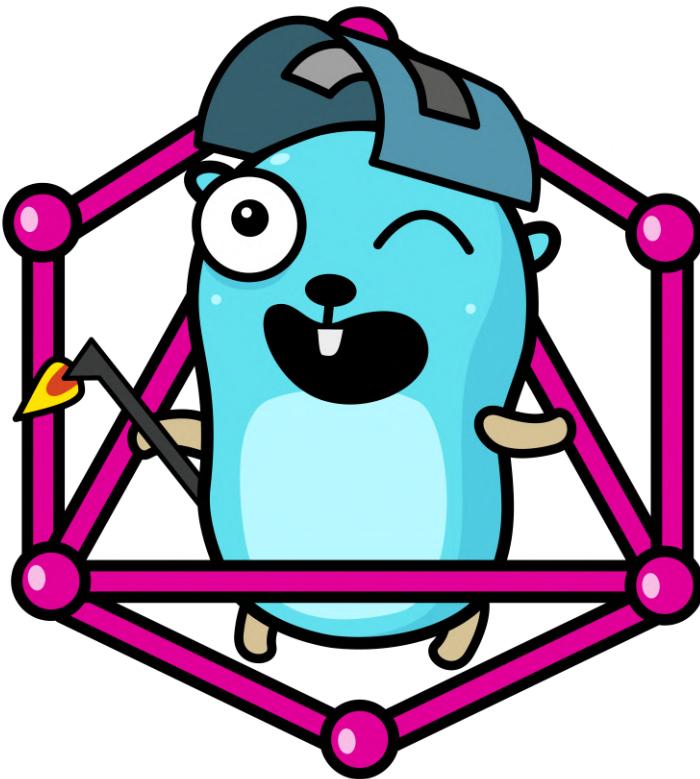
Exemple :

```
alice := Personne{nom: "Alice", age: 30}

fmt.Println(alice.nom) // Alice

alice.nom = "Alice Smith"
fmt.Println(alice.nom) // Alice Smith
```

Exercices



Ressources

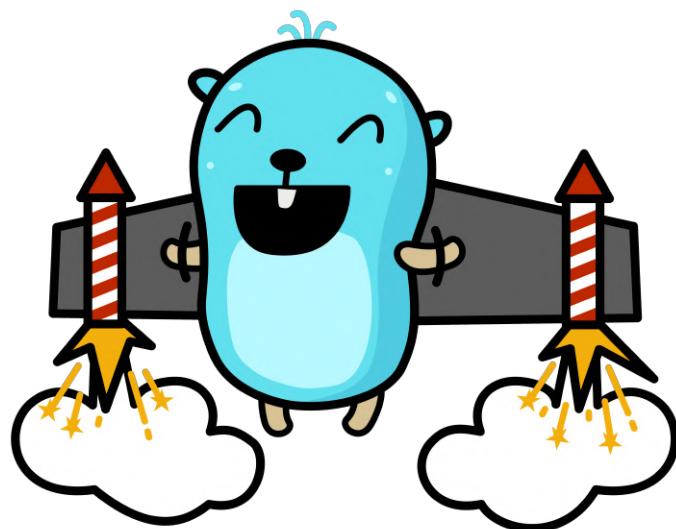
Tutoriels sur les Structures (Structs) en Go

- gobyexample.com/structs
- go101.org/article/struct.html

Article sur les Structures et Méthodes en Go (en français)

- devopssec.fr/article/structures-et-methodes-golang

Fonctions, packages et la bibliothèque standard de Go



Fonctions



Fonctions

Les fonctions en Go

Les fonctions sont des blocs de code réutilisables.

Elles accomplissent des tâches spécifiques lorsqu'elles sont appelées.

Elles organisent le code et favorisent la modularité.

Déclaration d'une fonction

```
func nomDeLaFonction() {  
    // Corps de la fonction  
}
```

- `func` déclare une fonction.
- `nomDeLaFonction` est le nom choisi.
- Le corps exécute le code.

Fonctions

Arguments et Résultats

Les fonctions peuvent prendre des arguments et renvoyer des résultats.

- Prennent des arguments.
- `a` et `b` sont les arguments de la fonction addition.
- `int` est le type du résultat renvoyé.
- Utilise `return` pour renvoyer un résultat.
- Appel : `addition(1, 2)`.

```
func addition(a, b int) int {  
    return a + b  
}  
  
func main() {  
    fmt.Println(addition(1, 2)) // 3  
}
```

Fonctions

Exemple de fonction

Exemple : Fonction de calcul de moyenne

- Cette fonction prend un tableau de `notes` et renvoie la `moyenne`.

```
func calculMoyenne(notes []float64) float64 {  
    total := 0.0  
    for _, note := range notes {  
        total += note  
    }  
    return total / float64(len(notes))  
}
```

Fonctions

Fonctions recursive

- Une fonction peut s'appeler elle-même, ce qui est utile pour résoudre des problèmes récursifs.

```
func factorielle(n int) int {  
    if n == 0 {  
        return 1  
    }  
    return n * factorielle(n-1)  
}
```

- Cette fonction calcule la factorielle d'un nombre en utilisant la récursivité.
- Par exemple, la factorielle de 5 est $5 * 4 * 3 * 2 * 1 = 120$.

Fonctions

Retours multiples

Les fonctions peuvent renvoyer plusieurs résultats.

- Permet de retourner plusieurs résultats dans une seule fonction.
- Souvent utilisé pour renvoyer une valeur et une erreur.

