FORMATION GO(LANG)

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE (1/2)

- Premiers pas
- Bases du langage
- Types
- Collections
- Structures de contrôle
- Méthodes
- Interfaces

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE (2/2)

- Programmation concurrente
- Testing
- Outils et bonnes pratiques
- Bibliothèque standard
- Compilation
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

LOGISTIQUE

- Horaires
- Déjeuner & pauses
- Autres questions ?

© Copyright 2022 Zenika. All rights reserved

PREMIERS PAS

© Copyright 2022 Zenika. All rights reserved

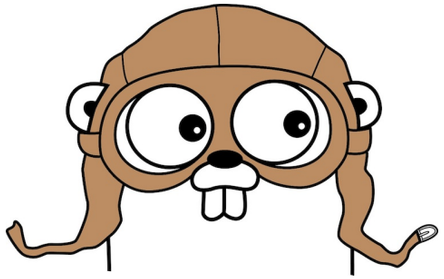
SOMMAIRE

- Présentation de Go(lang)
- Projet Go(lang)
- Installation du SDK
- Premiers pas

© Copyright 2022 Zenika. All rights reserved

POURQUOI UN NOUVEAU LANGAGE ?

- Les développements chez Google sont compliqués, longs, mais ils sont performants
- Il y a des génies chez Google mais tout le monde chez Google n'est pas un génie
- Ne plus avoir à faire le choix entre rapide, facile, et performant



© Copyright 2022 Zenika. All rights reserved

GO(LANG)

> Go is a general-purpose programming language with a clean syntax and advanced features. -- Caleb Doxsey, Introducing Go

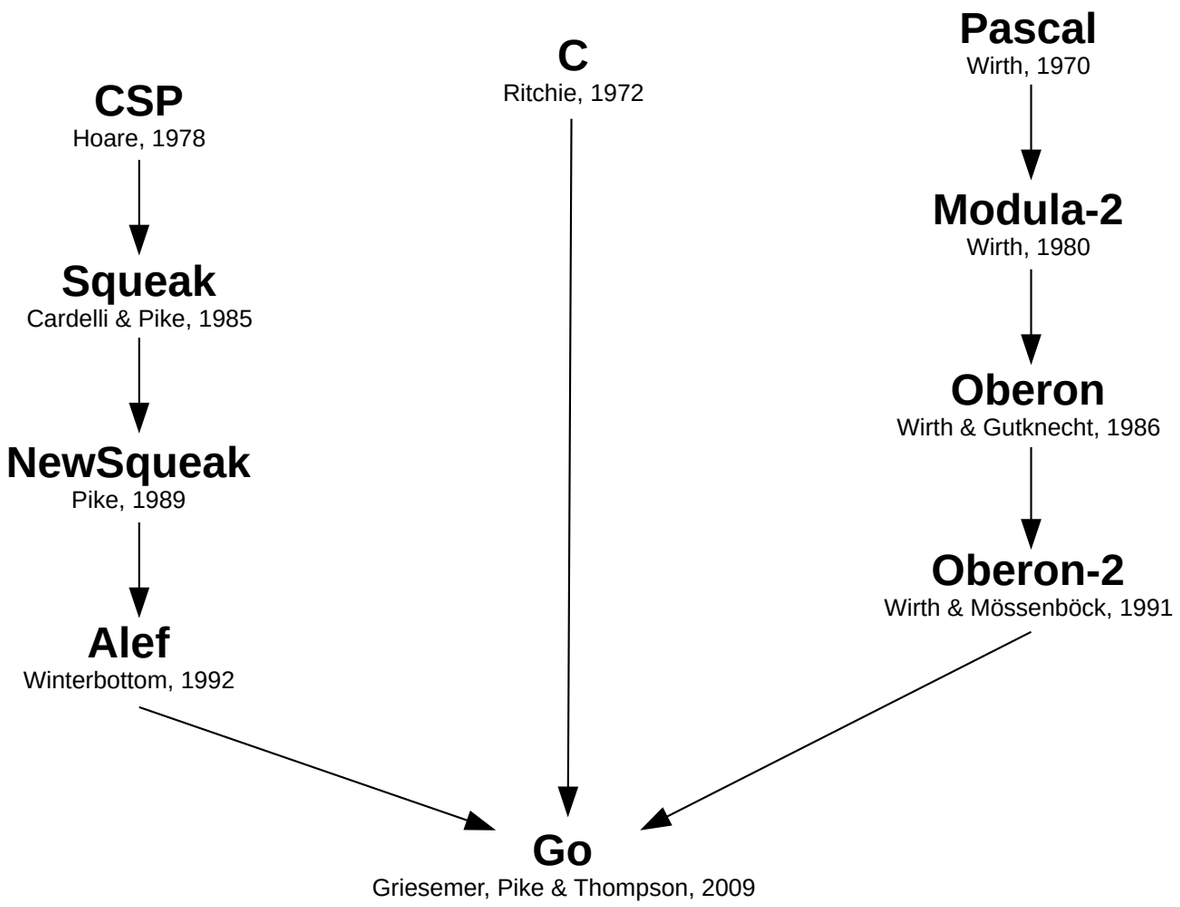
Langage créé chez Google en 2009 par Robert Griesemer, Rob Pike et Ken Thompson.

> Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. -- golang.org

Golang est souvent utilisé pour le différencier du jeu de Go ou de l'outil de pipelining Go.CD

© Copyright 2022 Zenika. All rights reserved

ORIGINES



© Copyright 2022 Zenika. All rights reserved

PHILOSOPHIE

> Complexity is multiplicative. -- Rob Pike

La simplicité est au centre du langage Go.

- Demande plus de travail au début du projet, pour réduire une idée à son essence.
- Demande plus de discipline au cours de la vie du projet pour étudier et accepter un changement et garder l'intégrité conceptuelle de l'application (« Conceptual integrity » — Fred Brooks, The Mythical Man-Month).
- Garantit l'état stable, sécurisé et cohérent de l'application au fil du temps.

© Copyright 2022 Zenika. All rights reserved

CARACTÉRISTIQUES DE GO

- Cherche à être explicite plutôt que concis
- Facile à appréhender
- Compilation rapide vers des binaires natifs
- Nativement adapté à la programmation concurrente
- Librairie standard excellente (pas de gros framework type Spring)

© Copyright 2022 Zenika. All rights reserved

OÙ UTILISE-T'ON GO ? (1/2)

> Go: 90% Perfect, 100% of the time -- Brad Fitzpatrick

- Clients en ligne de commande
- Embarqué (tinygo)
- Webassembly
- Jeu vidéo (ebiten)
- Machine Learning (gorgonia)
- Web (Revel)

© Copyright 2022 Zenika. All rights reserved

OÙ UTILISE-T'ON GO ? (2/2)

Go révèle tout son potentiel dans le contexte du Cloud :

- Docker
- Kubernetes
- Prometheus
- Traefik
- Outils Hashicorp: Terraform, Vault, ...

La compilation en binaires natifs donne des applications qui démarrent très vite et sont économes en mémoire.

Cela permet également d'avoir des images docker "from scratch"/distroless légères (<10Mo), adaptées pour des microservices.

Excellent support dans GCP.

© Copyright 2022 Zenika. All rights reserved

FONCTIONNALITÉS (SURVOL)

> Less is exponentially more. -- Rob Pike

Vous y trouverez

- Garbage collection
- Système de package
- Fonctions « first-class citizen »
- Structures, Interfaces…
- Cross-compilation
- Concurrence

© Copyright 2022 Zenika. All rights reserved

FONCTIONNALITÉS (SURVOL)

Ce que vous ne trouverez pas

- Conversion implicite
- Constructeurs / Destructeurs
- Surcharge d'opérateur
- Héritage
- « Generics »
- Exceptions
- Annotations…

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Présentation de Go(lang)
- Projet Go(lang)
- Installation du SDK
- Premiers pas

© Copyright 2022 Zenika. All rights reserved

PROJET GO(LANG)

Le projet Go(lang) c'est :

- un langage et ses spécifications
- des outils
  - `go ...` compilation, test, analyse, téléchargement de sources
  - `godoc` documentation (pkg.go.dev en ligne)
- un « parcours culturel de simplicité radicale » sous forme de blogs et articles
  - The Go Programming Language (golang.org)
  - go.dev
  - A Tour of Go
  - Effective Go
- un bac à sable (playground) : exécuter du code en ligne.

© Copyright 2022 Zenika. All rights reserved

RELEASES

- Actuellement en version 1.17
- Une release tous les 6 mois
- https://github.com/golang/go
- Contributions via Gerrit
- La version 2.0 est en cours de réflexion (https://github.com/golang/go/wiki/ExperienceReports)

© Copyright 2022 Zenika. All rights reserved

COMMUNAUTÉ

- Plus d'un million de développeurs dans le monde
- Stack Overflow, Reddit, GitHub, Slack
- Awesome Go, Golang Weekly
- PodCast GoTime

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Présentation de Go(lang)
- Projet Go(lang)
- Installation du SDK
- Premiers pas

© Copyright 2022 Zenika. All rights reserved

INSTALLATION

- Binaires officiels : https://golang.org/dl/
- Instructions : https://golang.org/doc/install

- Windows : Installeur ou zip
- Mac OS X : Installeur (`pkg`) ou via Homebrew
- Linux : Packages de votre distribution ou `.tar.gz`

- Vérifier que Go est dans votre `$PATH`

```
$ go version
go version go1.17 linux/amd64
```


© Copyright 2022 Zenika. All rights reserved

Lab 1

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Présentation de Go(lang)
- Projet Go(lang)
- Installation du SDK
- Premiers pas

© Copyright 2022 Zenika. All rights reserved

HELLO, 世界

Créons notre premier programme en Go. Pour changer, nous allons écrire un « Hello, World » … mais avec Unicode 😇.

helloworld.go :

```
package main

import "fmt"

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

Pour l'exécuter, nous allons utiliser la commande `go run`

```
$ go run helloworld.go
Hello, 世界
```


© Copyright 2022 Zenika. All rights reserved

GO TOOLS : RUN & BUILD

`go run` compile et exécute le binaire généré.

Il est aussi possible de le faire en 2 étapes : compiler puis exécuter.

```
$ go build helloworld.go
$ ./helloworld
Hello, 世界
# Pour changer le nom du binaire
$ go build -o hello helloworld.go
$ ./hello
Hello, 世界
```


© Copyright 2022 Zenika. All rights reserved

GO MODULES

Module : Ensemble de packages versionnés ensemble

Création d'un module avec :

```
go mod init github.com/my/module
```

Sert de base aux packages, exemple :

```
github.com/my/module            # package racine
|-- github.com/my/module/pkg1   # sous-package pkg1
|-- github.com/my/module/pkg2   # sous-package pkg2
...
```


© Copyright 2022 Zenika. All rights reserved

GO TOOLS (1/3)

GO GET

Pour ajouter/mettre à jour une dépendance à notre module, nous pouvons utiliser `go get`.

Elle est ajoutée dans le fichier `go.mod` et téléchargée depuis `proxy.golang.org`.

```
$ go get -v github.com/spf13/cobra
go: downloading github.com/spf13/cobra v1.3.0
go: downloading github.com/inconshreveable/mousetrap v1.0.0
go: downloading github.com/spf13/pflag v1.0.5
go: added github.com/inconshreveable/mousetrap v1.0.0
go: added github.com/spf13/cobra v1.3.0
go: added github.com/spf13/pflag v1.0.5
```


© Copyright 2022 Zenika. All rights reserved

GO TOOLS (2/3)

GO INSTALL

Pour télécharger, compiler et installer un outil, nous pouvons utiliser `go install`:

```
$ go install golang.org/x/lint/golint@latest
go: downloading golang.org/x/lint v0.0.0-20210508222113-6edffad5e616
go: downloading golang.org/x/tools v0.0.0-20200130002326-2f3ba24bd6e7
```

L'exécutable est installé dans `$GOPATH/bin` ou `$HOME/go/bin` si la variable d'environnement `GOPATH` n'existe pas.

© Copyright 2022 Zenika. All rights reserved

GO TOOLS (3/3)

GO ENV

`go env` lit les variables d'environnement Go :

```
$ go env
GOARCH="amd64"
GOBIN="/home/user/bin/"
# […]
GOOS="linux"
GOPATH="/home/user/go"
# […]
$
$ go env GOPROXY
https://proxy.golang.org,direct
```


© Copyright 2022 Zenika. All rights reserved

BASES DU LANGAGE

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

PACKAGES (1/5)

Une application Go est composée de différents packages (à minima 1)

- Espace de nommage et sémantique
- Partage de code
- Organisation en arborescence
- Optimisation de la compilation

© Copyright 2022 Zenika. All rights reserved

PACKAGES (2/5)

Un package dispose de *2 désignations*

NOM COURT

Par convention le nom court du package correspond au nom du dossier, exemple : `foo`

Cependant, ce n'est qu'une convention, la seule règle est :

un seul nom de package doit être défini par dossier.

NOM COMPLET

Le nom complet du package est préfixé par le chemin du module, suivi du chemin du package à l'intérieur du module.

Exemple : `github.com/vdemeester/helloworld/foo`

© Copyright 2022 Zenika. All rights reserved

PACKAGES (3/5)

Tous les fichiers Go doivent débuté par la clause `package` avec le nom correspondant à leur package.

📄 `foo/file1.go`

```
package foo

// ...
```

📄 `foo/file2.go`

```
package foo

// ...
```


© Copyright 2022 Zenika. All rights reserved

PACKAGES (4/5)

Pour un binaire, le package *doit* être `main` et contenir une fonction `main`.

```
package main

func main() {
  // Point d'entrée du programme
}
```

Dans chaque fichier, il est possible d'ajouter une ou plusieurs fonctions `init`.

Les fonctions `init` d'un package sont exécutées à l'import de ce package.

```
var arr [10]int
func init() {
  for i := range arr {
    arr[i] = i * i
  }
}
```


© Copyright 2022 Zenika. All rights reserved

PACKAGES (5/5)

Un package interne a un accès restreint :

- Défini par un segment `internal` dans le chemin
- Accessible uniquement à la descendance du répertoire parent d'`internal`

```
foo/bar
foo/bar/internal/util
foo/bar/baz
foo/xyz
```


© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

IMPORTS (1/3)

Permet d'importer un package avec ses fonctions et ses variables visibles.

- l'import se fait via le *nom complet*.

```
import "github.com/docker/libcompose/project"
```

- Plusieurs imports sont possibles, la convention étant de les grouper

```
import (
    "fmt"

    "github.com/docker/libcompose/project"
)
```


© Copyright 2022 Zenika. All rights reserved

IMPORTS (2/3)

L'import se fait via le *nom complet*, mais l'utilisation par le nom court.

```
import "github.com/docker/libcompose/project"

func main() {
    project.NewProject() // […]
}
```

Il est également possible d'appliquer un alias au package importé.

```
import compose "github.com/docker/libcompose/project"

func main() {
    compose.NewProject() // […]
}
```


© Copyright 2022 Zenika. All rights reserved

IMPORTS (3/3)

Les cycles d'imports ne sont pas autorisés par le compilateur.

