

# Go - Cours 1

## Les fondamentaux



# Programme du jour

## Introduction

---

Part 1

Qui suis-je

Les objectifs du module

Installation de Go

## Le cours

---

Part 2

Live Coding (peut-être)

---

Part 3

## La pratique

---

Part 4

# Introduction

# Qui suis-je

- Axelle Lança
- Développeuse full stack (Go, Python, React/vue)  
depuis 5 ans
- J'aime pas le front
- Pratique du Go depuis 4 ans
- [axelle@lanca.fr](mailto:axelle@lanca.fr)

# Le module

- 35 heures
- 5 journées de 7 heures
- Notation à définir mais probablement :
  - 4 TP de 5 points
  - Devoir final : un projet en groupe sur le jour 5

# Objectifs du cours

- Maîtriser la syntaxe de Go
- Écrire, compiler et tester un programme Go
- Savoir structurer un projet Go complexe
- Utiliser les outils fournis avec le langage
- Voir 2/3 package externes

# Thèmes abordés

- Installation de Go
- Syntaxe et les types de Go
- Fonctions et structures
- Gestion des erreurs
- Tests unitaires
- CLI
- API
- Concurrence, channels
- Les interfaces
- etc.

# Installation de Go

- Pour PC et Linux : <https://go.dev/dl> et suivre les étapes
- Mac : brew install go
- Vérifier l'installation : go version

# Le cours

# Pourquoi un nouveau langage ?

## La philosophie de Go

- Contexte : Crée chez Google en 2009 (Rob Pike, Ken Thompson, Robert Griesemer).
- **Frustrations avec les langages existants (C++, Java) :**
  - Lenteur de Compilation : Des cycles de développement de plusieurs dizaines de minutes.
  - Complexité Excessive : Des langages devenus trop vastes, difficiles à maîtriser.
  - Gestion de la Concurrence : Les threads traditionnels sont complexes et sources de bugs.
- Go n'est pas une révolution, c'est une simplification radicale.

# Les 4 piliers de Go

## 1. Simplicité :

- ~25 mots-clés. Un langage qui tient dans la tête.
- Lisibilité avant tout. `gofmt` impose un style unique.

## 2. Performance & Déploiement :

- Compilé en un binaire natif unique.
- Pas de dépendances externes ou de machine virtuelle à installer sur le serveur.

## 3. Concurrence :

- Les Goroutines et les Channels sont au cœur du langage.
- Conçu pour les processeurs multi-cœurs modernes.

## 4. Outilage Intégré :

- Gestion des dépendances (`go mod`), tests (`go test`), formatage (`go fmt`), etc

# L'Environnement de Travail : GOPATH vs. Modules

- **L'Ancienne Méthode (avant Go 1.11) : GOPATH**
  - Un seul dossier pour TOUS les projets Go sur la machine.
  - Problème majeur : impossible d'avoir des versions différentes d'une même librairie pour des projets différents ("Dependency Hell").
- **La Méthode Moderne (Obligatoire aujourd'hui) : Go Modules**
  - Chaque projet est un module autonome.
  - Peut vivre n'importe où sur votre disque dur.
  - Défini par un fichier go.mod à sa racine.

# Mettre en Place un Projet Go

- 1. Créer un dossier :** `mkdir mon-projet && cd mon-projet`
- 2. Initialiser le module :** `go mod init github.com/votre-nom/mon-projet`
  - Crée le fichier go.mod.
- 3. Les Fichiers Clés :**
  - go.mod : Le passeport du projet. Contient son nom et la liste de ses dépendances directes avec leurs versions.
  - go.sum : Le sceau de sécurité. Contient les signatures de TOUTES les dépendances pour garantir des builds reproductibles.

# Anatomie d'un programme

```
package main

import "fmt"

// main est le point d'entrée
// de tout programme exécutable.
func main() {
    fmt.Println("Hello, World!")
}
```

- `package main`: Déclare un programme exécutable.
- `import "fmt"`: Importe une librairie standard pour les entrées/sorties.
- `func main()`: La fonction qui sera exécutée au lancement.

# Architecture d'un Projet Go

```
mon-api/
├── go.mod
├── go.sum
└── cmd/
    └── api/
        └── main.go
internal/
└── domain/
    └── produit.go      // Définit la struct Produit
└── handler/
    └── http/
        └── produit_handler.go // Gère les requêtes HTTP pour les produits
store/
└── produit_store.go   // Gère la logique de la base de données pour les produits
```

- **Objectif :** Organiser le code pour qu'il soit lisible et maintenable.
- **/cmd :** Contient les points d'entrée (`main.go`) de vos applications.
  - Ex : `/cmd/api/`, `/cmd/worker/`
- **/internal :** Le cœur de votre logique, privé au projet.
  - **/domain :** Définition des `structs` métier (ex: `Produit`).
  - **/handler :** Couche HTTP (gère les requêtes/réponses).
  - **/store :** Couche de persistance (parle à la base de données).
- **Principe clé :** Séparation des préoccupations.