Exemple : Fonction de division

```
func division(a, b int) (int, error) {  
    if b == 0 {  
        return 0, errors.New("division par zéro")  
    }  
    return a / b, nil  
}
```

Fonctions

Fonctions anonymes

Les fonctions anonymes sont des fonctions sans nom.

- Assignables à des variables ou utilisées comme arguments pour d'autres fonctions.
- Également appelées fonctions littérales.

Exemple : Fonction anonyme

```
func execute(f func()) {
    f()
}

func main() {
    myFunc := func() {
        fmt.Println("Hello")
    }
    execute(myFunc) // Hello
}
```

Fonctions

`defer`

Le mot-clé `defer` permet de reporter l'exécution d'une instruction jusqu'à la fin de la fonction courante.

```
func main() {
    defer fmt.Println("World")
    fmt.Println("Hello")
    defer fmt.Println("!")
    defer fmt.Println("Bye")
}
```

- Les instructions `defer` sont exécutées dans l'ordre inverse (LIFO).

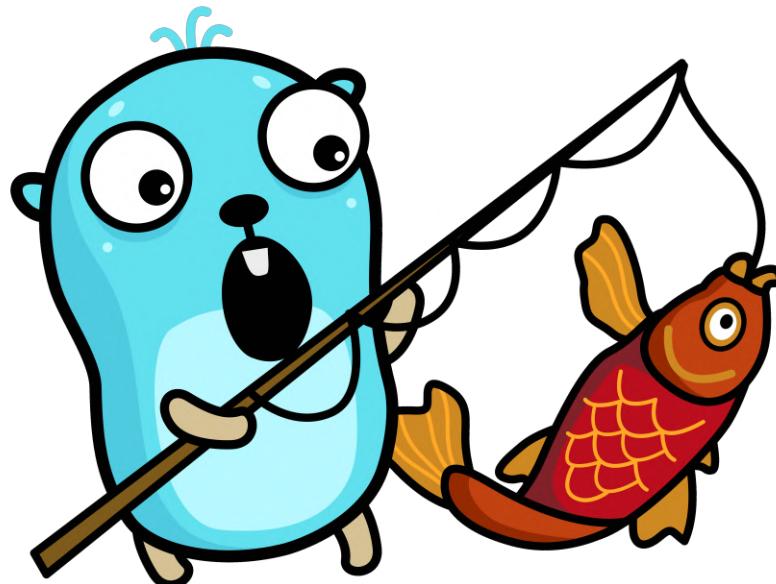
```
Hello
Bye
!
World
```

Fonctions

Conclusion

- **Les fonctions** sont des blocs de code réutilisables pour effectuer des tâches spécifiques.
- Elles peuvent prendre des *arguments*, renvoyer des *résultats* et même être récursives.
- Les fonctions sont un élément **essentiel** de la programmation en Go.
- Elles permettent d'organiser le code et de le rendre *modulaire*.
- On peut utiliser des *fonctions anonymes* et les passer en *argument* à d'autres fonctions.

Exercices



Ressources

Tutoriels sur les Fonctions en Go

- gobyexample.com/functions
- go101.org/article/function-declarations-and-calls.html

Article sur les Fonctions en Go (en français)

- devopssec.fr/article/fonction-golang

Fonctions receveur de structure



Fonctions receveur de structure

Les fonctions receveur sont des fonctions qui agissent sur une structure.

- Les structures Go peuvent inclure des fonctions, définissant ainsi des méthodes propres à la structure.
- Les fonctions au sein des structures simplifient la manipulation des données associées.
- Elles permettent d'encapsuler la logique et les opérations spécifiques à la structure dans des méthodes.

Fonctions receveur de structure

Déclaration d'une fonction receveur

- La fonction `area` est définie comme une fonction receveur de la structure `Rectangle`, en utilisant la syntaxe `func (r Rectangle)`.
- Cela permet à la fonction `area` d'accéder aux champs de la structure `Rectangle`, qui est passée en tant que receveur.

```
type Rectangle struct {
    length int
    width  int
}

func (r Rectangle) area() int {
    return r.length * r.width
}
```

Fonctions receveur de structure

Appel d'une fonction receveur

Comment appeler une fonction receveur ?

- Une fois la fonction receveur déclarée, vous pouvez l'appeler en utilisant une instance de la structure.

```
type Rectangle struct {
    length int
    width  int
}

func (r Rectangle) area() int {
    return r.length * r.width
}

rect := Rectangle{length: 10, width: 5}
area := rect.area()
fmt.Println("L'aire du rectangle est:", area)
```

Fonctions receveur de structure

Utilisation de pointeurs avec des fonctions receveur

Pourquoi utiliser des pointeurs avec des fonctions receveur ?

- Les fonctions receveur travaillent sur une copie de la structure sans pointeurs.
- Les pointeurs permettent de modifier la structure originale.
- Ils sont plus performants avec de grandes structures (éitant la copie).

```
type Rectangle struct {
    length int
    width  int
}

// Fonction receveur avec un pointeur sur Rectangle
func (r *Rectangle) resize(newLength, newWidth int) {
    r.length = newLength
    r.width = newWidth
}
```

```
rect := Rectangle{length: 10, width: 5}
rect.resize(20, 10)
fmt.Println("Nouvelle dimension du rectangle:",
rect.length, "x", rect.width)
```

Fonctions receveur de structure

Méthodes : Valeurs vs Pointeurs

Comparaison entre les fonctions receveur avec valeur et pointeur

- Les fonctions receveur **avec valeur** n'altèrent pas la structure.
- Les fonctions receveur **avec pointeur** ont le pouvoir de la modifier.

```
type Rectangle struct {
    length int
    width  int
}

// Méthode qui ne modifie pas les champs
func (r Rectangle) dimensions() (int, int) {
    return r.length, r.width
}

// Méthode qui modifie les champs
func (r *Rectangle) setDimensions(length, width int) {
    r.length = length
    r.width = width
}
```

- **Valeurs** pour des méthodes sans modification de champs.
- **Pointeurs** pour des méthodes avec modifications ou pour améliorer les performances avec de grandes structures.

