

# Le langage Go

Un langage de programmation *impérative et concurrence*

# Le langage Go

- ▶ Développé en 2007–2009 chez Google par 3 ingénieurs sous leur 20% en libre travail:
  - Robert Griesemer
  - Rob Pike
  - Ken Thompson (un des inventeurs de C)
- ▶ Lancement public le 8 janvier 2010
  - en logiciel libre
  - [golang.org](http://golang.org)
- ▶ Premier langage du 21<sup>ème</sup> siècle

# Le langage Go

- ▶ Go combine
  - Compilation rapide (comme Java)
  - Exécution rapide (comme C/C++)
  - Facilité de programmation (comme Python)
- ▶ Go est un langage avec des types forts et une mémoire protégée
  - Pas possible d'utiliser plusieurs types dans une expression
  - Pas d'arithmétique des pointeurs
  - Avec un ramasse-miette

# Paradigme Go

- ▶ Langage impératif
- ▶ Avec des éléments de programmation concurrente intégrés
  - Programmation réseau
- ▶ Pas orienté-objet
  - Mais avec interfaces, méthodes et polymorphisme
- ▶ Aussi un langage fonctionnel
  - Autorise les fonctions lambda

# Concepts absents

- ▶ Pas de surcharge de fonctions
- ▶ Pas de conversions implicites
- ▶ Pas de classes!
- ▶ Pas de types paramétrisés
- ▶ Pas d'exceptions
- ▶ Pas d'assertions

# Exécution Go

- ▶ Environ 20% plus lent que C
- ▶ 2x plus rapide que Java et 70% moins gourmand en mémoire

# Éléments Go

- ▶ Code facile à lire, structuré en package
  - Dont un package *main*
  - Un package peut avoir une fonction *init()*
  - Utilise le Unicode
  - Sensible à la casse
- ▶ Nombreux concepts orthogonaux
- ▶ 25 mots-clé
- ▶ 36 identificateurs prédéfinis
  - Types et fonctions de base

# Un petit programme Go

```
package main

import "fmt" // importation

func main() {
    fmt.Println("Hello le monde")
}

// débuter par une lettre majuscule
// donne à une fonction une visibilité
// externe au package (exportation)
```

# Types

- ▶ int, int8, int16, int32
- ▶ byte, uint, uint16, uint32, uint64
- ▶ float32, float64
- ▶ complex, complex64, complex128
- ▶ bool
- ▶ String
- ▶ et les pointeurs...

# Les variables et fonctions en Go

```
package main

import (
    "fmt"
)

const pi= 3.1416 // declaration du type optionnel: const pi float= 3.1416
var x int = 5 // variable globale

func main() {
    var ( // declaration en groupe
        a float64 = 8.8
        b float64
    )
    b= fonction(a)
    fmt.Printf("valeur: %f", b)
}

func fonction(z float64) float32 {
    u := 3.3 // declaration initialisee
    return u*z
}
```

# Fonctions à plusieurs retours

```
func main() {  
  
    var s int  
    var d int  
  
    s, d = plusmoins(7,9)  
    fmt.Printf("resulat= %d et %d", s , d)  
  
    for i,j:= 1,5 ; j<100 ; i,j= i+1, j+5 {  
        fmt.Printf("%d et %d", i , j)  
    }  
}  
  
// une fonction peut retourner plus d'une valeur  
func plusmoins(a int, b int) (somme int, difference int) {  
  
    somme= a+b  
    difference= a-b  
  
    return  
}
```

# Fonction avec code d'erreur

```
func imc(taille float64, poids float64)
(float64, bool) {

    if taille > 0.0 {
        return poids / (taille*taille), true
    } else {
        return 0.0, false
    }
}
```

# Test valeur de retour

```
// if initialisaton; condition
if valeur, ok := imc(1.50, 55); ok {

    fmt.Printf("valeur: %f\n", valeur)

}
```

# Fonction en paramètre

```
type Point struct {
    x float64
    y float64
}

func Distance(p1 Point, p2 Point) (distance float64) {

    distance = math.Sqrt(math.Pow(p1.x - p2.x, 2.0) +
        math.Pow(p1.y - p2.y, 2.0))

    return
}

func calc(p1 Point, p2 Point,
         d func(Point, Point)(float64))(float64) {

    return d(p1,p2)
}
```