```
package foo
import "bar"
...
```

```
package bar
import "foo"
...
```

```
can't load package: import cycle not allowed
package formation-go/cycle/foo
    imports formation-go/cycle/bar
    imports formation-go/cycle/foo
import cycle not allowed
package formation-go/cycle/foo
    imports formation-go/cycle/bar
    imports formation-go/cycle/foo
```


© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

VISIBILITÉ : EXPORTS

En Go, il existe 2 types de visibilité pour les éléments d'un package :

- accessible depuis l'extérieur ⇒ exporté
- inaccessible depuis l'extérieur ⇒ non-exporté

> Les identifiers exportés commencent par une lettre en majuscule

```go
package foo

var Bar string = "bar" // foo.Bar

func Baz() {}          // foo.Baz()
func not() {}          // foo.not() ne compile pas
```


© Copyright 2022 Zenika. All rights reserved

Lab 2

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

VARIABLES : DÉCLARATION

Une variable est la déclaration d'un emplacement de stockage avec un type spécifique et un nom associé.

```
// var name type

var s string

var i int
```

On peut grouper des variables

```
var x, y int

var (
  s string
  b bool
)
```


© Copyright 2022 Zenika. All rights reserved

VARIABLES : ASSIGNATION

Il est possible d'assigner une valeur à une variable lors de sa déclaration

```
// var name type = expression

var message string = "Hello World"
```

ou bien plus tard

```
var greeting, who string

greeting, who = "Hello", "World"
fmt.Println(greeting, who)

greeting = "Good bye"
fmt.Println(greeting, who)
```


© Copyright 2022 Zenika. All rights reserved

VARIABLES : INFÉRENCE DE TYPE

Le compilateur Go est capable de déduire le type à partir de la valeur assignée, on parle alors d'inférence de type

```
var message = "Hello World" // message is a string

var i = 42 // i is an int
```

On peut grouper plusieurs variables de types différents

```
var message, i = "Hello World", 42 // message is a string and i is an int
```


© Copyright 2022 Zenika. All rights reserved

VARIABLES : NOTATION COURTE

Une notation courte `:=` existe pour être utilisée à la place de `var`

```
x := 42 // x is an int

x, y := 6, 7 // y is also an int
```

On peut l'utiliser dans les structures de contrôle

```
if err := os.Remove("oldFile.txt"); err != nil {
  fmt.Println("Could not delete file", err)
}
```

⚠️ *Attention ! Quelques gotchas...*

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

CONSTANTES

Une constante est une variable qui ne peut plus changer par la suite.

Les constantes sont évaluées à la compilation et non à l'exécution.

```
const x string = "Hello"
const l = len(x)

func main() {
  const y = "world"
  fmt.Printf("%s %s %d\n", x, y, l) // Affiche "Hello world 5"

  y = "Bye" // erreur de compilation : cannot assign to y
}
```

Les constantes sans type explicite bénéficient d'une précision très élevée.

```
package math

const Pi = 3.14159265358979323846264338327950288419716939937510582097494459
```


© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

FONCTIONS : DÉFINITION ET SYNTAXE

Une fonction est une section de code indépendante qui "mappe" zéro ou plusieurs paramètres en entrée avec zéro ou plusieurs paramètres en sortie.

```
func hello(world string) string {
    return fmt.Sprintf("Hello, %s !", world)
}
```

Une fonction est définie par :

- `func` suivi du nom de la fonction
- les paramètres d'entrée
- le ou les types de retours
- le corps de la fonction entre accolades

© Copyright 2022 Zenika. All rights reserved

FONCTIONS : PARAMÈTRES

Une fonction peut avoir zéro, un ou x paramètres.

```
func main() {} // 0 paramètre

func foo(bar string) {} // 1 paramètre

func bar(foo string, baz int) {} // *x* paramètres

func baz(foo, bar string, evil bool) {} // mutualisation du type
```

Il est également possible de définir des fonctions dites "variadiques", en utilisant `...` avant le type du dernier paramètre.

```
func foo(bar string, baz ...int) {}
// […]
foo("hello")
foo("hello", 1)
foo("hello", 1, 2)
list := []int{1, 2, 3}
foo("hello", list...)
```


© Copyright 2022 Zenika. All rights reserved

FONCTIONS : VALEURS DE RETOURS

Une fonction peut avoir zéro, une ou x valeurs de retour.

```
// 0 valeur de retour
func main() {}
// 1 valeur de retour
func foo() int {}
// x valeurs de retour
func bar() (int, string, error) {}
```

Lors de l'appel, toutes les valeurs doivent être récupérées

```
var f, err = os.Open(name) // os.Open renvoie un fichier et une erreur

// On omet la réponse renvoyée par http.Post
var _, err = http.Post("http://example.com/resource", "application/json", &buf)
```


© Copyright 2022 Zenika. All rights reserved

FONCTIONS : CLOSURES

Les closures sont des fonctions anonymes à l'intérieur d'une autre fonction.

```
func main() {
    hello := func() string {
        return "世界"
    }
    fmt.Printf("Hello, %s\n", hello())
}
```

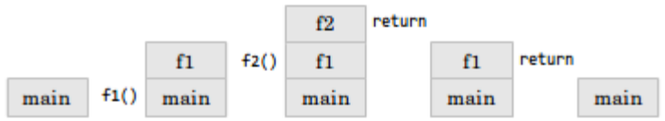
Les fonctions étant des *first class citizens*, il est également possible de retourner une fonction.

```
func helloFactory() func() string {
    return func() string {
        return "世界"
    }
}
func main() {
    helloFn := helloFactory()
    fmt.Printf("Hello, %s\n", helloFn())
}
```


© Copyright 2022 Zenika. All rights reserved

FONCTIONS : PILE D'APPEL

Les fonctions sont intégrées dans une pile d'appel.




```go
func main() {
  fmt.Println(f1())
}
func f1() int {
  return f2()
}
func f2() int {
  return 1
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 3

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

POINTEURS : MUTABILITÉ

En Go, seules les constantes sont immuables.

Les paramètres étant passés par valeur (recopie), ils sont considérés immuables et ne peuvent être modifiés en dehors de la fonction appelée

```
func foo(bar string) string {
    bar = "qwerty"
    return bar
}
func main() {
    var str string = "azerty"
    fmt.Printf("str: %s", foo(str)) // "qwerty"
    fmt.Printf("str: %s", str)      // "azerty"
}
```

Pour modifier un paramètre de fonction, il faut le passer en référence et non en valeur, en utilisant un pointeur

© Copyright 2022 Zenika. All rights reserved

POINTEURS : DÉFINITION

Un pointeur est l'adresse mémoire d'une variable.

Un pointeur permet de lire et modifier la valeur d'une variable indirectement (i.e. sans connaître son nom).

```
x := 1
p := &x          // p, de type *int (pointer to int), pointe vers x
fmt.Println(*p) // "1"
*p = 2           // équivalent à x = 2
fmt.Println(*p) // "2"
fmt.Println(x)  // "2"
```

& permet de récupérer l'adresse d'une variable

* permet de résoudre l'adresse que contient un pointeur

© Copyright 2022 Zenika. All rights reserved

POINTEURS : PASSAGE DE RÉFÉRENCE

Il est possible de passer des pointeurs à une fonction et de retourner un pointeur

Attention aux effets de bords

```
func incr(p *int) int {
    *p++
    return *p
}
v:=1
fmt.Println(incr(&v)) // "2"
fmt.Println(v)  // et v vaut 2
```


© Copyright 2022 Zenika. All rights reserved

POINTEURS : NEW

L'opérateur new permet de créer une variable anonyme de type T

```
p := new(int)   // p, type *int, pointe vers une variable anonyme
fmt.Println(*p) // "0" (valeur à vide de int)
*p = 2          // met à jour la variable anonyme
fmt.Println(*p) // "2"
```


© Copyright 2022 Zenika. All rights reserved

Lab 4

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

PORTÉES : BLOCS

> Go a une portée lexicale en utilisant les blocs
>
> -- Spécifications Go

Il existe plusieurs blocs en Go :

- le bloc univers (`universe block`). Les types, fonctions et constantes de base (`built-in`) sont dans ce bloc.
- le bloc package. Les fonctions et variables déclarées hors d'une fonction sont dans ce bloc, qu'elles soient exportées ou non.
- le bloc fichier (`file block`). Les imports sont dans ce bloc.
- les blocs accolades (fonctions et accolades simples).

© Copyright 2022 Zenika. All rights reserved

PORTÉES : BLOC

Les déclarations n'ont de portée qu'à l'intérieur de leur bloc lexical (et ses enfants), autrement dit des plus proches accolades parentes.

```
// Compile
var x string = "Hello"

func main() {
    y := "world"
    {
        fmt.Printf("%s, %s\n", x, y)
    }
}
// Ne compile pas
func main() {
    {
        y := "Hello, world !"
    }
    fmt.Printf("%s", y)
}
```


© Copyright 2022 Zenika. All rights reserved

PORTÉES : SHADOWING

Attention : un identifiant peut être déclaré plusieurs fois avec le même nom tant que c'est dans deux blocs différents.

Par exemple, il est possible de cacher (shadow) une fonction package-level par une variable locale.

```
// fonction package-level
func f() {}

// variable package-level
var g string = "g"

func main() {
    f := "f"
    // la variable locale f 'cache' la fonction f
    fmt.Println(f) // "f"
}
```


© Copyright 2022 Zenika. All rights reserved

PORTÉE : DURÉE DE VIE

Les variables ont une durée de vie différente selon leurs "scopes" :

- une variable package-level a la même durée de vie que l'exécution de l'application
- une variable locale a une durée de vie dynamique :
  - à chaque exécution du bloc, une nouvelle instance est créée
  - cette variable vit jusqu'à ce qu'elle devienne inaccessible (i.e. plus personne ne la référence)
  - une fois une variable inaccessible, son espace mémoire peut être recyclé (garbage collection).

© Copyright 2022 Zenika. All rights reserved

PORTÉE : ÉCHAPPEMENT (POINTEURS)

Attention: le recyclage de l'espace de mémoire d'une variable ne se fait que lorsque la variable devient inaccessible.

Il est cependant possible qu'une variable locale vive plus longtemps que le bloc dans lequel elle a été déclarée.

```
// x s'échappe de f
var global *int
func f() {
    var x int = 1
    global = &x
}
// y peut être rapidement recyclé
func g() {
    y := new(int)
    *y = 1
}
```

On dit alors que "x s'échappe de f".

© Copyright 2022 Zenika. All rights reserved

SOMMAIRE

- Packages
- Imports
- Visibilité
- Variables
- Constantes
- Fonctions
- Pointeurs
- Portée
- Erreurs

© Copyright 2022 Zenika. All rights reserved

ERREURS : TYPE ERROR

En Go, les erreurs sont remontées par l'intermédiaire du dernier paramètre de sortie des fonctions.

Les erreurs sont de type `error`, et peuvent être comparées à `nil`.

```
func PrintHelloWorld() error {
  var err error
  _, err = fmt.Println("Hello world")
  if err != nil {
    // Une erreur s'est produite
    return err
  }
  return nil
}
```

De nouvelles erreurs peuvent être créées avec `errors.New` ou encore `fmt.Errorf` :

```
err1 := errors.New("something went wrong")
err2 := fmt.Errorf("file %s doesn't exist", f)
```


© Copyright 2022 Zenika. All rights reserved

ERREURS : WRAPPING (1/2)

Les erreurs n'implémentent pas de stacktrace automatique.

Pour donner plus de contexte à une erreur, on la *wrap* dans une autre, et leurs messages sont concaténés.

Une manière simple de wrapper une erreur est d'utiliser la directive `%w` à la fin d'un message dans `fmt.Errorf` :

```
err1 := errors.New("file doesn't exist")
err2 := fmt.Errorf("failed to load configuration: %w", err1)

// err2.String(): "failed to load configuration: file doesn't exist"
```


© Copyright 2022 Zenika. All rights reserved

ERREURS : WRAPPING (2/2)

Les méthodes `errors.Is` et `errors.As` du package `errors` de la bibliothèque standard permettent de remonter la chaîne des erreurs wrappées, ce qui est utile pour trouver la cause d'un problème.

```
var (
    resourceNotFoundError *failure.ResourceNotFoundError
    invalidValueError     *failure.InvalidValueError
)

switch {
case errors.Is(err, resourceNotFoundError):
  return 404
case errors.Is(err, invalidValueError):
  return 400
default:
  return 500
}
```


© Copyright 2022 Zenika. All rights reserved

ERREURS : GESTION DES RESSOURCES

Le mot clé `defer` permet de reporter l'appel d'une fonction jusqu'à la fin de l'exécution de la fonction englobante.

Souvent utilisé pour le nettoyage.

```
func PrintFile(fname string) error {
  f, err := os.Open(fname)
  if err != nil {
    return err
  }
  defer f.Close()
  _, err = io.Copy(os.Stdout, f)
  if err != nil {
    return err
  }
  return nil
}
```

Les appels à `defer` sont empilés et dépilés en LIFO.

© Copyright 2022 Zenika. All rights reserved

ERREURS : PANIC ET RECOVER

L'appel à `panic` interrompt le flot normal d'exécution et dépile la stack d'appels.

Il est possible de reprendre le contrôle en faisant appel à `recover` dans une deferred function.

```
func f() {
    defer fmt.Println("deferred")
    fmt.Println("panic !")
    panic("panic")
}

func main() {
    defer func() {
        r:= recover()
        if r != nil {
            fmt.Println("recovered from", r)
        }
    }()
    f()
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 5

© Copyright 2022 Zenika. All rights reserved

TYPES

© Copyright 2022 Zenika. All rights reserved

TYPES EN GO

Il y a 4 catégories de type en Go

- Type basique : `int`, `float`, `string`, `bool`, `complex`
- Type agrégé : `array`, `struct`
- Type référence : `pointer`, `slice`, `map`, `function`, `channel`
- Type interface : `interface`

Ce chapitre se penche sur les deux premières catégories.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

NOMBRES

3 types de nombres:

- Entiers `int`
- Flottants `float`
- Complexes `complex`

Déclinés en différentes précisions.

Ont tous comme valeur par défaut `0`.

© Copyright 2022 Zenika. All rights reserved

NOMBRES ENTIERS : PRÉCISIONS

Go fournit plusieurs types pour les entiers en fonction de leur type (relatifs `signed` vs naturel `unsigned`) et de leur taille (8, 16, 32 et 64 bits).

- `int8` (-128,127), `uint8` (0,255)
- `int16` (-32768, 32767), `uint16` (0,65535)
- `int32` (-2147483648, 2147483647), `uint32` (0, 4294967295)
- `int64`, `uint64` ("plage" de 2^64)

Ainsi que quelques types supplémentaires.

- `int` et `uint` ayant une précision minimun de 32 bits (varie selon les systèmes/architectures)
- `byte` équivalent à `int8`
- `rune` équivalent à `int32` (utilisé pour représenter un caractère unicode)

© Copyright 2022 Zenika. All rights reserved

NOMBRES ENTIERS : OPÉRATEURS

Il y a niveau de priorité entre les opérateurs binaires, les opérateurs logiques et les opérateurs de comparaison :

1. *, /, %, <<, >>, &, &^
2. +, −, |, ^
3. ==, !=, <, <=, >, >=

```
var x, y = 12, 6

// '+' addition (unary positive)
x + y // "18"

// '%' modulo
x % y // "0"

// '|'  OR
x | y  // "14" (00001100 | 00000110 -> 00001110)

// '*' then '+' then '=='
y + y * x == 78 // "true" (6 + 72 = 78)
```


© Copyright 2022 Zenika. All rights reserved

NOMBRES FLOTTANTS : PRÉCISIONS

Go fournit deux tailles pour les nombres flottants : 32 et 64 bits.

- `float32` : 6 décimales, entre `1.4e-45` et `3.4e38`
- `float64` : 15 décimales, entre `4.9e-324` et `1.8e308`

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d ex = %8.3f\n", x, math.Exp(float64(x)))
}

//    x = 0   ex =    1.000
//    x = 1   ex =    2.718
//    x = 2   ex =    7.389
//    x = 3   ex =   20.086
//    x = 4   ex =   54.598
//    x = 5   ex =  148.413
//    x = 6   ex =  403.429
//    x = 7   ex = 1096.633
```


© Copyright 2022 Zenika. All rights reserved

NOMBRE FLOTTANTS : NAN

Le cas spécial de la représentation Not a Number (NaN), selon la norme IEEE754.