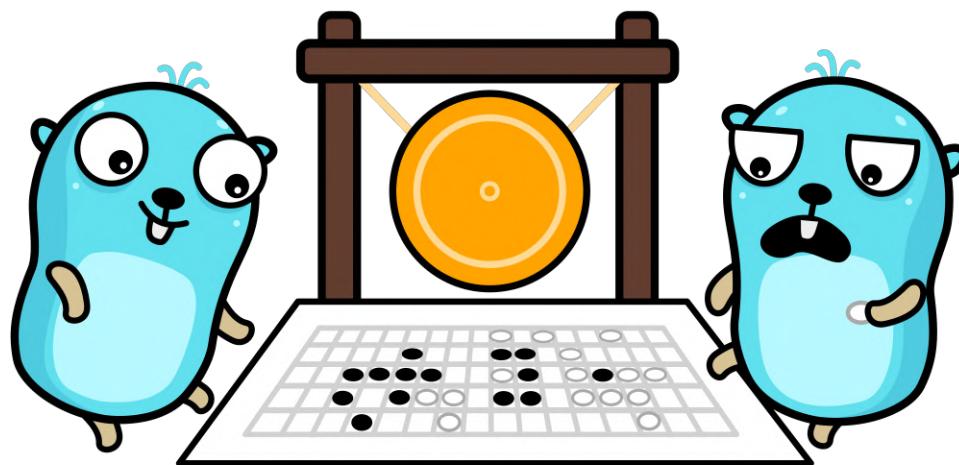
Fonctions receveur de structure

Avantages des fonctions receveur

Pourquoi préférer les fonctions receveur ?

- Organisation du code améliorée en regroupant les fonctions liées à une structure.
- Emulation d'une approche orientée objet, malgré son absence en Go.
- Clarté sémantique accrue, améliorant la compréhension et l'expressivité du code.
- Favorise l'encapsulation des données, une pratique clé en programmation.

Exercices



Gestion des erreurs en Go

Comprendre la gestion des erreurs est essentiel pour écrire des programmes robustes en Go.



panic(err)

Gestion des erreurs en Go

Les erreurs en Go

- Une erreur en Go décrit une condition anormale ou une défaillance.
- Le type `error` est une interface intégrée pour une gestion uniforme des erreurs en Go.

```
type error interface {
    Error() string
}
```

Les fonctions retournent souvent un `error` en tant que dernière valeur pour signaler toute erreur éventuelle.

```
func FaitQuelqueChose() (resultat int, err error) {
    // ...
    if anErrorOccurred {
        return 0, errors.New("une erreur est survenue")
    }
    // ...
    return resultat, nil
}
```

Gestion des erreurs en Go

Retourner une erreur

Comment retourner une erreur ?

- Une fonction renvoie une erreur en cas d'échec.
- C'est la méthode de gestion des erreurs en Go.

```
func Diviser(a, b int) (int, error) {  
    if b == 0 {  
        return 0, errors.New("division par zéro")  
    }  
    return a / b, nil  
}
```

Gestion des erreurs en Go

Traitement des erreurs

Il est important de toujours vérifier les erreurs retournées par les fonctions.

```
func Diviser(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("division par zéro")
    }
    return a / b, nil
}

func main() {
    resultat, err := Diviser(10, 0)
    if err != nil {
        log.Fatalf("Une erreur est survenue : %v", err)
    }
    fmt.Println("Résultat de la division :", resultat)
}
```

Gestion des erreurs en Go

Erreurs personnalisées

Créer des erreurs personnalisées

- Personnalisation des erreurs possible pour ajouter plus de contexte ou de données sur l'erreur.

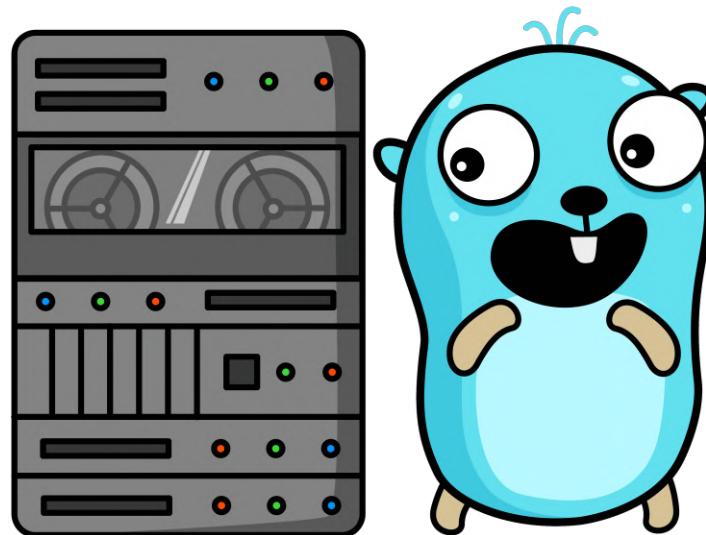
```
type ErreurDivisionParZero struct{}

func (e *ErreurDivisionParZero) Error() string {
    return "tentative de division par zéro"
}

func Diviser(a, b int) (int, error) {
    if b == 0 {
        return 0, new(ErreurDivisionParZero)
    }
    return a / b, nil
}
```

```
resultat, err := Diviser(10, 0)
if err != nil {
    switch err.(type) {
    case *ErreurDivisionParZero:
        fmt.Println("Division par zéro détectée :", err)
    default:
        fmt.Println("Autre erreur détectée :", err)
    }
}
```

Les packages en Go



GOPHER

Les packages en Go

Qu'est-ce qu'un Package ?