# Fonction lambda

```
func main() {  
  
    a := Point{2.,4.}  
    b := Point{5.,9.}  
  
    dist := calc(a,b,Distance)  
  
    fmt.Printf("resulat= %f\n", dist)  
  
    dist = calc(a,b,  
        func(p Point, q Point)float64{ // definition d'une fonction lambda  
            return math.Abs(p.x-q.x)+math.Abs(p.y-q.y) })  
  
    fmt.Printf("resulat= %f\n", dist)  
}  
/*  
Affectation:  
d1:= func(p Point, q Point)float64{ return math.Abs(p.x-q.x)+math.Abs(p.y-q.y) }  
Appel:  
func(p Point, q Point)float64{ return math.Abs(p.x-q.x)+  
            math.Abs(p.y-q.y) }(p1,p2)  
*/
```

# Les pointeurs

```
func main() {  
  
    var p *int // pointeur  
    var i int  
    i= 7  
    ptr:= &i  
  
    *ptr= i+5 // déréférence  
  
    fmt.Printf("resulat= %d\n", i)  
}
```

# Construction des structures

```
type Point struct {
    x int
    y int
}

var (
    p1= Point{1,2}
    p2= Point{y:7}
    pp= &Point{3,4} // pointeur a un Point
)

func main() {
    // allocation dynamique
    ptr1:= new(Point)
    ptr2:= &Point{9,8}
}
```

# Pointeurs et structures

```
type Point struct {
    x int
    y int
}

func main() {
    un := Point{8, 1}
    complement(&un)
    fmt.Printf("resultat= %d et %d\n", un.x, un.y)
}

func complement(p *Point) {
    // operateur de de-reference non-requis
    p.x, p.y = -p.y, -p.x
}
```

# Les fabriques

```
type point struct {    // point est maintenant privé
    x int           // les autres packages doivent
    y int           // utiliser la fabrique
}

func main() {
    p1 := NewPoint(1,2)
    fmt.Printf("point= %v\n", p1)
    // point= &{1 2}
}

func NewPoint(i, j int) *point {
    p:= new(point)
    p.x, p.y = i, j
    return p
}
```

# Tableaux en Go

```
package main
import "fmt"

func moyenne(tab [5]int) (moyenne float64) {
    // for index, valeur := range collection
    for _, valeur := range tab {
        moyenne+= (float64)(valeur)
    }
    moyenne /= (float64)(len(tab))
    return
}

func main() {
    // le tableau est un type
    var tableau = [5]int{3, 4, 8, 9, 2}
    m := moyenne(tableau) // passage par valeur
    fmt.Printf("resultat= %f\n", m)
}
```

# Slices en Go

- ▶ Un ‘slice’ est une référence à un segment contigüe d’éléments dans un tableau
- ▶ Les slices sont utilisées plus fréquemment que les tableaux en Go
- ▶ Un slice a une dimension et une capacité
- ▶ Pour les créer:
  - `var slice []int = tableau[start:fin]`
  - `slice := make([]int, 10, 100)`

# Exemple avec slices

```
// tab est une slice
func moyenne(tab []int) (moyenne float64){
    // for index, valeur := range collection
    for _, valeur := range tab {
        moyenne+= (float64)(valeur)
    }
    moyenne /= (float64)(len(tab))
    return
}
func main() {

    var tableau = [5]int{3, 4, 8, 9, 2}

    m := moyenne(tableau[:]) // tous les elements
    fmt.Printf("resultat= %f\n", m)
    m = moyenne(tableau[2:]) // elements 2 a la fin
    fmt.Printf("resultat= %f\n", m)
    m = moyenne(tableau[1:3]) // element 1 a 3 (exclu)
    fmt.Printf("resultat= %f\n", m)
}
```

# Lire le clavier

```
package main

import "fmt"

func main() {
    var nom string

    fmt.Printf("Votre nom? ")
    fmt.Scanf("%s", &nom)
    fmt.Printf("\nBonjour %s\n", nom)
}
```