# Les commandes essentielles

- `go run .`
  - Compile et exécute le code en une seule étape.
  - Idéal pour le **développement**.
- `go build .`
  - Compile le code et produit un binaire exécutable.
  - Idéal pour la **production**.
- `go fmt .`
  - Formate automatiquement tout le code du projet.
  - **Non négociable : à utiliser tout le temps !**

# Variables, Types & Conventions

- **Déclaration :**

- Longue : `var nom string = "Alice"`
- Avec inférence : `var nom = "Alice"`
- Courte : `nom := "Alice"` (la plus commune dans les fonctions)

- **Types de base :** `string`, `int`, `float64`, `bool`, `byte`, `rune`.

- **La Valeur Zéro :** Toute variable a une valeur par défaut (`0`, `""`, `false`). Pas de `null` ou `undefined` pour les types de base.

- **La Convention de Casse = Visibilité :**

- `maVariable` (minuscule) -> **privé** au package.
- `MaVariable` (majuscule) -> **public** (exporté).

# Déclarer ses propres types

```
type Temperature float64
type Distance    float64
type UtilisateurID int

var tempMaison Temperature = 21.5
var distanceParisLyon Distance = 450.0
var premierUtilisateur UtilisateurID = 1

// Même si Temperature et Distance sont toutes les deux des float64,
// le système de types ne les considère pas comme interchangeables.
// if tempMaison == distanceParisLyon { ... } // Erreur de compilation !
```

# Les Fonctions

- **Syntaxe :** `func nom(param type) (typeRetour1, typeRetour2) { ... }`
- **Passage des Arguments : TOUJOURS par Valeur**
  - La fonction reçoit une **copie** des données.
  - Elle ne peut pas modifier la variable originale.
- **Retours Multiples : La Force de Go**
  - Permet de retourner un résultat ET une erreur.
  - Le pattern idiomatique : `valeur, err := maFonction()`

```
//      (1)      (2)          (3)          (4)
func nomDeLaFonction(param1 string, param2 int) (string, error) {
    // (5)
    // Corps de la fonction : la logique
    // ...
    // (6)
    return "un résultat", nil
}
```

# La Gestion des Erreurs en Go

- **Philosophie** : Les erreurs sont des valeurs, pas des exceptions.
  - Pas de `try...catch...finally`.
  - Le flux de contrôle est explicite et linéaire.
- **Le Pattern Omniprésent** : `if err != nil`
  - Une fonction qui peut échouer retourne une `error`.
  - On teste cette erreur **immédiatement** après l'appel.

```
nombre, err := strconv.Atoi("pas un nombre")
if err != nil {
    // On gère l'erreur ici.
    fmt.Println("Erreur:", err)
    return
}
// Le code continue seulement si err est nil.
```

# Structures de Contrôle (1/2)

- **if / else if / else**
  - Classique, mais sans parenthèses autour des conditions.
  - **Version idiomatique :** `if` avec instruction d'initialisation.

```
if n, err := strconv.Atoi(s); err == nil {  
    // n n'existe que dans ce bloc if/else  
}
```

- **for : La Seule et Unique Boucle**
  1. **Complète (style C) :** `for i := 0; i < 10; i++`
  2. **Conditionnelle (style while) :** `for n < 100`
  3. **Infinie :** `for {}`
  4. **Itération (range) :** `for index, valeur := range maSlice`

# Structures de Contrôle (2/2)

- **switch : L'Aiguillage Amélioré**
  - **Pas de fallthrough par défaut :** Un `break` est implicite à la fin de chaque `case`. C'est plus sûr !
  - **Cas multiples :** `case "a", "e", "i", "o", "u":`
  - **switch "tagless" (sans étiquette) :** Utile pour une chaîne de `if/else` plus propre.

```
switch {  
    case age < 18: ...  
    case age >= 18: ...  
}
```

# Collections (1/3) : Array vs. Slice

- **Array** : `[4]int`
  - Taille **fixe**, définie à la compilation.
  - Passé par **valeur** (copie lourde et coûteuse).
  - Rarement utilisé directement.
- **Slice** : `[]int`
  - Vue **dynamique** et flexible sur un tableau sous-jacent.
  - Contient un pointeur, une longueur (`len`) et une capacité (`cap`).
  - Passé par **référence** (copie légère).
  - À utiliser **99% du temps**.