Les packages sont essentiels pour organiser le code en Go, créant des unités réutilisables et maintenables.

- Les packages regroupent des fichiers de code associés.
- Chaque dossier dans le chemin de code Go représente un package.
- Le nom du package correspond au nom du dossier.

```
- /chemin/vers/code/
  - main.go          // appartient au package main
  - util/
    - helper.go    // appartient au package util
```

Les packages en Go

Créer un package

Déclaration d'un package

- La première ligne d'un fichier Go spécifie le package auquel il appartient.

```
package util // Le fichier appartient au package util
```

Organisation

Tous les fichiers dans le même dossier doivent appartenir au même package.

Fonctionnalité

- Les packages encapsulent du code en vue de sa réutilisation dans d'autres parties du programme.

Les packages en Go

Importer un package

Utilisation simple

- Utilisez le mot-clé `import` pour utiliser le contenu d'un autre package.

```
import "fmt" // fmt est un package de la bibliothèque standard
```

Importation de son propre package

```
import "chemin/vers/code/util"
```

Importation de plusieurs packages

```
import (  
    "fmt"  
    "math"  
    "chemin/vers/code/util"  
)
```

Ouvrons une parenthèse sur les modules en Go



Les modules en Go

Introduction aux Modules

Les modules sont essentiels pour la gestion des dépendances en Go, simplifiant la gestion des packages et des versions.

- Un module est un groupe de packages Go utilisés conjointement.
- Chaque module est défini par un fichier `go.mod` qui indique les dépendances.

Les modules en Go

Créer un module

Initialisation d'un module

- Utilisez la commande `go mod init` pour initialiser un nouveau module.

```
go mod init monmodule
```

Dépendances

- Les dépendances sont ajoutées automatiquement au fichier go.mod lors de l'importation de modules externes.

Structure d'un module

- Un module est un dossier contenant un fichier `go.mod` et un ou plusieurs packages Go.

```
projet/
    go.mod
    main.go
    util/
        helper.go
```

Les packages en Go

⚠ La portée des identificateurs

Public vs Privé

- En Go, si un identifiant commence par une **lettre majuscule**, il est **accessible à l'extérieur** du package.
- Si un identifiant commence par une **lettre minuscule**, il est **privé** au package.
- Cette règle s'applique aux **fonctions, variables, constantes, types**, etc.

```
package util

// Public, car commence par une majuscule
func Helper() { ... }

// Privé, car commence par une minuscule
func assistant() { ... }
```

Accès

- Les éléments publics d'un package sont accessibles partout où le package est importé.
- Les éléments privés ne sont accessibles que dans le package d'origine.

Les packages en Go

Bonnes pratiques pour les packages

Nom du package

- Optez pour des noms courts et évocateurs.

Taille

- Favorisez des packages petits, focalisés sur une seule responsabilité.

Couplage

- Limitez les dépendances entre packages pour réduire le couplage.

Réutilisation

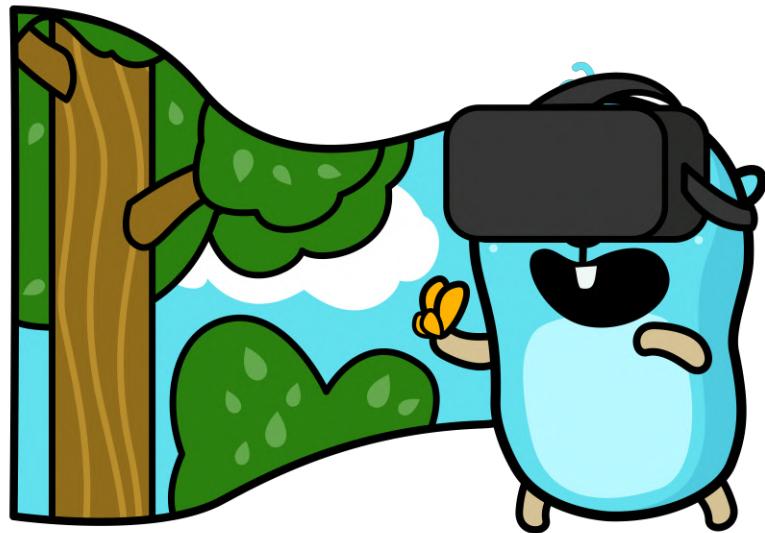
- Concevez les packages en envisageant leur réutilisation.

Exemple

GO! GO!



La bibliothèque standard de Go



La bibliothèque standard de Go

Introduction

La bibliothèque standard de Go est une collection de packages maintenus par l'équipe de Go et inclus dans chaque installation.

Ces bibliothèques offrent des fonctionnalités cruciales pour le langage et s'intègrent efficacement au système d'exploitation.