# Lire et écrire dans un fichier

```
func CopierFichier(dstN, srcN string)
                    (ecrit int64, err error) {
    src, err := os.Open(srcN)
    if err != nil {
        return
    }

    dst, err := os.Create(dstN)
    if err != nil {
        src.Close() // fermer le fichier source
        return      // en cas d'erreur
    }

    écrit, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

# Exécution différée

- ▶ L'énoncé `defer` permet de différer l'exécution d'un bloc jusqu'à la fin de la fonction qui le contient
- ▶ Celui-ci est principalement utilisé pour faire du nettoyage avant de quitter la fonction
- ▶ Les paramètres d'une fonction différée sont évalués lorsque l'énoncé est rencontré
- ▶ L'exécution de la fonction différée est garantie peu importe comment la fonction qui la contient retourne

# Exécution différée

```
func CopierFichier(dstN, srcN string)
                    (ecrit int64, err error) {
    src, err := os.Open(srcN)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstN)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

# Pas d'exceptions en Go

- ▶ Le retour de codes d'erreur remplace les exceptions
- ▶ Lorsqu'une erreur grave se produit, il est possible d'utiliser l'énoncé *panic*
  - À n'utiliser que pour les erreurs imprévues
  - Correspond aux violations d'assertions
- ▶ En cas de panique la fonction s'interrompt immédiatement, les fonctions différentes sont exécutées et retourne à la fonction appelante qui déclenche à son tour une nouvelle panique

# Déclencher une panique

```
func main() {
    fmt.Println("Début")
    var desMots []string
    traite(desMots)
    fmt.Println("Fin")
}

func traite(mots []string) int {

    defer func() {
        fmt.Println("quelques nettoyages")
    }()

    if len(mots) == 0 { // erreur!
        panic("aucun mot!")
    }

    // traitement du tableau de mots...

    return len(mots)
}
```

# Résultat de la panique

```
Command Prompt
C:\Users\louis\Dropbox\CSI2520\go>go run panic.go
Début
quelques nettoyages
panic: aucun mot!

goroutine 1 [running]:
runtime.panic(0x48d600, 0xc084008210)
    C:/Users/ADMINI~1/AppData/Local/Temp/2/makerelease250988475/go/src/pkg/runtime/panic.c:266 +0xc8
main.traite(0x0, 0x0, 0x0, 0x0)
    C:/Users/louis/Dropbox/CSI2520/go/panic.go:22 +0x7f
main.main()
    C:/Users/louis/Dropbox/CSI2520/go/panic.go:9 +0xca
exit status 2
```

# Recouvrer après panique

- ▶ L'énoncé *recover* permet de regagner le contrôle après une panique
- ▶ À utiliser dans une fonction différée
- ▶ En l'absence de panique, le *recover* retourne simplement *nil*

# L'énoncé *recover*

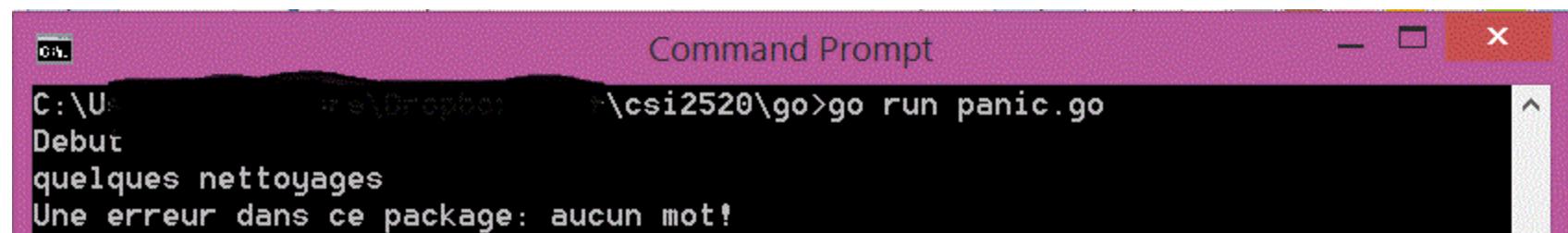
```
func main() {
    fmt.Println("Debut")

    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Une erreur dans ce package: %v", r)
        }
    }()

    var desMots []string
    traite(desMots)

    fmt.Println("Fin")
}
```