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) //  "0 -0 +Inf -Inf NaN"
```

La fonction `math.IsNaN` permet de tester si un argument n'est pas un nombre, et `math.NaN` retourne la valeur Not a Number.

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
fmt.Println(math.IsNaN(nan)) // true
fmt.Println(math.IsNaN(1/z)) // False - (+∞)
fmt.Println(math.IsInf(1/z, 0)) // True - (+∞)
```

Documentation officiel
Représentation IEEE754-1985

© Copyright 2022 Zenika. All rights reserved

NOMBRES COMPLEXES

Go fournit deux tailles pour les nombres complexes : 64 et 128 bits.

- `complex64`
- `complex128`

La fonction built-in `complex` construit un nombre complexe à partir de ses composants réel et imaginaire.

La fonction `real` permet d'extraire la partie réelle d'un nombre complexe.

La fonction `imag` permet d'extraire la partie imaginaire d'un nombre complexe.

```
var x complex128 = complex(1, 2) // 1+2i
y := 3 + 4i
fmt.Println(x*y)                 // "(-5+10i)"
fmt.Println(real(x*y))           // "-5"
fmt.Println(imag(x*y))           // "10"
```


© Copyright 2022 Zenika. All rights reserved

BOOLÉENS

Un booléen en Go est représenté par le type `bool` et peut avoir comme valeur `true` ou `false`.

La valeur par défaut est `false`.

Opérateurs booléens par ordre de priorité :

- `&&` (`AND`): ET logique
- `||` (`OR`) : OU logique

Les conditions dans les `if` et `for` utilisent les booléens.

Les opérateurs de comparaison `==` et `!=` produisent des booléens.

© Copyright 2022 Zenika. All rights reserved

CHAÎNES DE CARACTÈRES

Une chaîne de caractères est une séquence immuable d'octets.

La valeur par défaut est une chaîne vide `""`.

Les chaînes de caractères peuvent contenir des données arbitraires, y compris un octet de valeur `0`, mais en général elles contiennent des textes lisibles par l'homme et sont classiquement interprétées comme des séquences de code Unicode (UTF8 — `runes`).

La fonction `len` permet d'obtenir le nombre d'octets d'une chaîne de caractères. L'opération index (`s[i]`) permet de récupérer le i-ème octet d'une chaîne `s` (où `0 <= i < len(s)`).

```
s := "hello" + ", world"
fmt.Println(len(s))        // "12"
fmt.Println(s[0], s[7])    // "104 119"  ('h' and 'w')
fmt.Println(s[:5], s[7:]) // "hello" "world"
s = "Hello, worl世"
fmt.Println(len(s))        // "14"
fmt.Println(utf8.RuneCountInString(s)) // "12" (conseillé avant go 1.11)
fmt.Println(len([]rune(s)))       // "12" (depuis go 1.11 aussi optimisé)
```


© Copyright 2022 Zenika. All rights reserved

CHAÎNES DE CARACTÈRES

Go permet d'insérer des caractères spéciaux grâce au caractère d'échappement : `\`.

échappement	caractère inséré
`\a`	“alert” or bell
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\'`	single quote (only in the rune literal `'\''`)
`\"`	double quote (only within `"..."` literals)
`\\`	backslash

© Copyright 2022 Zenika. All rights reserved

CHAÎNES DE CARACTÈRES

Il est également possible de déclarer une chaîne de caractères multi-ligne en utilisant *`*(*backquotes*) à la place de`"`.

```
const GoUsage = `Go is a tool for managing Go source code.

Usage:
    go command [arguments]
...`
```


© Copyright 2022 Zenika. All rights reserved

CHAÎNES DE CARACTÈRES

Quatre packages standards sont particulièrement importants dans l'utilisation des chaînes de caractères.

- `bytes` pour manipuler les chaînes de caractères sous forme de tableau d'octet.
- `strconv` pour effectuer des conversions entre les types de base et les chaînes de caractères (`boolean`, `int`, ...)
- `strings` pour effectuer des opérations standards sur les chaînes de caractères (`Split`, `Join`, `HasPrefix`, ...)
- `unicode` pour manipuler les caractères unicode d'une chaîne de caractères.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

DÉFINITION DE TYPE

Le mot clé `type` permet de définir un nouveau type en le nommant

```
// Definition de types spécifiques pour les températures
type Celsius int
type Fahrenheit int

var nantes Celsius = 25

fmt.Printf("La température à Nantes est de %d°C\n", nantes)
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

CONVERSION DE TYPE

Pour certains types de base, comme les nombres, il est possible de convertir une variable `x` de type `X` vers un type `T` en utilisant l'expression `T(x)`.

Il existe également des fonctions dans la bibliothèque standard permettant la conversion de certains types de base vers d'autres.

Ex: pour la conversion d'une `string` en `int`, on peut utiliser `strconv`.

```go
var i int = 42

// en float64
var f float64 = float64(i)
fmt.Printf("%T: %f\n", f, f) // float64: 42.000000

// en entier non signé
s := string(i)
fmt.Printf("%T: %s\n", s, s) // string: *

s = strconv.Itoa(i)
fmt.Printf("%T: %s\n", s, s) // string: 42

i, _ = strconv.Atoi("-" + s) // string to int
fmt.Printf("%T: %d\n", i, i) // int: -42
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

SWITCH DE TYPE

Le switch de type permet de retrouver le type d'une variable

```
switch x.(type) {  // x est de type interface{}
case int, uint:    // …
case bool:         // …
case string:       // …
default:           // …
}
```

il est utilisé par `fmt.Printf`

```
switch a := b.(type) {
  case int :
    // In this block var a has type int
  case string :
    // In this block var a has type string
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 6

© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

STRUCTURES : DÉFINITION

Une structure en Go est une agrégation permettant de grouper ensemble zéro, une ou plusieurs valeurs (nommées ou non) de n'importe quel type.

Définie par le mot clé `struct`, peut être utilisée comme tout autre type.

L'ordre des champs est important pour l'identité d'une structure.

```
type Event struct {
    Lat, Lon    float64
    Date        time.Time
    Name        string
    description string
}
```


© Copyright 2022 Zenika. All rights reserved

STRUCTURES : INSTANCIATION

Il y a plusieurs méthodes de déclaration et d'assignation d'une structure.

```
// Avec quelques valeurs
var aMeetup = Event{
  Lat: 34.01,
  Lon: -118.49,
  Name: "A very cool Meetup",
}

// … sur une seule ligne, avec tous les champs dans le bon ordre ou 'struct literals'
var red = color.RGBA{255, 0, 0, 0}
```

```
var anotherEmptyMeetup = Event{} // Champs à vide
var anotherPointlessMeetup = &Event{} // … avec un pointeur
```

```
var myMeetup Event // Champs à vide
myMeetup.Name = "My Meetup" // Donnons-lui un nom
```


© Copyright 2022 Zenika. All rights reserved

STRUCTURES : COMPARAISON

Pour comparer des structures, la définition est assez simple : si tous les champs sont comparables, alors la structure est comparable sinon Go retournera une erreur à la compilation.

```
type Point struct { X, Y int }

p := Point{ 1, 2 }
q := Point{ 2, 1 }
r := Point{ 1, 2 }

fmt.Println(p == q)     // false
fmt.Println(p == r)     // true

type nonComparable struct {
    myfunc func() bool // pas comparable
}

v1 := nonComparable{}
v2 := nonComparable{}

v1 == v2
// prog.go:16: invalid operation: v1 == v2 (struct containing [10]func() bool cannot be
compared)
```


© Copyright 2022 Zenika. All rights reserved

Lab 7

© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

STRUCTURES : COMPOSITION

Go n'est pas un langage orienté objet, mais une des possibilités des structures permet de s'en rapprocher : les structures embarquées.

```
type Point struct { X, Y int }

type Circle struct {
    Point
    Radius int
}
```

seul un type nommé peut être embarqué

```
// Un tableau n'est pas un type nommé
type DoesNotCompile struct {
    [10]int
}

// En utilisant la définition de type, on peut ruser
type myArray [10]int
type DoesCompile struct {
    myArray
}
```


© Copyright 2022 Zenika. All rights reserved

COMPOSITION : PROMOTION

Lors d'une composition, les champs sont promus sur la structure parente, quand cela est possible.

```
type Wheel struct {
    Circle
    Spokes int
}

var w Wheel
w.X = 8       // équivalent à w.Circle.Point.X = 8
w.Y = 8       // équivalent à w.Circle.Point.Y = 8
w.Radius = 8  // équivalent à w.Circle.Radius = 8
w.Spokes = 20

w.Circle.Radius = 5
```


© Copyright 2022 Zenika. All rights reserved

COMPOSITION : INSTANCIATION

La création d'une structure possédant des champs anonymes suit les règles vues précédemment.

Il faut cependant faire attention, notamment avec les `struct literals` (déclaration sur une seule ligne).

```
// ne compile pas : unknown fields
w = Wheel{ 8, 8, 5, 20 }
// ne compile pas : unknown fields
w = Wheel{ X: 8, Y: 8, Radius: 5, Spokes: 20 }

// compile
w = Wheel{ Circle{ Point{ 8, 8 }, 5 }, 20}

// Si on utilise la déclaration avec des noms, il faut préciser le nom
// (du type)
w = Wheel {
    Circle: Circle {
        Point: Point{ X: 8, Y: 8 },
        Radius: 5,
    },
    Spokes: 20,
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 8

© Copyright 2022 Zenika. All rights reserved

PLAN

- Types basiques
- Définition de type
- Conversion de type
- Switch de type
- Structures
- Composition
- Alias de type

© Copyright 2022 Zenika. All rights reserved

ALIAS DE TYPE

Depuis Go 1.9 il est possible de créer des alias de type.

Principalement destiné au refactoring de packages existants.

```
type A = B // Créer un alias du type B nommé A
```

À ne pas confondre avec la définition de type.

Un type et son alias peuvent être utilisés indifféremment sans avoir à recourir au transtypage.

© Copyright 2022 Zenika. All rights reserved

COLLECTIONS

© Copyright 2022 Zenika. All rights reserved

PLAN

- Arrays
- Slices
- Maps

© Copyright 2022 Zenika. All rights reserved

ARRAYS

Un tableau (array) est une séquence de zéro ou plusieurs éléments d'un type particulier et de longueur fixe.

```
var x [5]int
x[4] = 100
fmt.Println(x) // [0 0 0 0 100]

x[5] = 10 // invalid array index 5 (out of bounds for 5-element array)
```

Comme dans d'autres langages, les tableaux sont indexés en partant de 0.

```
// Assignation "in line"
a := [3]string{"hello", "world"}
a[2] // "", valeur "à vide" d'une string
// ... pour que le compilateur devine le nombre d'éléments
a := [...]string{"hello", "world", "!"}
```

Il est possible de déclarer un tableau en clé/valeur où la clé est l'index dans le tableau de la valeur.

```
a := [2]string{ 1: "world" } // ["" "world"]
b := [...]int{ 99: 1 }       // [ … 1] tableau de taille 100
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Arrays
- Slices
- Maps

© Copyright 2022 Zenika. All rights reserved

SLICES (1/6)

Un slice est une séquence de longueur variable dont les éléments ont tous le même type.

Un slice est une structure de données légère qui permet d'accéder à une sous-séquence (ou la séquence entière) des éléments d'un tableau (aussi appelé *"tableau interne d'un slice"*).

Un slice est constitué de 3 composants :

- un pointeur vers le premier élément du tableau interne
- une capacité, la taille maximale du slice `cap(…)`.
- une taille correspondant au nombre d'éléments dans le slice `len(…)` et forcément plus petite ou égale à la capacité.

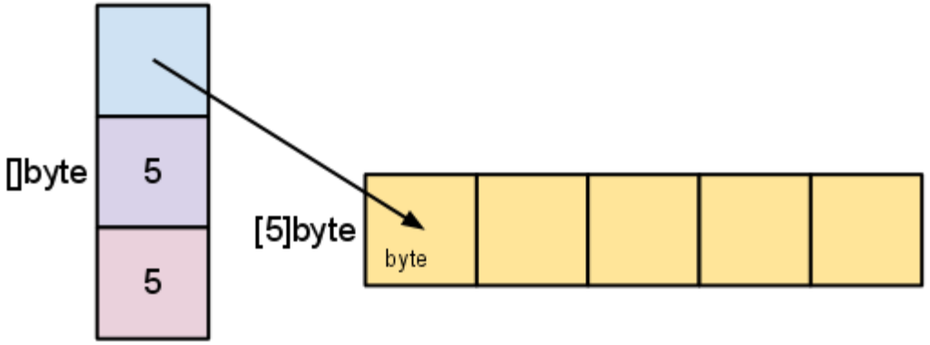
```
// Comme un tableau mais sans taille.
a := []string{}
// Avec make (pour spécifier taille et capacité)
make([]string, 5)     // taille = 5, capacité = 5
make([]string, 5, 10) // taille = 5, capacité = 10
```


© Copyright 2022 Zenika. All rights reserved

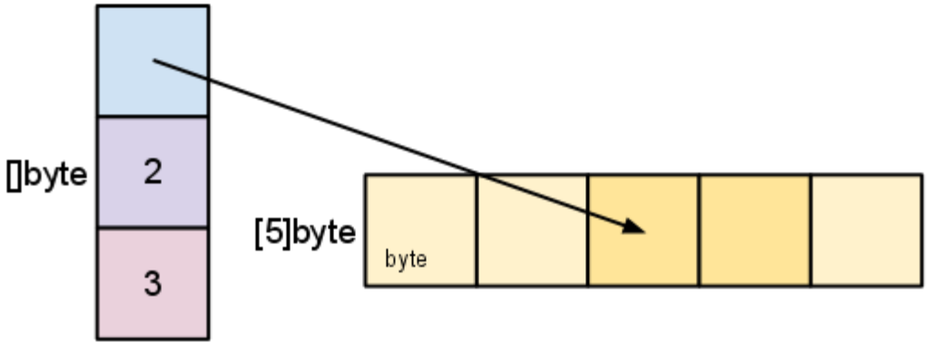
SLICES (2/6)

Plusieurs slices peuvent partager le même tableau interne.

- `s`



- `s[2:4]`



© Copyright 2022 Zenika. All rights reserved

SLICES (3/6)

Le slice operator `slice[i:j]` créé un nouveau slice qui référence les éléments entre `i` et `j-1`, où `0 <= i <= j <= cap(slice)`.

- Si `i` est absent, il vaut `0`
- Si `j` est absent, il vaut `len(slice)`.