Bibliothèques courantes :

- `fmt` : Formatage et sortie.
- `math` : Fonctions mathématiques.
- `os` : Interactions système d'exploitation.

```
fmt.Println("Hello, World!")
formattedString := fmt.Sprintf("Hello, %s!", "World")
```

```
x := math.Sqrt(4)
y := math.Pow(2, 3)
```

```
file, err := os.Open("fichier.txt")
os.Remove("fichier.txt")
os.Exit(1)
```

La bibliothèque standard de Go

Utilisation de la bibliothèque standard

```
// Utilisation de fmt
import "fmt"

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

```
// Utilisation de math
import "math"

func main() {
    result := math.Sqrt(16)
    fmt.Println("Le carré de 16 est", result)
}
```

```
// Utilisation de os
import "os"

func main() {
    _, err := os.Stat("fichier.txt")
    if os.IsNotExist(err) {
        fmt.Println("Le fichier n'existe pas")
    }
}
```

La bibliothèque standard de Go

La bibliothèque `log` en Go

La bibliothèque `log` de Go est destinée à l'enregistrement des informations de débogage.

Elle propose un moyen simple et puissant de gérer les logs, avec personnalisation de la sortie et du format des logs.

Fonctionnalités clés :

- Enregistrement standardisé des informations.
- Personnalisation de la sortie.
- Prise en charge des niveaux de log.

La bibliothèque standard de Go

Utilisation de la bibliothèque `log`

```
package main

import (
    "log"
    "os"
)

func main() {
    // Création ou ouverture du fichier de log
    file, err := os.OpenFile("info.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    log.Println("Message de log en console")

    // Configuration du logger
    log.SetOutput(file)
    log.Println("Message de log dans le fichier")
}
```

La bibliothèque standard de Go

Conclusion

- La bibliothèque standard de Go est essentielle pour tout développement en Go.
- Elle offre des outils solides et performants tels que `fmt`, `math`, et `os`, qui sont indispensables au quotidien.
- La maîtrise de la bibliothèque `log` est également cruciale pour le débogage et la surveillance efficaces des applications Go.

Exercices



Ressources

Fonctions et Manipulation de Chaînes

- gobyexample.com/string-functions
- yourbasic.org/golang/packages-explained/

Packages et Ordre d'Exécution en Go

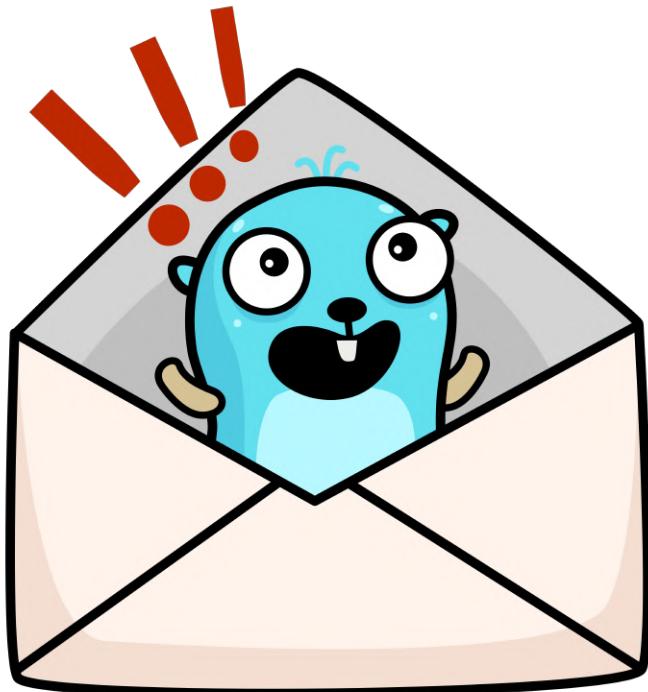
- yourbasic.org/golang/package-init-function-main-execution-order/
- go101.org/article/packages-and-imports.html

Gestion des Logs et Portée des Variables

- yourbasic.org/golang/log-to-file/
- devopssec.fr/article/portee-des-variables-golang

Fichiers en Go

Comment lire et écrire des fichiers en Go ?



Fichiers en Go

Introduction à la gestion de fichiers en Go

- La manipulation de fichiers est cruciale dans de nombreuses applications Go.
- Les packages `os` et `io` permettent de lire, écrire et gérer les fichiers en toute simplicité.
- **Exemple : Lire un fichier**
- **Exemple : Écrire un fichier**

```
func main() {  
    data, err := os.ReadFile("exemple.txt")  
    message := string(data)  
    if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Println(message)  
}
```

```
func main() {  
    message := "Bonjour Go !"   
    err := os.WriteFile("exemple.txt", []byte(message), 0644)  
    if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Println("Fichier écrit avec succès")  
}
```

Fichiers en Go

Techniques Avancées de Manipulation de Fichiers

- Ouverture de fichiers avec `os.Open` :

```
file, err := os.Open("exemple.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

- Création de fichiers avec `os.Create` :

```
file, err := os.Create("nouveau.txt")
if err != nil {
    log.Fatal(err)
}
file.Close()
```

- Lecture tamponnée avec `bufio` :

```
file, _ := os.Open("exemple.txt")
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
file.Close()
```

- Copie de fichiers avec `io.Copy` :

```
src, _ := os.Open("source.txt")
defer src.Close()
dst, _ := os.Create("destination.txt")
defer dst.Close()
io.Copy(dst, src)
```

Bibliothèque JSON en Go



go magic()

Bibliothèque JSON en Go

Maîtriser la bibliothèque `json` en Go

La bibliothèque `json` en Go est essentielle pour convertir des données entre les structures Go et le format JSON, facilitant la communication avec des APIs, des services web et des systèmes de stockage de données.