# Collections (2/3): Manipuler les Slices

- **Création :**

- Littéral : `nombres := []int{10, 20, 30}`
- `make` : `slice := make([]int, 5, 10) // len=5, cap=10`

- **Ajouter un élément :** `append`

- **IMPÉRATIF** : Toujours ré-assigner le résultat !
- `nombres = append(nombres, 40)`

- **Itérer :** `for...range`

- `for index, valeur := range nombres { ... }`
- Pour ignorer l'index : `for _, valeur := range nombres { ... }`

# Collections (3/3): Les Maps

- **Concept :** Dictionnaire clé-valeur, table de hachage.

- `map [TypeDeLaClé] TypeDeLaValeur`

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

- **Manipulation :**

- Ajouter/Modifier : `ages["Alice"] = 30`

- Supprimer : `delete(ages, "Bob")`

- **Vérifier l'existence : Le "comma ok idiom"**

```
valeur, ok := ages["Charlie"]
if ok {
    // La clé existe
} else {
    // La clé n'existe pas
}
```

- **Attention :** L'ordre d'itération sur une map n'est **jamais** garanti !

# Travaux pratiques

# Consignes

- **Projet :** Créer un Mini-CRM en Ligne de Commande.

- **Fonctionnalités :**

1. Afficher un menu principal en boucle.

2. **Ajouter** un contact (ID, Nom, Email).

3. **Lister** tous les contacts.

4. **Supprimer** un contact par son ID.

5. **Mettre à jour** un contact.

6. **Quitter** l'application.

7. **Ajouter un contact** grâce à des flags

- **Concepts à utiliser :** "comma ok idiom", `for {}`, `switch`, `map`, `if err != nil`, `strconv`, `os.Stdin`, `bufio` etc.

- Travail individuel ou par 2

- **Rendu :** un lien d'un repo github avec un Readme correct

# Cheatsheets

# Fonctions fmt

Fonction	Description	Exemple	Sortie
<code>fmt.Println()</code>	Affiche sans saut de ligne	<code>fmt.Println("Hello")</code>	<code>Hello</code>
<code>fmt.Println()</code>	Affiche avec un saut de ligne	<code>fmt.Println("Hello")</code>	<code>Hello\n</code>
<code>fmt.Printf()</code>	Affiche avec un format spécifié (placeholders)	<code>fmt.Printf("Age: %d", 30)</code>	<code>Age: 30</code>
<code>fmt.Sprint()</code>	Retourne une chaîne (ne l'imprime pas)	<code>s := fmt.Sprint("Hi")</code>	<code>s = "Hi"</code>
<code>fmt.Sprintln()</code>	Retourne une chaîne avec saut de ligne	<code>s := fmt.Sprintln("Hi")</code>	<code>s = "Hi\n"</code>
<code>fmt.Sprintf()</code>	Retourne une chaîne formatée (comme <code>Printf</code> mais sans imprimer)	<code>s := fmt.Sprintf("Age: %d", 30)</code>	<code>s = "Age: 30"</code>
<code>fmt.Fprint(w, ...)</code>	Écrit dans un writer (ex : un fichier ou buffer)	<code>fmt.Fprint(os.Stdout, "Hello")</code>	écrit dans stdout
<code>fmt.Fprintf(w, ...)</code>	Format + écrit dans un writer	<code>fmt.Fprintf(os.Stderr, "Error: %v", err)</code>	<code>Error: erreur</code> dans stderr
<code>fmt.Fprintln(w, ...)</code>	Écrit dans un writer avec saut de ligne	<code>fmt.Fprintln(os.Stdout, "Hello")</code>	<code>Hello\n</code> dans stdout

# Formatage

## Go Formatage Cheatsheet (%fmt)

Spécificateur	Description	Exemple de sortie
%v	Valeur par défaut	true, 123, map[a:b]
%+v	Valeur avec champs nommés	{Name:Axelle Age:30}
%#v	Syntaxe Go	main.User{Name:"Axelle"}
%T	Type de la valeur	int, string
%%	Littéral %	%
%t	Booléen	true
%b	Binaire	1101
%d	Décimal	123
%o	Octal	173
%x / %X	Hexadécimal	7b / 7B
%f	Float (décimal)	3.14
%e / %E	Float scientifique	1.23e+03 / 1.23E+03
%s	String	bonjour
%q	String entre guillemets	"bonjour"
%p	Pointeur	0xc000010230
%c	Caractère	A
%U	Unicode formaté	U+0041