```
months := []string{ "Janvier", "Février", "Mars", …, "Décembre" }

Q1 := months[:3]      // ["Janvier", "Février", "Mars"], len:3, cap:12
Q2 := months[3:6]     // ["Avril", "Mai", "Juin"], len:3, cap:9
summer := months[5:8] // ["Juin", "Juillet", "Aout"], len:3, cap:7
Q4 := months[9:]      // ["Octobre", "Novembre", "Décembre"], len:3, cap:3

endlessSummer := summer[:5] // ["Juin", "Juillet", "Aout", "Septembre", "Octobre"]

Q1[:20] // panic: out of range
```


© Copyright 2022 Zenika. All rights reserved

SLICES (4/6)

Un slice n'étant qu'un pointeur vers un tableau, nous pouvons :

- passer un slice en tant que paramètre d'une fonction
- copier un slice
- créer un alias à ce tableau
- modifier ce tableau en utilisant n'importe quel alias.

```
months := []string{ "Janvier", "Février", "Mars", …, "Décembre" }
summer := months[5:9]
fmt.Printf("%s", summer[1]) // Juillet
fmt.Printf("%s", months[6]) // Juillet

summer[1] = "July"
fmt.Printf("%s", summer[1]) // July
fmt.Printf("%s", months[6]) // July
```


© Copyright 2022 Zenika. All rights reserved

SLICES (5/6)

Pour ajouter des éléments à un slice, il est conseillé d'utiliser la fonction 'built-in' append.

Elle prend en paramètre le slice, N éléments du type du slice et renvoie un nouveau slice.

```
a := []string{ "Janvier", "Février", "Mars" }
b := append(a, "Avril")
c := append(b, "Mai")

fmt.Println(len(a), cap(a)) // "3, 3"
fmt.Printf("%q\n", a)       // ["Janvier" "Février" "Mars"]
fmt.Println(len(b), cap(b)) // "4 6"
fmt.Printf("%q\n", b)       // ["Janvier" "Février" "Mars" "Avril"]
fmt.Println(len(c), cap(c)) // "5 6"
fmt.Printf("%q\n", c)       // ["Janvier" "Février" "Mars" "Avril" "Mai"]

// b et c partage le même tableau interne…
fmt.Printf("%q\n", b[:5])   // ["Janvier" "Février" "Mars" "Avril" "Mai"]
// … mais pas a
fmt.Printf("%q\n", a[:5])   // panic : out of range
```


© Copyright 2022 Zenika. All rights reserved

SLICES (6/6)

La fonction append vérifie si le slice peut grandir, c'est-à-dire si sa capacité n'est pas épuisée.

- Si c'est le cas, elle ajoute un élément au tableau interne et renvoie un slice plus grand.
- Sinon elle alloue un nouveau tableau avec une capacité augmentée, recopie les valeurs de l'ancien tableau, et renvoie un nouveau slice qui aura comme tableau interne le nouveau.

Quand Go doit allouer un nouveau tableau, il double la taille du précédent pour éviter trop de réallocation (d'où le passage d'une capacité de 3 à 6).

© Copyright 2022 Zenika. All rights reserved

Lab 9

© Copyright 2022 Zenika. All rights reserved

PLAN

- Arrays
- Slices
- Maps

© Copyright 2022 Zenika. All rights reserved

MAPS (1/3)

- C'est une référence vers une collection de paires clé/valeur non-ordonnée initialisée grâce à la fonction 'built-in' `make`.
- Chaque clé est unique
- Le type des clés doit être comparable via l'opérateur `==`
- À chaque clé correspond une valeur qui peut être récupérée, modifiée ou supprimée.
- Une map est déclarée sous la forme `map[K]V` ou `K` est le type des clés et `V` le type de valeurs.

```go
// Si non initialisé
var x map[string]int
x["clé"] = 10 // runtime error: assignment to entry in nil map

x := make(map[string]int)
x["key1"] = 10
fmt.Println(x) // map[key1:10]
x["key2"] = 1
fmt.Println(x) // map[key1:10 key2:1]
```


© Copyright 2022 Zenika. All rights reserved

MAPS (2/3)

Comme pour les structures, arrays et slices, il est possible de déclarer une map en literals.

```
ages := map[string]int{
    "bob":   29,
    "alice": 31,
}
```

Pour supprimer un élément de la map, on utilise la fonction 'built-in' `delete`.

```
fmt.Println(ages) // map[bob:31 alice:29]
delete(ages, "bob")
fmt.Println(ages) // map[alice:29]
```

Pour tester si une clé existe, on utilise une notation similaire à l'assignation mais côté droit. On parle de *s'inscrire* à une map et cela retourne 2 valeurs : l'élément s'il existe, et un boolean représentant cette existence.

```
age, ok := ages["bob"]
if ok {
    fmt.Println("Bob is in tha house!")
}
```


© Copyright 2022 Zenika. All rights reserved

MAPS (3/3)

Comme pour les slices, les maps ne sont pas comparables entre elles.

Comme pour les slices, une map est une référence vers une table de hashage.

Cela veut dire que la copie d'une map ou le passage en paramètre de cette map fait référence à la même table de hashage.

```go
func foo(m map[string]int) {
    m["bar"] = 100
}

func main() {
    x := make(map[string]int)
    x["foo"] = 10
    fmt.Println(x) // map[foo:10]
    foo(x)
    fmt.Println(x) // map[foo:10 bar:100]
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 10

© Copyright 2022 Zenika. All rights reserved

STRUCTURES DE CONTRÔLE

© Copyright 2022 Zenika. All rights reserved

PLAN

- if
- for
- switch

© Copyright 2022 Zenika. All rights reserved

CONDITION IF

Pour exécuter un bloc en fonction d'une condition, on peut utiliser `if` et son pendant `else`.

La syntaxe du `if` est presque la même qu'en C sauf qu'il n'y a pas de parenthèses englobantes et les accolades sont obligatoires.

```
if i % 2 == 0 {
    // Divisible par 2
} else if i % 3 == 0 {
    // Divisible par 3
} else {
    // Pas divisible par 3 et par 2
}
```

Il est également possible d'utiliser une notation courte pour déclarer une variable qui sera évaluée par la condition.

```
if err := foo(); err != nil {
    fmt.Println("bar")
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- if
- for
- switch

© Copyright 2022 Zenika. All rights reserved

BOUCLES FOR (1/3)

Go possède une seule façon de faire des boucles avec le mot clé `for`.

Comme pour `if`, la déclaration d'une boucle `for` ressemble fortement à une boucle `for` en `C` sans les parenthèses.

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}

// équivalent à une boucle while
sum := 1
for sum < 1000 {
    sum += sum
}

// boucle infinie
for {
    // Forever and ever !
}
```


© Copyright 2022 Zenika. All rights reserved

BOUCLES FOR (2/3)

Il existe une autre façon de déclarer des boucles `for` pour parcourir un tableau, un slice ou une map : `range`.

`range` permet de simplifier le parcours de ces collections puisqu'elle permet d'obtenir :

- l'index et la valeur pour les tableaux et les slices
- la clé et la valeur pour les maps

```
a := []string{ "Janvier", "Février", "Mars", "Avril" }
for index, value := range a {
    fmt.Println(index, value)
}
// Si on ne veut pas l'index, on peut l'ignorer avec _
for _, value := range a {
    fmt.Println(value)
}

m := map[string]int{ "foo":1, "bar": 10, "baz": 100 }
for key, value := range m {
    fmt.Println(key, value)
}
```


© Copyright 2022 Zenika. All rights reserved

BOUCLES FOR (3/3)

Support de `continue`, `break` et des labels.

```
func FizzBuzz(n int) {
    i := 0
loop:
    for {
        i++
        if i == n {
            break loop
        }
        s := ""
        if i%3 == 0 {
            s += "fizz"
        }
        if i%5 == 0 {
            s += "buzz"
        }
        if len(s) > 0 {
            fmt.Println(s)
            continue loop
        }
        fmt.Println(i)
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- if
- for
- switch

© Copyright 2022 Zenika. All rights reserved

SWITCH (1/2)

Comme beaucoup de langages de programmation, Go dispose d'une instruction `switch` permettant d'éviter d'écrire une longue et complexe série de `if`/`else`.

- Le comportement par défaut (`default`) est facultatif.
- Il est possible d'utiliser une expression dans la déclaration d'un cas (`case`), par exemple, pour calculer une valeur.
- Pour passer d'un cas au suivant (les chaîner), il faut utiliser le mot clé `fallthrough`.

```
score := 4
switch score {
case 1, 2, 3, 4, 5:
    fmt.Println("Meh !")
case 5 - 5:
    fmt.Println("Pire que tout !")
default:
    fmt.Println("Tricheur !")
}
```


© Copyright 2022 Zenika. All rights reserved

SWITCH (2/2)

`switch` supporte une forme alternative, sans valeur en paramètre, avec des `case` booléens.

```
score := 7
switch {
case score > 6:
    fmt.Println("Bravo !")
case score > 3:
    fmt.Println("Pas mal !")
default:
    fmt.Println("Bof !")
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 11

© Copyright 2022 Zenika. All rights reserved

MÉTHODES

© Copyright 2022 Zenika. All rights reserved

PLAN

- Object oriented "like"
- Composition
- Encapsulation

© Copyright 2022 Zenika. All rights reserved

MÉTHODES : RÉCEPTEUR

Bien que techniquement Go ne soit pas un langage orienté objet, les structures, méthodes et interfaces permettent d'avoir un style orienté objet.

En orienté objet, on dit qu'une méthode est une fonction sur une instance d'un objet.

Go n'ayant pas de classes, les méthodes se définissent directement sur les types. On dit que les valeurs sont des récepteurs d'une méthode.

```go
type User struct {
    FirstName, LastName string
}

func (u User) Greeting() string {
    return fmt.Sprintf("Bonjour %s %s", u.FirstName, u.LastName)
}

func main() {
    u := User{"John", "Doe"}
    fmt.Println(u.Greeting()) // "Bonjour John Doe"
}
```


© Copyright 2022 Zenika. All rights reserved

MÉTHODES : RÉCEPTEUR POINTEUR

Pour rappel, lors de l'appel d'une fonction, Go passe les paramètres par valeur (par copie).

Si l'on souhaite modifier une variable ou si un argument est trop coûteux à copier, il est possible de passer des paramètres par référence.

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}

p1 = &Point{ 1, 2 }
p1.ScaleBy(2)
p2 = Point{ 1, 2 }
// Marche également, le compilateur passera `&p` implicitement
p2.ScaleBy(2)
```

Une convention existe en Go : si au moins une des méthodes d'une structure possède un pointeur comme récepteur alors toutes les méthodes doivent faire de même.

© Copyright 2022 Zenika. All rights reserved

MÉTHODES : RÉCEPTEUR VIDE

Comme pour les paramètres des fonctions qui peuvent accepter `nil`, il est possible d'utiliser `nil` comme récepteur d'une méthode.

Pour rappel, `nil` correspond souvent à une "valeur à vide" d'un type.

```go
type IntList struct {
    Value int
    Tail *IntList
}

func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }
    return list.Value + list.Tail.Sum()
}

var l IntList
fmt.Println(l)       // {0 <nil>}
fmt.Println(l.Sum()) // "0"
```


© Copyright 2022 Zenika. All rights reserved

VALEURS ET EXPRESSIONS DE MÉTHODE

Une méthode peut être assignée à une variable. On dit alors que la variable a une valeur de méthode (method value).

```
type Rocket struct { /* … */ }
func (r *Rocket) Launch() { /* … */ }

r := new(Rocket)
time.AfterFunc(10 * time.Second, r.Launch)
```

Les expressions de méthode (method expressions) sont une notion liée aux valeurs de méthode. Une expression de méthode, écrite `T.f` ou `(*T).f`, rend une fonction qui prend les mêmes paramètres que la méthode `f` avec comme premier paramètre le récepteur (i.e une instance de `T`).

```
p := Point{1, 2}
q := Point{4, 6}
distance := Point.Distance
fmt.Println(distance(p, q))  // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Object oriented "like"*
- Composition
- Encapsulation

© Copyright 2022 Zenika. All rights reserved

MÉTHODES : COMPOSITION

Nous avons déjà parlé un peu de composition avec les types embarqués permettant d'accéder aux champs de ces types de la même manière que s'ils étaient déclarés dans le type courant.

Le même principe s'applique avec les méthodes.

```
type Point struct { X, Y int }
type ColoredPoint struct {
    Point
    color string
}

func (p Point) Distance() {
    // […]
}

p := Point{ 1, 2 }
cp := ColoredPoint{ Point{ 1, 2 }, "red"}

p.Distance()
cp.Point.Distance()
cp.Distance()
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Object oriented "like"*
- Composition
- Encapsulation

© Copyright 2022 Zenika. All rights reserved

MÉTHODES : ENCAPSULATION (1/2)

L'encapsulation consiste à cacher les attributs d'une classe aux autres classes, c'est-à-dire empêcher l'accès aux données par un autre moyen que l'utilisation de méthodes.

En Go, l'encapsulation se traduit, pour les structures, par le mécanisme de champ exporté ou non. Cela implique que l'unité d'encapsulation est le package et non le type (très différent des autres langages).

L'encapsulation offre trois avantages :

- Le client ne peut pas modifier directement les variables d'un objet ou d'une structure.
- Cacher les détails d'implémentation donne une plus grande liberté d'évolution au concepteur du code en empêchant le client de s'appuyer sur des choses qui pourraient changer.
- Permet d'empêcher le client de setter des attributs de l'objet ou la structure de manière arbitraire.

© Copyright 2022 Zenika. All rights reserved

MÉTHODES : ENCAPSULATION (2/2)

Un exemple d'encapsulation avec le package `log`.

```
package log

type Logger struct {
    flags  int
    prefix string
    // ...
}

func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

Un client de ce package `log` ne peut que créer un `Log` et utiliser des méthodes dessus, mais n'a aucun accès à la structure interne de `Log`.

© Copyright 2022 Zenika. All rights reserved

Lab 12

© Copyright 2022 Zenika. All rights reserved

INTERFACES

© Copyright 2022 Zenika. All rights reserved

PLAN

- Anatomie d'une interface
- Le type interface
- Implémentation implicite (interface satisfaction)
- Valeur d'interface

© Copyright 2022 Zenika. All rights reserved

ANATOMIE D'UNE INTERFACE (1/2)

Une interface exprime des généralisations ou des abstractions sur le comportement d'un type. Cela permet d'écrire des fonctions plus souples et adaptables car non-liées aux détails d'implémentation.

On retrouve principalement le concept d'interface dans les langages orientés objet. Les interfaces en Go ont une particularité : elles sont *implémentées de manière implicite*.

En d'autres termes, un type `T` implémente l'interface `I` dès lors qu'il possède les méthodes nécessaires (même signature, etc.). Cela permet par exemple de créer des interfaces pour des types déjà existants.

© Copyright 2022 Zenika. All rights reserved

ANATOMIE D'UNE INTERFACE (2/2)

Jusqu'à maintenant nous n'avons vu que des types concrets, i.e. qui spécifient la représentation exacte de ces valeurs (`struct` et types primitifs) et méthodes.