- **Fonctionnalités clés :**

- **Sérialisation** : Convertit des structures Go en JSON.
- **Désérialisation** : Interprète le JSON en structures Go.

Utilisez `json` efficacement pour gérer des données complexes et interagir avec des systèmes hétérogènes, renforçant ainsi la flexibilité et la puissance de vos applications Go.

Bibliothèque JSON en Go

Exemple avec `json`

- Sérialisation d'une structure Go en JSON :

```
type Utilisateur struct {
    Nom   string
    Age   int
}
utilisateur := Utilisateur{"Alice", 30}
jsonData, _ := json.Marshal(utilisateur)
fmt.Println(string(jsonData))

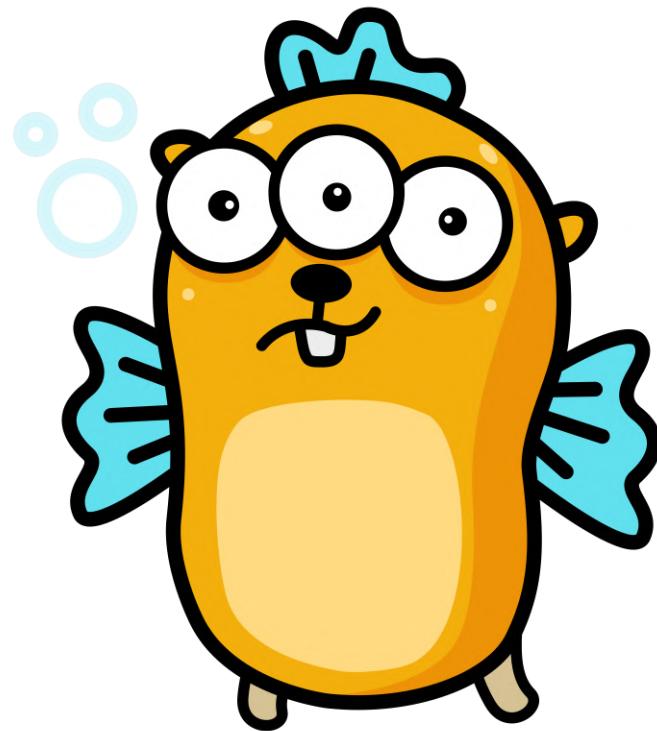
>{"Nom":"Alice","Age":30}
```

- Désérialisation de JSON en structure Go :

```
type Utilisateur struct {
    Nom   string
    Age   int
}
var utilisateur Utilisateur
jsonData := []byte(`{"Nom":"Alice","Age":30}`)
json.Unmarshal(jsonData, &utilisateur)
fmt.Println(utilisateur)

{Alice 30}
```

Exercices



Ressources

Lecture et Écriture de Fichiers

- gobyexample.com/reading-files
- gobyexample.com/writing-files

Traitement JSON

- gobyexample.com/json

Manipulation de Fichiers et Traitement Ligne par Ligne

- yourbasic.org/golang/read-file-line-by-line/
- yourbasic.org/golang/append-to-file/
- yourbasic.org/golang/temporary-file-directory/
- yourbasic.org/golang/json-example/

Lire et Écrire dans un Fichier en Go (en français)

- devopssec.fr/article/lire-et-ecrire-dans-un-fichier-golang

Développement Web en Go



Développement Web en Go

Go : Idéal pour le développement Web.

Sa légèreté et sa rapidité en font un langage de choix.

La bibliothèque standard de Go offre tout le nécessaire pour les applications Web.

Bibliothèque SQL



Bibliothèque SQL

Introduction à la bibliothèque SQL

La bibliothèque SQL de Go facilite l'interaction avec les bases de données relationnelles.

- Installation du driver PostgreSQL : `go get -u github.com/lib/pq`
- Installation du driver MySQL : `go get -u github.com/go-sql-driver/mysql`
- Installez le driver correspondant à votre base de données.
- Cela ajoutera la dépendance au fichier `go.mod` .

Bibliothèque SQL

Connexion à PostgreSQL

- Importer le package SQL : `import "database/sql" `
- Importer le driver PostgreSQL : `import _ "github.com/lib/pq" `
- **Ouvrir une connexion :**

```
db, err := sql.Open("postgres", "host=<localhost> user=<username> password=<password> " +  
    "dbname=<dbname> sslmode=disable")  
if err != nil {  
    log.Fatal(err)  
}  
defer db.Close()
```

Bibliothèque SQL

Exécution de Requêtes

- **Requête `SELECT` :**

```
rows, err := db.Query("SELECT id, name FROM users")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
```

- **Requête `INSERT` :**

```
res, err := db.Exec("INSERT INTO users (name) VALUES ($1)", "Alice")
if err != nil {
    log.Fatal(err)
}
id, err := res.LastInsertId()
if err != nil {
    log.Fatal(err)
}
fmt.Println(id)
```

- **Parcourir les résultats :**

```
for rows.Next() {
    var id int
    var name string
    if err := rows.Scan(&id, &name); err != nil {
        log.Fatal(err)
    }
    fmt.Println(id, name)
}
```

Bibliothèque SQL

Requêtes Paramétrées

```
stmt, err := db.Prepare("SELECT id, name FROM users WHERE age > $1")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()
rows, err := stmt.Query(18)
```

Bibliothèque SQL

Gestion des Transactions

Transaction : un groupe de requêtes à exécuter de manière atomique (toutes ou aucune).