# Résultat de *recover*



Command Prompt

```
C:\Users\Debut\Dropbox\csi2520\go>go run panic.go
quelques nettoyages
Une erreur dans ce package: aucun mot!
```

# Méthodes et récepteurs

- ▶ Une méthode est une fonction agissant sur une variable associée à un certain type (ou structure)
  - Le type peut être un pointeur (par référence); obligatoire si la méthode modifie les attributs de l'instance
- ▶ La structure et ses méthodes ne sont pas liés ensemble
  - pas d'encapsulation comme avec les classes
  - Les propriétés et les comportements sont des concepts orthogonaux
- ▶ Elles doivent seulement faire partie du même package

# Définition et appel d'une méthode

```
// structure
type Point struct {
    x float64
    y float64
}

// methode (ici un pointeur afin d'éviter une copie de l'objet)
func (pt *Point) norme() float64 {
    return math.Sqrt(pt.x*pt.x + pt.y*pt.y)
}

func main() {

    a := Point{2.,4.}
    // appel a la methode
    n := a.norme() // traduit en (&a).norme()
    fmt.Printf("resultat= %f\n", n)
    ptr := &Point{2.,4.}
    // appel a la methode
    // il y aurait déréférencement automatique si
    // le récepteur était une valeur
    m := ptr.norme()
    fmt.Printf("resultat= %f\n", m)
}
```

*Go specs: If  $x$  is addressable and  $\&x$ 's method set contains  $m$ ,  $x.m()$  is shorthand for  $(\&x).m()$*

# Encapsulation et exportation

Package point

```
// structure
type Point struct { // le type est exporté
    x float64    // mais pas ses attributs
    y float64
}

// getter
func (pt *Point) GetX() float64 { // méthode exportée
    return pt.x
}
```

# Interfaces

```
// structure avec type embarqué
type PointColore struct {
    Point
    couleur string
}

// une autre structure
type Boite struct {
    poids float64
    couleur string
}

// interface
type Coloreur interface {

    SetCouleur(string)
    Couleur() string
}
```

# Interfaces

- ▶ Un type n'a pas besoin de déclarer explicitement qu'il réalise une interface
  - Il satisfait implicitement l'interface si il possède les méthodes requises
- ▶ Un type peut satisfaire plusieurs interfaces
- ▶ Une interface peut être définie dans un autre package que les types qui la satisfont
- ▶ Une variable interface peut référer à n'importe quelle instance satisfaisant celle-ci

# *Satisfaire une interface*

```
func (p *PointColore) Couleur() string {
    return p.couleur
}

// doit etre pointeur afin de modifier la couleur
func (p *PointColore) SetCouleur(newCouleur string) {
    p.couleur = newCouleur
}

func (p *Boite) SetCouleur(newCouleur string) {
    p.couleur = newCouleur
}

func (p *Boite) Couleur() string {
    return p.couleur
}
```

# Polymorphisme

```
func main() {  
  
    var p Coloreur  
    p= new(PointColore)      // pour être accepté, le type  
    p.SetCouleur("rose")     // doit définir les méthodes  
                            // de l'interface  
    tableau := [4]Coloreur{  
        &PointColore{Point{1.1,2.2}, "rouge"},  
        &Boite{32.4, "jaune"},  
        &PointColore{Point{1.1,2.2}, "bleu"}, p}  
  
    for _,element := range tableau {  
        fmt.Printf("couleur= %s\n",  
                  element.Couleur())  
    }  
}
```

L'utilisation d'une référence (pointeur) est obligatoire car c'est le pointeur à des Boîte qui satisfait l'interface



*Un pointeur peut accéder aux méthodes de son type (déréférencement automatique), mais pas l'inverse*

*Go specs: The method set of any other named type T consists of all methods with receiver type T. The method set of the corresponding pointer type \*T is the set of all methods with receiver \*T or T (that is, it also contains the method set of T).*