On sait *ce qu'est* le type et *ce qu'il fait*.

Il existe cependant d'autres sortes de types : les « type interface » (interface type). Un type interface est un type abstrait : il ne spécifie pas la représentation exacte de ces valeurs, uniquement les méthodes.

On sait uniquement *ce que le type fait*.

Une interface définit un *contrat* entre une implémentation et ses appelants.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Anatomie d'une interface
- Le type interface
- Implémentation implicite (interface satisfaction)
- Valeur d'interface

© Copyright 2022 Zenika. All rights reserved

LE TYPE INTERFACE

Un type interface spécifie un ensemble de méthodes qu'un type concret doit définir pour pouvoir être considéré comme une instance de cette interface.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Comme pour les structures, il est possible de composer une ou plusieurs interfaces dans une autre.

```
type ReadCloser interface {
    Reader
    Closer
}

// Same as
type ReadCloser interface {
    Reader
    Close() error
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Anatomie d'une interface
- Le type interface
- Implémentation implicite (interface satisfaction)
- Valeur d'interface

© Copyright 2022 Zenika. All rights reserved

IMPLÉMENTATION IMPLICITE (1/3)

Un type implémente (ou satisfait) une interface dès lors qu'il possède toutes les méthodes que l'interface définit.

Par exemple, `os.File` satisfait `io.Reader`, `io.Writer`, `io.Closer` et donc également `io.ReadWriter` ou `ReadCloser` définis ci-dessus.

Une expression peut être assignée à une interface seulement si son type satisfait l'interface.

```
var w io.Writer
w = os.Stdout           // OK: *os.File has Write method
w = new(bytes.Buffer)   // OK: *bytes.Buffer has Write method
w = time.Second         // compile error: time.Duration lacks Write method
```

```
var w io.Writer = os.Stdout
w.Write([]byte("hello"))    // OK: io.Writer has Write method
w.Close()                   // compile error: io.Writer lacks Close method
```


© Copyright 2022 Zenika. All rights reserved

IMPLÉMENTATION IMPLICITE (2/3)

Il existe quelques subtilités par rapport à l'implémentation implicite et la façon dont les méthodes d'un type sont déclarées.

Rappel : une méthode peut avoir comme récepteur un type T ou un pointeur de ce type *T.

```
type IntSet struct { /* */ }
func (*IntSet) String() string // satisfait fmt.Stringer

var _ fmt.Stringer = &IntSet{} // OK
var _ fmt.Stringer = IntSet{}  // compile error
//cannot use IntSet literal (type IntSet) as type fmt.Stringer in assignment:
//    IntSet does not implement fmt.Stringer (String method has pointer receiver)
```


© Copyright 2022 Zenika. All rights reserved

IMPLÉMENTATION IMPLICITE (3/3)

De par leur nature, il n'est pas nécessaire de déclarer une relation entre un type concret et l'interface qu'il satisfait.

Cela dit, il est parfois utile de documenter et d'affirmer la relation entre les deux, quand cette relation est attendue mais pas forcée dans le reste du programme.

```
// *bytes.Buffer must satisfy io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

Il n'est pas nécessaire d'allouer une nouvelle variable puisque n'importe quelle valeur du type `bytes.Buffer` fera l'affaire (même `nil`). De même, comme nous n'allons pas utiliser la variable que l'on crée, on le remplace par l'identificateur vide, `_`.

Si `bytes.Buffer` ne satisfait pas l'interface `io.Writer`, alors nous aurons une erreur de compilation.

© Copyright 2022 Zenika. All rights reserved

Lab 13

© Copyright 2022 Zenika. All rights reserved

PLAN

- Anatomie d'une interface
- Le type interface
- Implémentation implicite (interface satisfaction)
- Valeur d'interface

© Copyright 2022 Zenika. All rights reserved

INTERFACE VIDE

Une interface peut définir zéro méthode. Ce type d'interface est appelée interface vide. C'est une interface qui ne décrit aucun comportement ou contrat.

Un interface vide satisfait *tous* les types concrets. Elle permet donc à des méthodes comme `fmt.Println` de prendre des arguments de n'importe quel type.

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

Pour pouvoir utiliser une valeur de type `interface{}` il est nécessaire d'avoir recours à l'assertion de type.

© Copyright 2022 Zenika. All rights reserved

VALEUR D'INTERFACE

Une valeur de type interface, ou encore appelée valeur d'interface, a deux composants:

- un type concret : dynamic type
- une valeur de ce type : dynamic value

La valeur à vide d'une interface est `nil`. Appeler une méthode sur une interface à vide provoque un `panic`.

Deux interfaces sont comparables avec `==` et `!=` ; deux interfaces sont considérées comme égales si elles ont toutes les deux comme valeur dynamique `nil`, ou si elles ont le même type dynamique et la même valeur dynamique.

© Copyright 2022 Zenika. All rights reserved

ASSERTION DE TYPE (1/2)

Une assertion de type est une opération applicable à une valeur de type interface.

La syntaxe est la suivante : `x.(T)` où `x` est l'expression d'un type interface et où `T` un type.

Elle vérifie si le type dynamique de l'opérande (`x`) correspond au type précisé entre parenthèse (`T`).

```
var w io.Writer = os.Stdout
rw := w.(io.ReadWriter) // rw is of type ReadWriter
rw = w.(bytes.Buffer)  // panic:
```

Si une assertion de type apparaît dans une assignation qui attends 2 valeurs de retour, alors elle ne `panic` pas et retourne un booléen.

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File)      // success:  ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // failure: !ok, b == nil
```


© Copyright 2022 Zenika. All rights reserved

ASSERTION DE TYPE (2/2)

- Si T est un type concret, alors l'assertion de type vérifie que le type dynamique de x est identique à T.

```
var w io.Writer = os.Stdout
f := w.(*os.File)      // success: f == os.Stdout
c := w.(*bytes.Buffer) // panic: interface holds *os.File, not *bytes.Buffer
```

- Si T est un type interface, alors l'assertion de type vérifie que le type dynamique de x satisfait l'interface T.

```
var w io.Writer = os.Stdout
rw := w.(io.ReadWriter) // success: *os.File has both Read and Write
w = new(ByteCounter)
rw = w.(io.ReadWriter) // panic: *ByteCounter has no Read method
```


© Copyright 2022 Zenika. All rights reserved

CONSEIL SUR L'UTILISATION DES INTERFACES (1/2)

Utilisation non idiomatique des interfaces

Lors de la conception d'un nouveau package, les développeurs Go novices ont tendance à créer un ensemble d'interfaces — surtout s'ils ont un background de language orienté objet — et seulement ensuite satisfaire ces interfaces par des types concrets.

Cette approche se traduit par beaucoup d'interfaces et trop d'abstractions inutiles, sans compter que l'utilisation des interfaces a un coût à l'exécution.

© Copyright 2022 Zenika. All rights reserved

CONSEIL SUR L'UTILISATION DES INTERFACES (2/2)

Utilisation idiomatique des interfaces

Les packages permettent de restreindre les structures et champs visibles. *Les interfaces sont seulement nécessaires quand il y a deux ou plusieurs types concrets qui doivent être traités de manière uniforme*.

Un autre usage peut être si le type qui doit satisfaire l'interface est dans un package différent pour des raisons de dépendance — cela permet de découpler deux paquets.

La plupart des interfaces en Go sont relativement petites (une ou deux méthodes), car simples à satisfaire et à composer.

© Copyright 2022 Zenika. All rights reserved

GENERICS

© Copyright 2022 Zenika. All rights reserved

PLAN

- Nécessité des generics
- Contraintes
- Fonctions génériques
- Types génériques
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

NÉCESSITÉ DES GENERICS

Sans generics les paramètres dans les signatures de fonctions doivent être typés explicitement :

```
func ReverseStrings(list []string) {
    var length = len(list)
    for i := 0; i < length/2; i++ {
        list[i], list[length-1-i] = list[length-1-i], list[i]
    }
}

var l = []string{"a", "b", "c"}

ReversStrings(l)

fmt.Println(l) // [c b a]
```


© Copyright 2022 Zenika. All rights reserved

NÉCESSITÉ DES GENERICS

Les generics permettent de créer des fonctions dont la signature contient des types indéterminés :

```
func ReverseAnything[T any](list []T) {
    var length = len(list)
    for i := 0; i < length/2; i++ {
        list[i], list[length-1-i] = list[length-1-i], list[i]
    }
}

var l = []string{"a", "b", "c"}

ReverseAnything(l)

fmt.Println(l) // [c b a]
```


© Copyright 2022 Zenika. All rights reserved

LE TYPE ANY

Introduit en même temps que les generics, le type `any` est un alias du type `interface{}` :

```
type any = interface{}
```


© Copyright 2022 Zenika. All rights reserved

LE TYPE ANY

L'utilisation du type any ne permet pas de réaliser n'importe quel traitement générique :

```
func Join[T any](list []T) (result string) {
  for _, v := range list {
    result += v.String() // v.String undefined (type T has no field or method String)
  }
  return result
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Nécessité des generics
- Contraintes
- Fonctions génériques
- Types génériques
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : MÉTHODES

L'application de contraintes sur les arguments de type en utilisant les interfaces permet de réaliser des traitements génériques nécessitant la présence de méthodes :

```
func Join[T fmt.Stringer](list []T) (result string) {
  for _, v := range list {
    result += v.String()
  }
  return result
}
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : UNIONS

Les contraintes sur la présence de méthodes ne sont parfois pas appropriées pour certains traitements génériques :

```
func Sum[T any](list []T) T {
  var total T
  for _, v := range list {
    total += v // invalid operation: operator + not defined on total
  }
  return total
}
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : UNIONS

Les unions de types permettent d'écrire des traitements génériques pouvant fonctionner pour un ensemble prédéfini de types :

```
func Sum[T Number](list []T) T {
  var total T
  for _, v := range list {
    total += v
  }
  return total
}

type Number interface {
  int | int8 | int16 | int32 | int64 |
  uint | uint8 | uint16 | uint32 | uint64 |
  float32 | float64
}
```

Seuls les types concrets ou les types interfaces ne contenant pas de méthodes sont autorisés dans les unions.

© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : APPROXIMATIONS

Par défaut, les contraintes par unions de types n'acceptent pas les types dérivés :

```
func Mean[T Number](list []T) T {
    return Sum(list) / T(len(list))
}

type Number interface {
  int | int8 | int16 | int32 | int64 |
  uint | uint8 | uint16 | uint32 | uint64 |
  float32 | float64
}

type Celsius float32

fmt.Println(Mean([]Celsius{3.5, 6, 10, 12.5})) // Celsius does not implement Number
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : APPROXIMATIONS

L'utilisation d'approximations de type permet d'accepter les types dérivés :

```
func Mean[T Number](list []T) T {
    return Sum(list) / T(len(list))
}

type Number interface {
  ~int | ~int8 | ~int16 | ~int32 | ~int64 |
  ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 |
  ~float32 | ~float64
}

type Celsius float32

fmt.Println(Mean([]Celsius{3.5, 6, 10, 12.5})) // 8
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : COMPARABLE

L'interface comparable du package builtin de la bibliothèque standard permet d'écrire des traitements génériques nécessitant l'utilisation des opérateurs d'égalité == != :

```
func IndexOf[T comparable](haystack []T, needle T) int {
    for i, v := range haystack {
        if v == needle {
            return i
        }
    }
    return -1
}

fmt.Println(IndexOf([]string{"foo", "bar", "baz"}, "bar")) // 1
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES : CONSTRAINTS.ORDERED

L'interface `Ordered` du package `golang.org/x/exp/constraints` permet d'écrire des traitements génériques nécessitant l'utilisation des opérateurs de comparaison `< <= >= >` :

```
func Max[T constraints.Ordered](list []T) T {
    var result = list[0]
    for _, v := range list[1:] {
        if v > result {
            result = v
        }
    }
    return result
}

fmt.Println(Max([]int{42, 0, 86})) // 86
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Nécessité des generics
- Contraintes
- Fonctions génériques
- Types génériques
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

FONCTIONS GÉNÉRIQUES

Les fonctions génériques peuvent déclarer un ou plusieurs paramètres de type, qui peuvent être utiliser dans leurs signatures :

```
func Map[T, R any](src []T, mapper func(T) R) []R {
    var dst = make([]R, 0, len(src))
    for _, v := range src {
        dst = append(dst, mapper(v))
    }
    return dst
}
```


© Copyright 2022 Zenika. All rights reserved

FONCTIONS GÉNÉRIQUES : INSTANCIATION

Lors de la compilation à chaque appel d'une fonction générique, celle-ci est instanciée avec les types utilisés lors de l'appel :

```
fmt.Println(Map([]int{1, 2, 3}, strconv.Itoa)) // ["1", "2", "3"]
// Instancie func Map(src []int, mapper func(int) string) []string

fmt.Println(Map([]int{4, 5, 6}, strconv.Itoa)) // ["4", "5", "6"]
// Pas de nouvelle instanciation

fmt.Println(Map([]Celsius{14.5, 37}, func (v Celsius) string {
  return fmt.Sprintf("%.2f°C", v)
})) // ["14.50°C", "37.00°C"]
// Instancie func Map(src []Celsius, mapper func(Celsius) string) []string
```


© Copyright 2022 Zenika. All rights reserved

FONCTIONS GÉNÉRIQUES : INSTANCIATION

Il est possible d'instancier explicitement une fonction générique :

```
Map[int, string]([]int{1, 2, 3}, strconv.Itoa)
```

Cela est parfois nécessaire dans certains cas :

```
func Identity[T any](v T) T {
  return v
}

var list = []string{"foo", "bar", "baz"}

Map(list, Identity) // cannot use generic function Identity without instantiation

Map(list, Identity[string]) // compiles
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Nécessité des generics
- Contraintes
- Fonctions génériques
- Types génériques
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

TYPES GÉNÉRIQUES

Il est également possible de créer des types génériques :

```
type Set[T comparable] map[T]struct{}

var s = make(Set[string])

s["foo"] = struct{}{}
s["foo"] = struct{}{}
s["bar"] = struct{}{}

fmt.Println(len(s)) // 2
```


© Copyright 2022 Zenika. All rights reserved

TYPES GÉNÉRIQUES : MÉTHODES

Des méthodes ayant pour récepteur un type générique peuvent être déclarées en utilisant les paramètres du type dans leurs signatures :

```
func (s Set[T]) Add(v T) {
  s[v] = struct{}{}
}

func (s Set[T]) Has(v T) bool {
  var _, ok = s[v]
  return ok
}

var s = make(Set[string])

s.Add("foo")

fmt.Println(s.Has("foo")) // true
fmt.Println(s.Has("bar")) // false
```


© Copyright 2022 Zenika. All rights reserved

TYPES GÉNÉRIQUES : MÉTHODES

L'ajout de paramètres de type directement sur une méthode n'est pas autorisé :

```
// syntax error: method must have no type parameters
func (src Set[T]) Map[R comparable](mapper func(T) R) Set[R] {
  // ...
}
```

Il faut passer par une fonction :

```
func MapSet[T, R comparable](src Set[T], mapper func(T) R) Set[R] {
  var dst = make(Set[R], len(src))
  for v := range src {
    dst.Add(mapper(v))
  }
  return dst
}
```


© Copyright 2022 Zenika. All rights reserved

TYPES GÉNÉRIQUES : INSTANCIATION

Au même titre que les fonctions, les types génériques passent par la phase d'instanciation lors de la compilation :

```
type IntSet = Set[int]

var i = make(IntSet)

i.Add(6)

fmt.Println(i.Has(6)) // true
fmt.Println(i.Has(0)) // false
```


© Copyright 2022 Zenika. All rights reserved

Lab Generics

© Copyright 2022 Zenika. All rights reserved

PLAN

- Nécessité des generics
- Contraintes
- Fonctions génériques
- Types génériques
- Pour aller plus loin

© Copyright 2022 Zenika. All rights reserved

POUR ALLER PLUS LOIN

Les packages `golang.org/x/exp/slices` et `golang.org/x/exp/maps` fournissent des fonctions pour manipuler des slices et des maps contenant des éléments de n'importe quel type.

© Copyright 2022 Zenika. All rights reserved

PROGRAMMATION CONCURRENTE

© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

PROGRAMMATION CONCURRENTE

La programmation concurrente est la décomposition d'un programme en plusieurs activités autonomes qui peuvent être exécutées de manière parallèle.

Les problématiques nécessitant l'utilisation de programmation concurrente sont nombreuses et de plus en plus importantes.

Go permet deux styles de programmation concurrente :

- `CSP` (communicating sequential processes), où les valeurs sont communiquées entre chaque activité mais où les variables sont confinées à une seule activité. Avec Go, nous utilisons les `goroutines` et les `channels`.

- Shared memory multithreading, un modèle plus traditionnel où les variables sont partagées entre les différentes activités et l'utilisation de `locks` sera nécessaire.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

GOROUTINES (1/2)

En Go, chaque activité exécutée de manière concurrente est appelée `goroutine`.

Une `goroutine` est très similaire à un thread système.

Quand un programme démarre, la seule goroutine qui est présente est celle qui appelle la fonction `main`. Une nouvelle `goroutine` est créée en utilisant le mot clé `go`.