- Démarrer une transaction :

```
tx, err := db.Begin()  
if err != nil {  
    log.Fatal(err)  
}
```

- Exécuter des requêtes dans la transaction :

```
_, err = tx.Exec("INSERT INTO users (name) VALUES ($1)", "Alice")  
if err != nil {  
    tx.Rollback() // Annuler la transaction  
    log.Fatal(err)  
}  
// ...
```

- Valider la transaction :

```
err = tx.Commit() // Valider la transaction  
if err != nil {  
    log.Fatal(err)  
}
```

Utilisation de net/http comme Client en Go



Utilisation de net/http comme Client en Go

Introduction à net/http en tant que Client

- `net/http` pour les requêtes HTTP en Go.
- Idéal pour consommer des APIs REST.
- Ou bien pour faire des requêtes HTTP en tant que client.

Exemple d'une requête Get

```
resp, err := http.Get("https://pokeapi.co/api/v2/pokemon/pikachu")
if err != nil {
    log.Fatal(err)
}
defer resp.Body.Close()
body, err := io.ReadAll(resp.Body)
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(body))
```

Utilisation de net/http comme Client en Go

Autres types de requêtes HTTP

- Requêtes POST, PUT, DELETE pour des opérations CRUD.
- Exemple de requête POST :

```
jsonData := []byte(`{"name": "newpokemon", "type": "electric"}`)  
resp, err := http.Post("https://pokeapi.co/api/v2/pokemon", "application/json", bytes.NewBuffer(jsonData))
```

Utilisation de net/http comme Client en Go

Traitement des réponses JSON

```
{  
    "base_experience": 112,  
    "name": "pikachu",  
    "sprites": { "front_default": "...", ... },  
    ...  
}
```

```
type Pokemon struct {  
    BaseExperience int      `json:"base_experience"  
    Name          string   `json:"name"  
    PokedexOrder int      `json:"order"  
    Sprites       struct {  
        FrontDefault string `json:"front_default"  
    } `json:"sprites"  
}
```

```
func main() {  
    url := "https://pokeapi.co/api/v2/pokemon/pikachu"  
    resp, err := http.Get(url)  
    // ... Traitement des erreurs  
    defer resp.Body.Close()  
    body, err := io.ReadAll(resp.Body)  
    // ... Traitement des erreurs  
    var pokemon Pokemon  
    err = json.Unmarshal(body, &pokemon)  
    // ... Traitement des erreurs  
    fmt.Printf("%#+v\n", pokemon)  
}
```

Création d'un Serveur Web avec net/http en Go



Création d'un Serveur Web avec net/http en Go

- `net/http` : Crée des serveurs HTTP en Go.
- Utile pour les APIs REST ou les sites Web statiques.

Mise en Place d'un Serveur HTTP Simple

- `http.HandleFunc` pour les routes.
- Démarrage avec `http.ListenAndServe` .
- **Démarrage du Serveur** : Écoute sur le port 8080.

```
http.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Page d'accueil")
})
log.Fatal(http.ListenAndServe(":8080", nil))
```

Création d'un Serveur Web avec net/http en Go

Servir des Fichiers HTML Statiques

- `http.FileServer` : Pour servir un dossier de fichiers statiques.
- **Configurer la Route** : Définir la route pour le dossier des fichiers HTML.
- **Accès aux Fichiers** : Les fichiers du dossier 'static' sont accessibles via le serveur web.

```
func main() {
    // Dossier contenant les fichiers statiques
    fs := http.FileServer(http.Dir("static"))

    // Configuration de la route
    http.Handle("/", fs)

    // Démarrage du serveur
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Création d'un Serveur Web avec net/http en Go

Route Renvoyant une Réponse JSON

- Structuration des données pour la réponse.
- Sérialisation avec `json.Marshal`.

```
type Response struct {
    Message string `json:"message"`
}

http.HandleFunc("GET /json", func(w http.ResponseWriter, r *http.Request) {
    response := Response{Message: "Données en JSON"}
    jsonData, _ := json.Marshal(response)
    w.Header().Set("Content-Type", "application/json")
    w.Write(jsonData)
})
```

Création d'un Serveur Web avec net/http en Go

Gestion des Erreurs HTTP

- Envoi de codes d'état HTTP appropriés.

```
http.HandleFunc("GET /error", func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Erreur trouvée", http.StatusInternalServerError)
})
```

Exercices



Ressources

Service de Base de Données ElephantSQL

- [ElephantSQL - Service de base de données SQL géré](#)

Communication HTTP en Go

- [gobyexample.com/http-client](#)
- [gobyexample.com/http-server](#)

Exemple de Serveur HTTP en Go

- [yourbasic.org/golang/http-server-example/](#)

Documentation sur la Gestion de Bases de Données SQL en Go

- [go-database-sql.org - Vue d'ensemble](#)

Concurrence en Go



oh no

Concurrence en Go

Introduction à la concurrence

La concurrence en Go permet d'exécuter plusieurs tâches en parallèle, améliorant ainsi l'efficacité et la performance des programmes.

C'est comme si plusieurs caissiers dans un supermarché pouvaient servir plusieurs clients simultanément, accélérant le service.

Concurrence en Go

Les Goroutines, multitâche simplifié

Les goroutines sont des threads légers gérés par le runtime Go, permettant une exécution concurrente de fonctions.

Définition d'une Goroutine