```
f()    // appelle f() et attend qu'elle termine
go f() // crée une goroutine et appelle f() dedans
```

Lorsque la fonction `main` se termine, toutes les goroutines sont terminées de manière abrupte. Il n'est pas possible pour une goroutine de stopper une autre goroutine.

© Copyright 2022 Zenika. All rights reserved

GOROUTINES (2/2)

Voici un exemple d'utilisation de goroutines qui affiche un spinner le temps que la fonction `main` se termine.

```
func main() {
    go spinner(100 * time.Millisecond)
    time.Sleep(4 * time.Second)
    fmt.Printf("Slept 4 seconds")
}

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 14

© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

CHANNELS (1/2)

Si l'on considère les goroutines comme les activités d'un programme concurrent, alors les channels représentent la connexion entre ces activités.

Un channel est un mécanisme de communication qui permet à une goroutine d'envoyer des valeurs à une autre goroutine.

Chaque channel est un tuyau pour les valeurs d'un type particulier.

```
ch := make(chan int) // le type de ch est `chan int`
```

Comme pour les maps, un channel est une référence à la structure de données créé par `make`.

La valeur à vide est `nil`. Deux channels sont comparables avec l'opérateur `==`, et cette comparaison renvoie vrai si les deux références pointent vers la même structure de données.

© Copyright 2022 Zenika. All rights reserved

CHANNELS (2/2)

Un channel a deux opérations principales :

- Envoi (`send`) : envoie une valeur à d'une goroutine à l'autre à travers le channel. L'opérateur à utiliser est `<-` entre le channel et la valeur à envoyer.
- Réception (`receive`) : attend une valeur au travers du channel et peut l'assigner à une variable. L'opérateur est `<-` avant le channel et possiblement après l'assignation.

```
ch <- x  // envoi de x à travers le channel

x = <-ch // réception d'une valeur du channel ch et assignation à x
<-ch     // réception d'une valeur du channel ch, sans assignation
```

La troisième opération possible est la fermeture d'un channel (`close`). Une fois un channel fermé, les communications ne sont plus possibles ; et un envoi sur un channel fermé provoque un `panic`.

© Copyright 2022 Zenika. All rights reserved

UNBUFFERED CHANNELS

Un unbuffered channel est un channel qui *bloque la goroutine qui effectue l'envoi tant qu'une autre goroutine n'a pas réceptionné la valeur* du channel.

Une fois la réception effectuée, les goroutines continuent leurs vies et le channel peut être réutilisé.

Ainsi les unbuffered channels permettent de synchroniser deux goroutines, on les appelle donc parfois des channels synchrones.

```go
func main() {
    done := make(chan struct{})
    go func() {
        fmt.Println("doing some work !")
        time.Sleep(2 * time.Second)
        done <- struct{}{}
    }()
    fmt.Println("waiting...")
    <- done // on attend que la goroutine est finie
}
```


© Copyright 2022 Zenika. All rights reserved

BUFFERED CHANNELS (1/2)

Un buffered channel est un channel qui dispose d'une queue d'éléments. Il dispose d'un type et d'une capacité (que l'on précise en argument de `make`). Cette capacité représente la taille de la file d'attente.

L'opération d'envoi vers un buffered channel insert un élément à la fin de la file d'attente, et l'opération de réception enlève un élément au début de cette dernière.

L'opération d'envoi ne bloque que si la file d'attente est pleine, et ce jusqu'à ce qu'un élément se libère. L'opération de réception bloque en attendant une valeur.

```
ch := make(chan string, 3) // file d'attente à 3 places
ch <- "A"
ch <- "B"
ch <- "C"
fmt.Println(<-ch) // "A"

// cap indique la capacité
fmt.Println(cap(ch)) // 3
// len indique le nombre d'éléments dans la file
fmt.Println(len(ch)) // 2
```


© Copyright 2022 Zenika. All rights reserved

BUFFERED CHANNELS (2/2)

L'objectif est d'éviter d'envoyer et de réceptionner des éléments d'un buffered channel dans la même goroutine car il est très facile de se retrouver dans une situation de bloquage.

```
ch := make(chan string, 2)
ch <- "A"
ch <- "B"
ch <- "C" // bloque le programme car attend que la file se "dépile"
fmt.Println(<-ch)
```

Exemple de cas d'utilisation pour un buffered channel :

```
func mirroredQuery() string {
    responses := make(chan string, 3)
    go func() { responses <- request("asia.gopl.io") }()
    go func() { responses <- request("europe.gopl.io") }()
    go func() { responses <- request("americas.gopl.io") }()
    return <-responses // return the quickest response
}
func request(hostname string) (response string) { /* ... */ }
```


© Copyright 2022 Zenika. All rights reserved

PIPELINES (1/2)

Les channels peuvent également être utilisés pour connecter deux goroutines afin que la sortie (`output`) de l'une soit l'entrée (`input`) de l'autre. C'est ce qu'on appelle le pipelining.

```
func main() {
    done := make(chan struct{})
    naturals := make(chan int)
    go func() {
        for x := 0; x < 10 ; x++ {
            naturals <- x
        }
        done <- struct{}{}
    }()
    go func() {
        for {
            y := <- naturals
            fmt.Println(y)
        }
    }()
    <-done
}
```


© Copyright 2022 Zenika. All rights reserved

PIPELINES (2/2)

Il est alors pratique d'utiliser la fonction `close` pour indiquer quand une des goroutines du pipeline a terminé, en fermant le channel.

Il faut alors faire un check lors de la réception du channel pour savoir si celui est fermé ou non.

```
go func() {
    for {
        y, open := <- naturals
        if !open {
            done <- struct{}{}
            break // le channel a été fermé et "drainé" (plus rien dedans)
        }
        fmt.Println(y)
    }
}()
```

Il est également possible d'utiliser l'opérateur `range` pour une syntaxe moins verbeuse.

```
go func() {
    for y := range naturals { // pas de <- et de check de validité
        fmt.Println(y)
    }
    done <- struct{}{}
}()
```


© Copyright 2022 Zenika. All rights reserved

CHANNELS UNIDIRECTIONNELS

Lorsqu'on utilise les pipelines, on se retrouve assez souvent avec une goroutine qui a un channel pour l'entrée (`input`) et un pour la sortie (`output`). Il y a donc un channel où l'on ne va faire que recevoir, et l'un que l'on va utiliser que pour envoyer.

Il est très courant, lorsqu'on passe un channel en argument, qu'il soit utilisé uniquement dans un sens (réception ou envoi).

Il est possible et même recommandé de documenter cette intention. On parle alors de channel unidirectionnel :

- `chan<- int` est un channel "envoi" (`send-only`)
- `<-chan int` est un channel "réception" (`receive-only`).

```
func printIt(out chan<- int, in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
    close(out)
}
```


© Copyright 2022 Zenika. All rights reserved

SYNC.WAITGROUP

Le type `WaitGroup` du package `sync` permet d'attendre la fin de l'exécution de plusieurs goroutines.

`WaitGroup.Add()` doit être appelé avant le lancement des goroutines.

```
var wg sync.WaitGroup

func main() {
  wg.Add(2)
  go sleepFor(2 * time.Second) // This will take some time...
  go sleepFor(5 * time.Second) // This will take even more time...

  wg.Wait()
  fmt.Println("All done !")
}

func sleepFor(someTime time.Duration) {
  time.Sleep(someTime)
  wg.Done()
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 15

© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

MULTIPLEXAGE (1/2)

Il est parfois nécessaire d'attendre qu'une valeur arrive sur plusieurs channels, et réagir alors à celle-ci. Il faut alors faire du multiplexage de ces opérations, et avec Go, il faut utiliser l'instruction `select`.

```
select {
case <- ch1:
    // ...
case x := <- ch2:
    // ... use x...
case ch3 <- y:
    // ... passes there if a goroutine receive a value from ch3
default:
    // ...
}
```

Chaque cas (`case`) correspond à une communication, et chaque expression peut être assignée pour utilisation à l'intérieur du bloc.

`select` attend jusqu'à ce qu'une communication d'un des cas se passe et exécute le bloc. Toutes les autres communications ne se produisent alors pas.

© Copyright 2022 Zenika. All rights reserved

MULTIPLEXAGE (2/2)

Grâce au multiplexage il est possible de mettre des deadlines sur l'exécution d'une fonction.

```
func main() {
    done := make(chan struct{})

    go func() {
        /* Do something long enough... */
        done <- struct{}{}
    }()

    select {
    case <- time.After(10 * time.Second):
        fmt.Println("Something took to long, I quit!")
        return
    case <-done:
        fmt.Println("It's done! See ya!")
        return
    }
}
```

À noter que si plusieurs cas (`case`) sont prêts, `select` en prend un au hasard.

© Copyright 2022 Zenika. All rights reserved

Lab 16

© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

VARIABLES PARTAGÉES

Il n'est parfois pas possible ou souhaitable d'utiliser les channels de communications entre goroutines. Il faut alors travailler avec des variables partagées et toutes les problématiques qui les accompagnent.

Lors de l'utilisation de variables partagées il est important de mettre des gardes-fou sur ces dernières pour éviter de se retrouver avec des incohérences de données. Pour cela, on utilise des formes plus ou moins évoluées de verrouillage : `mutex`, moniteurs, sémaphores.

Il faut alors faire bien attention à leur utilisation car ils peuvent apporter deux problèmes majeurs:

- interblocages (`deadlocks`), i.e. deux activités (`goroutines`) qui sont en attente l'une de l'autres.
- famines (`race conditions`)

© Copyright 2022 Zenika. All rights reserved

RACE CONDITIONS (1/2)

Une race condition est une situation dans laquelle le programme ne donne pas le résultat correct dans certaines conditions (entrelacements) d'exécution de plusieurs goroutines.

Les race conditions sont pernicieuses car elles peuvent rester latentes dans le programme et n'apparaitre que très rarement et sont donc difficiles à reproduire et diagnostiquer.

```go
package bank

var balance int
func Deposit(amount int) { balance = balance + amount }
func Balance() int { return balance }
```

```go
// Alice:
go func() {
    bank.Deposit(200)                // A1
    fmt.Println("=", bank.Balance()) // A2
}()
// Bob:
go bank.Deposit(100)                 // B
```


© Copyright 2022 Zenika. All rights reserved

RACE CONDITIONS (2/2)

Cas et résultats possibles.

```
Alice first                Bob first          Alice/Bob/Alice
          0                        0                        0
  A1    200                B     100                A1    200
  A2 "= 200"               A1    300                B     300
  B     300                A2 "= 300"               A2 "= 300"
```

Il existe cependant une race condition : si le dépôt de Bob se passe au milieu du dépôt d'Alice, mais après que la balance ait été lue. Le résultat final est alors faux.

```
Data race
          0
A1r       0            ... = balance + amount
B       100
A1w     200            balance = ...
A2   "= 200"
```

Avec des types plus complexes (des structures), cela peut aller jusqu'à corrompre la mémoire.

© Copyright 2022 Zenika. All rights reserved

RACE CONDITIONS : RACE DETECTOR

Même avec le plus grand soin, il est très facile de faire des erreurs et de créer des race conditions avec la programmation concurrente. Heureusement, Go et sa suite d'outils sont équipés d'un outil d'analyse dynamique et facile à utiliser permettant de détecter ces race conditions. On parle de race detector.

Pour l'utiliser, il suffit d'ajouter l'argument `-race` à vous commandes `go build`, `go run` ou encore `go test`.

```
$ go run -race deposits.go
==================
WARNING: DATA RACE
Write at 0x0000011d8e78 by goroutine 7:
  race/bank.Deposit()
      /Users/eric/go/src/race/bank/bank.go:5 +0x3d

Previous write at 0x0000011d8e78 by goroutine 6:
  main.main.func1()
      /Users/eric/go/src/race/bank/bank.go:5 +0x41

Goroutine 7 (running) created at:
  main.main()
      /Users/eric/go/src/race/deposits.go:16 +0x63

Goroutine 6 (finished) created at:
  main.main()
      /Users/eric/go/src/race/deposits.go:11 +0x42
==================
Found 1 data race(s)
exit status 66
```


© Copyright 2022 Zenika. All rights reserved

EXCLUSION MUTUELLE

Il est possible d'utiliser un channel de capacité 1 pour s'assurer qu'au plus une goroutine accède à une variable partagée à un temps t. On appelle cela exclusion mutuelle, et on parle alors de sémaphore binaire.

```go
var (
    sema    = make(chan struct{}, 1) // a binary semaphore guarding balance
    balance int
)

func Deposit(amount int) {
    sema <- struct{}{} // acquire token
    balance = balance + amount
    <-sema // release token
}

func Balance() int {
    sema <- struct{}{} // acquire token
    b := balance
    <-sema // release token
    return b
}
```


© Copyright 2022 Zenika. All rights reserved

EXCLUSION MUTUELLE : SYNC.MUTEX

Ce pattern est tellement utile et utilisé, qu'il est supporté nativement avec le type `Mutex` du package `sync`.

Ce type dispose de deux méthodes : `Lock` et `Unlock`.

```
var (
    mu      sync.Mutex // guards balance
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}
```


© Copyright 2022 Zenika. All rights reserved

READ/WRITE MUTEX

Il existe un cas assez fréquent où la lecture d'une variable partagée par plusieurs goroutines en parallèle ne pose pas de problèmes, mais où il faut bloquer dans le cas d'une écriture.

Le package `sync` fournit un type pour ce cas là : `RWMutex`, qui fournit deux méthodes en plus : `RLock` et `RUnlock`.

```go
var (
    mu      sync.RWMutex // guards balance
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.RLock()
    b := balance
    mu.RUnlock()
    return b
}
```


© Copyright 2022 Zenika. All rights reserved

LAZY INITIALIZATION - SYNC.ONCE

Dans le cas d'étape d'initialisation chère (i.e. d'une structure), il est d'usage ou de bonne pratique, de reporter celle-ci au moment où elle est nécessaire. On parle alors de lazy initialization.

Le package `sync` fournit un type qui permet de faire cela, `Once` ; qui est une spécialisation d'un `RWMutex` pour une seule écriture.

```
var loadIconsOnce sync.Once
var icons map[string]image.Image

// Concurrency-safe.
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons)
    return icons[name]
}

func loadIcons() {
    icons = make(map[string]image.Image)
    icons["spades.png"] = loadIcon("spades.png")
    icons["hearts.png"] = loadIcon("hearts.png")
    icons["diamonds.png"] = loadIcon("diamonds.png")
    icons["clubs.png"] = loadIcon("clubs.png")
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Programmation concurrente
- Goroutines
- Channels
- Select (multiplexing)
- Variables partagées
- Context

© Copyright 2022 Zenika. All rights reserved

CONTEXT : PRÉSENTATION

Le package `context` a été ajouté en go 1.7.

Il offre une manière unifiée de prévenir une opération asynchrone qu'elle peut s'arrêter. Cela évite d'avoir des "fuites" de goroutines.

Pour cela, un contexte porte un channel `Done()`, qui est fermé lorsque le contexte termine.

Il est possible de former une arborescence de contextes : un contexte enfant ne se terminera pas après son parent.

© Copyright 2022 Zenika. All rights reserved

CONTEXT : CANCELLATION

`context.WithCancel` permet de créer un contexte enfant qui se terminera soit quand on appellera une fonction `cancel`, soit quand son contexte parent se terminera, selon ce qui arrive en premier.

```
childCtx, cancel := context.WithCancel(parentCtx)
defer cancel()
```

Il faut appeler cancel dès que l'on peut, cela permet de libérer les ressources liées à `childCtx`.

© Copyright 2022 Zenika. All rights reserved

CONTEXT : DEADLINE, TIMEOUT

On peut créer un contexte qui se terminera au plus tard à un instant donné avec `WithDeadline` :

```
deadline := time.Date(2020, 1, 1, 1, 1, 1, 0, time.Local)
childCtx, cancel := context.WithDeadline(parentCtx, deadline)
defer cancel()
```

Ou de manière relative à l'instant présent avec `WithTimeout` :

```
childCtx, cancel := context.WithTimeout(parentCtx, 2*time.Second)
defer cancel()
```


© Copyright 2022 Zenika. All rights reserved

CONTEXT : EXEMPLES

Par convention, le paramètre `ctx` est toujours le premier.

- Dans le package cloud.google.com/go/datastore :

```
func (c *Client) Count(ctx context.Context, q *Query) (n int, err error)
```

- Dans le package net/http :

```
func NewRequestWithContext(ctx context.Context, method, url string, body io.Reader)
(*Request, error)
```

- Dans un serveur HTTP, chaque requête reçue est accompagnée d'un contexte qui s'achève au plus tard quand la connexion s'achève.

```
func (r *Request) Context() context.Context
```


© Copyright 2022 Zenika. All rights reserved

CONTEXTES PARTICULIERS

- `context.Background()` retourne un contexte qui ne finit jamais
- `context.TODO()` retourne de même un contexte qui ne finit jamais, mais est à utiliser lorsqu'on ne sait pas encore quel contexte choisir

© Copyright 2022 Zenika. All rights reserved

CONTEXT : WITHVALUE()

Les contextes ont une autre fonctionnalité : porter des valeurs dont le cycle de vie est lié à celui du contexte.

C'est utile par exemple pour des informations de tracing.

```
childCtx := context.WithValue(parentCtx, "traceID", "1234")
```

Attention, il ne faut pas en abuser, et surtout pas pour passer des paramètres type ID d'utilisateur !

© Copyright 2022 Zenika. All rights reserved

Lab 17bis

© Copyright 2022 Zenika. All rights reserved

TESTING

© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

TESTS (1/6)

Programmer n'est pas une tâche facile; même les meilleurs développeurs sont incapables d'écrire des programmes qui fonctionnent exactement comme prévu à chaque fois. Par conséquent, une partie importante du processus de développement logiciel est la phase de test.

Écrire des tests pour notre code est un bon moyen d'assurer la qualité et d'améliorer la fiabilité.

© Copyright 2022 Zenika. All rights reserved

TESTS (2/6)

Pour la plupart des langages, il existe beaucoup de littérature et de frameworks sur le sujet des tests.

L'approche de Go est assez minimaliste en comparaison, mais a l'avantage d'être incluse dans la bibliothèque standard (ce qui n'est pas le cas dans beaucoup de langages). Cette approche minimaliste permet de se focaliser sur le test et non sur l'utilisation du framework de test; et inclut la possibilité d'écrire des tests de performances (benchmarks) et des exemples compilés / testés.

© Copyright 2022 Zenika. All rights reserved

TESTS (3/6)

Les tests en Go s'appuient sur un ensemble de conventions et sur le package `testing`.

- Les tests sont lancés grâce à la commande `go test`
- Un fichier de test a un nom qui termine par `_test.go` et appartient au même package que la fonction, structure ou méthode testée.
- Un test est une fonction; son nom doit commencer par `Test` et prendre un argument de type `*testing.T`.

```go
func TestName(t *testing.T) {
    // […]
}

func TestAnotherName(t *testing.T) {
    // […]
}

// I'm not seen >_<
func testSomething(t *testing.T) {
    // […]
}
```


© Copyright 2022 Zenika. All rights reserved

TESTS (4/6)

Le type `testing.T` dispose des méthodes principales suivantes :

- `Skip`, pour ignorer un test :

```
func TestSomething(t *testing.T) {
    if runtime.GOOS == "windows" {
        t.Skip("Needs porting to Windows")
    }
}
```

- `Fail` et `FailNow`, pour marquer le test en erreur. `Fail` continue l'exécution du test jusqu'au bout, `FailNow` l'arrête.

```
func TestAnotherSomething(t *testing.T) {
    fmt.Println("hello")
    t.Fail()
    fmt.Println("world")
}
func TestSomething(t *testing.T) {
    fmt.Println("hello")
    t.FailNow()
    fmt.Println("not executed")
}
```


© Copyright 2022 Zenika. All rights reserved

TESTS (5/6)

- `Log`, `LogF`, pour avoir des logs qui ne seront affichés que si le test est en erreur ou que les tests sont lancés en mode verbose (`-v`).
- `Error`, `Errorf`, pour marquer le test en erreur, avec un message (contrairement à `Fail`) tout en continuant l'exécution du test.
- `Fatal`, `Fatalf`, pour marquer le test en erreur et s'arrêter de suite, avec un message (contrairement à `FailNow`).

```go
func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    t.Log("You see me")
}

func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Fatal(`IsPalindrome("palindrome") = true`)
    }
    t.Log("You might not see me")
}
```


© Copyright 2022 Zenika. All rights reserved

TESTS (6/6)

Il est possible d'écrire des packages de tests externes

- Nom de package suffixé par `_test`
- Utile pour éviter les cycles d'imports
- Pas d'accès aux éléments non exportés du package testé

📄 `exports_test.go` :

```
package anagram
var IsAnagram = isAnagram
```

📄 `anagram_test.go` :

```
package anagram_test
import (
  "anagram"
  "testing"
)
func TestIsAnagram(t *testing.T) {
  if !anagram.IsAnagram("listen", "silent") {
    ...
  }
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

TABLE DRIVEN TESTS (1/2)

Lorsque l'on veut tester le comportement d'une fonction avec plusieurs inputs différents, il est courant de définir ses cas de tests dans une slice ou une hashmap, puis d'itérer dessus pour exécuter les tests.

Cela évite de réécrire plusieurs fois le même code de test et met en valeur ce qui change d'un cas de test à l'autre.

Cette approche est nommée Table Driven Tests.

© Copyright 2022 Zenika. All rights reserved

TABLE DRIVEN TESTS (2/2)

```go
func TestIsPalindrome(t *testing.T) {
  tests := map[string]struct{
    input    string
    expected bool
  }{
    "empty string":     { input: "", expected: true },
    "palindrome":       { input: "madam", expected: true },
    "not a palindrome": { input: "sir", expected: false },
  }

  for name, test := range tests {
    t.Run(func(t *testing.T) {
      if actual := isPalindrome(test.input); actual != test.expected {
        t.Errorf("unexpected: wanted [%v], got [%v]", test.expected, actual)
      }
    })
  }
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 17

© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

COUVERTURE DE CODE

De par leur nature, les tests ne sont jamais réellement complets.

> Les tests montrent la présence et non l'absence d'un bug
>
> -- Edsger Dijkstra

La couverture de test (`coverage`) est une mesure de quelles parties du code sont exercées (ou exécutées) par une suite de test.

Go supporte nativement différentes mesures de couverture de code avec :

- Les flags `-cover`, `-coverprofile` et `-covermode` de `go test` :

```
$ go test -coverprofile=c.out -covermode=count monpackage
ok      package          0.032s  coverage: 68.5% of statements
```

- L'outil `go tool cover` pour, traiter et générer des rapports :

```
$ go tool cover -html=c.out
```


© Copyright 2022 Zenika. All rights reserved

Lab 18

© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

TESTS DE PERFORMANCE (1/2)

Les tests de performance, comme leur nom l'indique, permettent de mesurer les performances d'un programme (ou d'une fonction) avec une charge de travail fixe.

Go supporte nativement l'écriture des tests de performances, via le type `testing.B` (pour `Benchmark`).

Comme pour les tests, par convention, le nom d'un test de performance doit commencer par `Benchmark` et prendre un argument de type `*testing.B`.

Le type `testing.B` offre les même méthodes que `testing.T` avec en plus d'autres liées à la mesure de performance.

```
func BenchmarkIsPalindrome(b *testing.B) {
        for i := 0; i < b.N; i++ {
                IsPalindrome("A man, a plan, a canal: Panama")
        }
}
```


© Copyright 2022 Zenika. All rights reserved

TESTS DE PERFORMANCE (2/2)

Le flag `-bench <regexp>` de l'outil `go test` active l'execution des tests de performance.

Les autres flags supportés sont `-benchmem` et `-benchtime <t>`

```
$ go test -bench=. -benchmem # -benchmem pour inclure des mesures de mémoire
PASS
BenchmarkIsPalindrome-8 2000000       697 ns/op       256 B/op       5 allocs/op
ok      monpackage              1.468s
```


© Copyright 2022 Zenika. All rights reserved

Lab 19

© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

PROFILING (1/2)

Les tests de performance sont utiles pour mesurer la performance d'une opération spécifique, mais lorsqu'il s'agit de rendre un programme lent plus rapide, il est difficile de savoir par où commencer.

La méthode de mesure qui permet d'identifier les parties critiques du code, sur lesquelles les gains de performance seraient plus importants, s'appelle profiling.

Go supporte plusieurs types de profiling :

- CPU profile : permet d'identifier les déclarations (`statements`) qui demandent plus de temps CPU.
- Heap profile : permet d'identifier les déclarations qui allouent le plus de mémoire.
- Blocking profile : permet d'identifier les opérations qui bloquent le plus longtemps les goroutines.

© Copyright 2022 Zenika. All rights reserved

PROFILING (2/2)

Les flags `-cpuprofile`, `-memprofile` et `-blockprofile` de l'outil `go test` activent ces types de profiling.

Bien qu'il soit possible de lancer la commande avec ces trois flags activés en même temps, il est conseillé de faire un type de profiling à la fois.

Un fois le profiling terminé on peut l'analyser avec l'outil pprof (`go tool pprof`).

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 -cpuprofile=cpu.log net/http
# […]
$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
# […]
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Tests
- Table Driven Tests
- Couverture de code
- Tests de performance
- Profiling
- Fonctions exemple

© Copyright 2022 Zenika. All rights reserved

FONCTIONS EXEMPLE

Les fonctions exemple sont des fonctions de documentation qui sont traitées de manière spécifique par `go test`. Le nom d'une fonction exemple commence par `Example`, et ne prend aucun paramètre.

- Elles servent de documentation vivante, car compilées par `go test`.
- Elles sont visibles dans la documentation (via `godoc`) à coté de la fonction testée.
- Si un commentaire `// Output :` est présent, `go test` exécute l'exemple et vérifie si la sortie standard correspond à la suite du commentaire.

```go
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
    fmt.Println(IsPalindrome("palindrome"))
    // Output:
    // true
    // false
}
```


© Copyright 2022 Zenika. All rights reserved

Lab 20

© Copyright 2022 Zenika. All rights reserved

OUTILS ET BONNES PRATIQUES

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck` (`golint` est déprécié)
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

FORMATTER VOS SOURCES AVEC GOFMT (1/2)

`gofmt` est un utilitaire qui formate du code écrit en Go. Il utilise les conventions de Go, c'est à dire les tabulations pour l'indentation et les espaces pour les alignements.

Un code passé au crible de `gofmt` est :

- Plus facile à écrire : vous ne pouvez plus vous soucier de problèmes de formatage,
- Plus facile à lire : quand tout code se ressemble vous évitez de faire des conversions mentales du code d'autrui en un style que vous pouvez comprendre,
- Plus facile à maintenir : des modifications mécaniques à la source ne causent pas de changements liés à la mise en forme du fichier; i.e. les `diffs` ne montrent que les changements réels,
- Sans controverse : jamais un débat sur l'espacement ou la position, plus jamais !