```
go maFonction()
```

Un simple préfixe `go` devant un appel de fonction crée une nouvelle goroutine.

Exemple détaillé de Goroutine

```
func saluer(msg string) {
    for i := 0; i < 10; i++ {
        fmt.Printf("%d - %s\n", i, msg)
        time.Sleep(650 * time.Millisecond)
    }
}

func main() {
    go saluer("Bonjour depuis la goroutine")
    time.Sleep(1 * time.Second)
    saluer("Bonjour depuis la fonction principale")
    fmt.Println("Fin du programme")
}
```

Concurrence en Go

Communication avec les canaux

Les canaux sont utilisés pour la communication entre goroutines, permettant l'échange de données de manière synchronisée et sécurisée.

Création et utilisation d'un canal

```
monCanal := make(chan int) // Création d'un canal
go func() {
    fmt.Println("Je suis une goroutine")
    time.Sleep(1 * time.Second)
    monCanal <- 42 // Envoi d'une valeur
}()
fmt.Println("Je suis la fonction principale")
valeur := <-monCanal // Attente et réception d'une valeur
fmt.Println("Reçu :", valeur)
```

- Ce canal permet d'envoyer et de recevoir des données entre goroutines.

Concurrence en Go

Synchronisation avec les WaitGroups

Les `WaitGroups` synchronisent les goroutines en attendant leur achèvement.

Création et utilisation d'une WaitGroup

```
var wg sync.WaitGroup // Création d'une WaitGroup
for i := 0; i < 3; i++ {
    wg.Add(1) // Incrémentation du compteur de la WaitGroup
    go func(i int) {
        defer wg.Done() // Décrémentation du compteur de la WaitGroup
        time.Sleep(time.Duration((i + 1) * time.Second))
        fmt.Println("Tâche", i)
    }(i)
}
wg.Wait() // Attente de la fin de toutes les goroutines (compteur à 0)
fmt.Println("Fin")
```

- Ici, `WaitGroup` attend que les trois tâches soient terminées.

Concurrence en Go

Mutex pour la sécurité des données

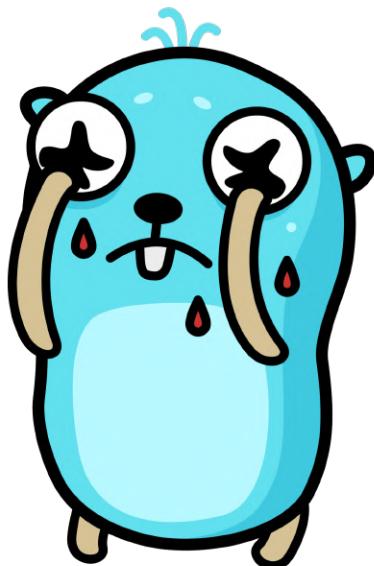
Les `Mutex` verrouillent l'accès aux données partagées entre goroutines, prévenant ainsi les conflits et les conditions de course.

Création et utilisation d'un Mutex

```
var mu sync.Mutex // Création d'un mutex
var compteur int
for i := 0; i < 10; i++ {
    go func() {
        mu.Lock() // Verrouillage du mutex (attente si déjà verrouillée)
        compteur++ // Incrémentation du compteur
        mu.Unlock() // Déverrouillage du mutex
    }()
}
time.Sleep(1 * time.Second)
fmt.Println(compteur)
```

- La `Mutex` assure que seul une goroutine à la fois peut modifier `compteur`.

Cas pratique



please,
stop it!



Concurrence en Go

Conclusion - La concurrence, un Atout majeur en Go

- La maîtrise de la concurrence en Go via les goroutines, canaux, `WaitGroup`, et `Mutex` offre un vaste potentiel pour optimiser et accélérer vos programmes.
- C'est un outil puissant qui, utilisé judicieusement, améliore grandement la performance et l'efficacité de vos applications.

Ressources

Goroutines et Canaux (Channels)

- gobyexample.com/goroutines
- gobyexample.com/channels
- gobyexample.com/mutexes
- gobyexample.com/atomic-counters
- gobyexample.com/waitgroups

Canaux en Profondeur

- go101.org/article/channel.html

Explication des Canaux (Channels) en Go

- yourbasic.org/golang/channels-explained/
- yourbasic.org/golang/wait-for-goroutines-waitgroup/
- yourbasic.org/golang/select-explained/

Goroutines et Canaux en Go (en français)

- devopssec.fr/article/goroutines-golang
- devopssec.fr/article/channels-golang

Fin du Cours de Go

Félicitations !

- Vous avez terminé avec succès le cours de programmation en Go.
- Vous avez maintenant acquis des compétences solides en Go.

Récapitulatif

- Vous avez appris les fondamentaux de Go, la manipulation de données, le développement web, et bien plus.
- Vous êtes désormais équipé pour créer des applications robustes et performantes en Go.

Prochaines Étapes

- Continuez à pratiquer et à explorer des projets plus complexes.
- Approfondissez votre connaissance de la bibliothèque standard de Go ainsi que celles de la communauté.
- Restez informé des dernières tendances et mises à jour de Go.

Ressources

Deux cheatsheets pour Go :

- github.com/a8m/golang-cheat-sheet
- [Devhints](#)

Autres sites sympas :

- [Your basic Go](#)
- [Go 101](#)
- [Go by example](#)
- [Devopssec](#)
- [FreeCodeCamp](#)