© Copyright 2022 Zenika. All rights reserved

FORMATTER VOS SOURCES AVEC GOFMT (2/2)

Un exemple de fonction passée au crible de `gofmt`.

```
 func (f *Filesystem) Files() []*File {
-   if len(f.files)<1 {f.captureFiles()}
+   if len(f.files) < 1 {
+       f.captureFiles()
+   }
    return f.files
 }
```

Avec l'option `-s` il est possible de simplifier certaines partie de vos sources.

```
-           final = append(final, first[start:len(first)]...)
+           final = append(final, first[start:]...)
```

Avec l'option `-r` il est possible de faire des refactoring simples. Un exemple avec la commande `gofmt -r '(a) -> a' -d *.go`.

```
    for i, line := range rstLines {
        if strings.HasPrefix(line, "<body>") {
-           rstLines = (rstLines[i+1 : len(rstLines)-3])
+           rstLines = rstLines[i+1 : len(rstLines)-3]
        }
    }
```


© Copyright 2022 Zenika. All rights reserved

Lab 21

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

SCRUTER VOS SOURCES AVEC GO VET

Go fournit son propre linter, `go vet`, pour analyser les sources et trouver les erreurs courantes, comme par exemple (liste non-exhaustive) :

- Assignations inutiles
- build tags invalides
- Passer des lock en valeur, et autre mauvais usage du package `sync`.
- Code inaccessible (et donc mort)
- Mauvaise expression de formatage (dans les fonctions type `fmt.Printf`).

```
/tmp/vet.go:11: +build comment must appear before package clause and be followed by a
blank line
/tmp/vet.go:18: bar passes Lock by value: sync.Mutex
/tmp/vet.go:52: unreachable code
/tmp/vet.go:21: arg 1 for printf verb %s of wrong type: int
/tmp/vet.go:22: possible formatting directive in Sprint call
/tmp/vet.go:29: comparison of function foo != nil is always true
/tmp/vet.go:42: direct assignment to atomic value
/tmp/vet.go:45: suspect and: c == false && c == true
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

RESPECTER LE STYLE AVEC STATICCHECK

`go vet` vérifie que le code ne contient pas d'erreurs de programmation réelles.

`staticcheck` vérifie s'il y a des violations de styles.

- Se base sur un ensemble de conventions accepté par la communauté Open Source Go.
- N'émet pas d'erreurs, mais des suggestions :

```
lint.go:5:2: should not use dot imports (ST1001)
lint.go:12:5: error var SomeError should have name of the form ErrFoo (ST1012)
lint.go:17:7: receiver name should be a reflection of its identity; don't use generic
names such as "this" or "self" (ST1006)
```


© Copyright 2022 Zenika. All rights reserved

Lab 22

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

ET PLEIN D'AUTRES LINTERS ENCORE

Il existe beaucoup d'autres linter dans la communauté Go. gometalinter permet de les agréger (liste non-exhaustive) :

- `gotype` - analyse syntaxique et sémantique du code.
- `gocyclo` - calcule la complexité cyclomatique des fonctions.
- `varcheck` - trouve les variables et constantes globales non utilisées.
- `dupl` - recherche le code potentiellement dupliqué.

© Copyright 2022 Zenika. All rights reserved

ET PLEIN D'AUTRES LINTERS ENCORE

- `ineffassign` - détecte lorsque des affectations de variables ne sont pas utilisées.
- `interfacer` - propose des interfaces plus simples qui pourraient être utilisées.
- `goconst` - trouve des chaînes de caractères dupliquées pouvant être remplacées par une constante.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

GODOC (1/3)

La documentation est très importante pour rendre du code accessible et maintenable.

godoc va parser votre code source pour en extraire de la documentation.

```
godoc # Génération de la documentation dans la console
godoc -html # Génération de la documentation en html
godoc -http :8080 # Création d'un serveur http pour consulter la documentation
```

Pour documenter une méthode, le commentaire doit commencer par le nom de la méthode.

```
// Fprint formats using the default formats for its operands and writes to w.
// Spaces are added between operands when neither is a string.
// It returns the number of bytes written and any write error encountered.
func Fprint(w io.Writer, a ...interface{}) (n int, err error) { ... }
```

De manière générale, tout ce qui est exporté doit être documenté.

© Copyright 2022 Zenika. All rights reserved

GODOC (2/3)

Pour la documentation de package, on doit commencer par Package <le nom du package>

```
// Package sort provides primitives for sorting slices and user-defined
// collections.
package sort
```

Il est important d'avoir une première phrase concise et auto-suffisante, car c'est elle qui sera mise en avant lors de documentation de plus haut niveau (ex: liste des packages).

Si la documentation d'un package est trop importante, on peut la positionner dans un fichier `doc.go` qui ne contiendra qu'un commentaire et la mention du package.

© Copyright 2022 Zenika. All rights reserved

GODOC (3/3)

On peut aussi documenter les bugs connus avec des commentaires sous le format suivant :

```
// BUG(who): The rule Title uses for word boundaries does not handle Unicode punctuation
properly.
```

Et ce qui est deprecated avec le format:

```
CompressedSize     uint32 // Deprecated: Use CompressedSize64 instead.
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

UTILISER UN IDE

Plusieurs IDEs gèrent Go nativement ou grâce à des plugins :

- GoLand de JetBrains
- Visual Studio Code + Extension Go
- Atom + Package go-plus
- Vim + vim-go

Ces IDEs permettent notamment d'automatiser l'exécution des linters.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

MODULES : DÉFINITION

Module : Ensemble de packages versionnés ensemble

Création d'un module avec :

```
go mod init github.com/my/module
```

Sert de base aux packages, exemple :

```
github.com/my/module
|-- github.com/my/module/pkg1
|-- github.com/my/module/pkg2
...
```


© Copyright 2022 Zenika. All rights reserved

MODULES : ~~GOPATH~~

Plus besoin du GOPATH

Permet de travailler à n'importe quel endroit du FileSystem.

Seul GOBIN (par défaut "$GOPATH/bin") est encore utilisé :

```
go get github.com/spf13/cobra/cobra

tree ${GOBIN:-$(go env GOPATH)/bin}
/home/user/go/bin
├── cobra
├── dlv
...
```


© Copyright 2022 Zenika. All rights reserved

MODULES : FICHIER GO.MOD

Situé à la racine du module.

Contient des directives : `require`, `go`, `replace`, `exclude`

Exemple :

```
module github.com/Zenika/marcel

go 1.12

require (
    github.com/Pallinder/go-randomdata v1.2.0
    github.com/gorilla/mux v1.7.3
    github.com/gorilla/websocket v1.4.1
    gopkg.in/src-d/go-billy.v4 v4.3.2
    gopkg.in/src-d/go-git.v4 v4.13.1
)
```


© Copyright 2022 Zenika. All rights reserved

MODULES : MINIMAL VERSION SELECTION

Algorithme de résolution des dépendances.

```
github.com/my/module
|- require github.com/foo/bar v1.0.1
|- require github.com/some/dep v1.2.3
...

github.com/foo/bar
|- require github.com/some/dep v1.3.1
...
```

Version minimale satisfaisant tous les requires.

© Copyright 2022 Zenika. All rights reserved

MODULES : SEMANTIC IMPORT VERSIONING

Implicitement `import` désigne une v1 :

```
import "github.com/sirupsen/logrus"
```

📄 `go.mod` :

```
require github.com/sirupsen/logrus v1.4.2
```

Les versions majeures doivent être dans le chemin d'import :

```
import "gopkg.in/src-d/go-git.v4"
```

📄 `go.mod` :

```
require gopkg.in/src-d/go-git.v4 v4.13.1
```


© Copyright 2022 Zenika. All rights reserved

Lab 23

© Copyright 2022 Zenika. All rights reserved

PLAN

- Formatter vos sources avec `gofmt`
- Scruter vos sources avec `go vet`
- Respecter le style avec `staticcheck`
- Et plein d'autres linters encore
- Documenter vos sources avec `godoc`
- Utiliser un IDE
- Gestion des dépendances : Modules
- Gestion des dépendances : GOPATH, vendoring et dep

© Copyright 2022 Zenika. All rights reserved

GOPATH, WORKSPACE GO (1/2)

Par convention, on considère que tout code Go est écrit et importé dans un workspace.

Ce workspace est défini par la variable d'environnement `GOPATH`.

```
$ mkdir $HOME/myGoWorkspace
$ export GOPATH=$HOME/myGoWorkspace
```

Depuis la version 1.8, la valorisation du `GOPATH` n'est plus obligatoire, il possède la valeur par défaut `$HOME/go`, ou bien `%USERPROFILE%/go` sur Windows.

Pour modifier le `GOPATH` de manière globale, il faut l'ajouter à votre `.bash_profile` (ou similaire).

Il est possible de connaître la valeur de votre `GOPATH` en utilisant la commande `go env GOPATH`

© Copyright 2022 Zenika. All rights reserved

GOPATH, WORKSPACE GO (2/2)

Le contenu du GOPATH possède une architecture standard :

```
$ tree -L 1 $GOPATH
/home/gopher/go
├── bin # Binaires des commandes
├── pkg # Binaires des packages
└── src # Sources des packages
```

Afin de bénéficier des binaires installés par les commandes go dans un contexte global, plusieurs manières sont possibles :

- utiliser la variable d'environnement GOBIN pour spécifier un autre dossier (par défaut $GOPATH/bin)

```
export GOBIN=$HOME/bin
```

- ajouter $GOPATH/bin dans votre PATH

```
export PATH="$GOPATH/bin:$PATH"
```


© Copyright 2022 Zenika. All rights reserved

GESTION DES DÉPENDANCES : VENDORING (1/2)

Avant la généralisation des modules, la gestion des dépendances en Go était un sujet brulant et encore très débattu. Les contours n'étaient pas encore bien définis, et la communauté expérimentait beaucoup.

> “I want to have reproducible builds on different computers at different times. This is currently problematic; let’s make things better.”
>
> – David Hinkes, Go+

Il y a cependant une technique qui se détachait : le vendoring.

> “If you’re using an externally supplied package and worry that it might change in unexpected ways, the simplest solution is to copy it to your local repository. (This is the approach Google takes internally.) […]”
>
> – Go FAQ

© Copyright 2022 Zenika. All rights reserved

GESTION DES DÉPENDANCES : VENDORING (2/2)

Le compilateur Go va chercher les packages importés dans :

- le dossier `vendor` du projet
- le `GOPATH`
- le `GOROOT`

Conseil: Il n'est pas recommandé d'utiliser la technique de vendoring lorsque que l'on écrit des librairies ; il est très facile d'avoir des conflits de code (ou alors, réécrivez vos imports).

```
./foo.go:12: cannot use it (type "broken-vendor/vendor/a".A) as type "broken-
vendor/vendor/b/vendor/a".A in argument to b.Do
```


© Copyright 2022 Zenika. All rights reserved

DEP (1/3)

- https://github.com/golang/dep
- Projet pour gérer les dépendances
- "dep is safe for production use"
- Sera inclu à terme dans la go toolchain (en experimental en 1.10)
- Pas le seul projet pour gérer les dépendances (`godep`, `glide`, `govendor`, etc.)

© Copyright 2022 Zenika. All rights reserved

DEP (2/3)

- dep init va générer un fichier Gopkg.toml avec vos dépendances

```
[constraint]]
  branch = "parse-constraints-with-dash-in-pre"
  name = "github.com/Masterminds/semver"
  source = "https://github.com/carolynvs/semver.git"

[[constraint]]
  name = "github.com/Masterminds/vcs"
  version = "1.11.0"

[[constraint]]
  branch = "v2"
  name = "github.com/go-yaml/yaml"
```

- Un fichier Gopkg.lock va aussi être généré pour fixer les versions de dépendances

© Copyright 2022 Zenika. All rights reserved

DEP (3/3)

- dep init va aussi initier votre répertoire vendor
- dep ensure va vous permettre de mettre à jour votre répertoire vendor si vous retirez/ajoutez/mettez à jour une dépendance.

```
dep ensure # Check de la synchro entre votre Gopkg.toml et vendor
dep ensure -add github.com/foo/bar # Ajout de la lib bar
dep ensure -update # Mets à jour vos dépendances en respectant les contraintes
                   # du Gopkg.toml
```


© Copyright 2022 Zenika. All rights reserved

BIBLIOTHÈQUE STANDARD

© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json
- Package sync

© Copyright 2022 Zenika. All rights reserved

BIBLIOTHÈQUE STANDARD

L'un des atouts de Go est sa Bibliothèque Standard. Elle contient de nombreux packages utiles pour développer de vraies applications.

Ces packages se trouvent dans le sous-répertoire `pkg` du répertoire `$GOROOT` et sont distribués avec Go.

Nous avons déjà vu un certain nombre de ces packages. Dans ce chapitre nous verrons une liste non exhaustive des packages les plus représentatifs.

La liste complète des packages de la Standard Library se trouve ici : https://pkg.go.dev/std

© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json
- Package sync

© Copyright 2022 Zenika. All rights reserved

PACKAGE FMT (1/2)

Formatage et scannage de chaînes :

```
// 1100 sur la sortie standard
fmt.Println("%b", 12)
// Renvoie la string "this statement is false"
fmt.Sprintf("this statement is %t", false)
// Envoie "World is 世界 in japanese" dans writer
fmt.Fprintf(writer, "World is %s in japanese", "世界")

// Scanner une string
var a, b, c string
fmt.Sscan("this statement is false", &a, &b, &c)
fmt.Println(a)
fmt.Println(b)
fmt.Println(c)
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE FMT (2/2)

Implémenter l'interface `fmt.Stringer` :

```
type game struct {
    player1 string
    player2 string
    score   map[string]int
}

func (g *game) String() string {
    return fmt.Sprintf("The game is between %v and %v. The score is %d-%d.",
        g.player1, g.player2,
        g.score[g.player1], g.score[g.player2],
    )
}

game := &game{"John", "Bill", map[string]int{"John": 10, "Bill": 8}}

fmt.Println(game)
// The game is between John and Bill. The score is 10-8.
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json
- Package sync

© Copyright 2022 Zenika. All rights reserved

PACKAGE STRINGS (1/2)

Manipulations de chaînes :

```
// strings.Contains
fmt.Println(strings.Contains("seafood", "foo"))         // true

// strings.Trim
fmt.Println(strings.Trim("!!Achtung! Achtung! ", "! ")) // ["Achtung! Achtung"]

// strings.Join / strings.Split
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
fmt.Printf("%q\n", strings.Split("a,b,c", ","))         // ["a" "b" "c"]
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE STRINGS (2/2)

Manipulations de chaînes :

```
// strings.IndexFunc
f := func(c rune) bool {
  return unicode.Is(unicode.Han, c)
}
fmt.Println(strings.IndexFunc("Hello, 世界", f))        // 7

// strings.Title
fmt.Println(strings.Title("her royal highness"))}      // Her Royal Highness

// strings.Map
rot13 := func(r rune) rune {
        return 'a' + (r-'a'+13)%26
}
fmt.Println(strings.Map(rot13, "zenika"))              // mravxn
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json

© Copyright 2022 Zenika. All rights reserved

PACKAGE IO (1/3)

Interactions avec les entrées/sorties :

```
// io.Copy
r := strings.NewReader("some io.Reader stream to be read\n")
if _, err := io.Copy(os.Stdout, r); err != nil {
    log.Fatal(err)
}

// io.MultiReader
r1 := strings.NewReader("first ")
r2 := strings.NewReader("second ")
r3 := strings.NewReader("third reader\n")
r := io.MultiReader(r1, r2, r3)
if _, err := io.Copy(os.Stdout, r); err != nil {  // first second third reader
    log.Fatal(err)
}
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE IO (2/3)

Interactions avec les entrées/sorties :

```
// io.MultiWriter
r := strings.NewReader("some io.Reader stream to be read\n")
var buf1, buf2 bytes.Buffer
w := io.MultiWriter(&buf1, &buf2)
if _, err := io.Copy(w, r); err != nil {
  log.Fatal(err)
}
fmt.Print(buf1.String())          // some io.Reader stream to be read
fmt.Print(buf2.String())          // some io.Reader stream to be read
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE IO (3/3)

Le package `io/fs` a été ajouté en go1.16 afin d'abstraire les interactions avec le système de fichier.

Il offre une interface standardisée quel que soit le type de système de fichier :

- Disque local
- Periphérique réseau distant
- Système de fichier virtuel / en mémoire
- Ou autre...

© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json

© Copyright 2022 Zenika. All rights reserved

PACKAGE CONTAINER/LIST

Liste doublement chaînées :

```
// Create a new list and put some numbers in it.
l := list.New()
e4 := l.PushBack(4)
e1 := l.PushFront(1)
l.InsertBefore(3, e4)
l.InsertAfter(2, e1)

// Iterate through list and print its contents.
for e := l.Front(); e != nil; e = e.Next() {
    fmt.Println(e.Value)
}
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE CONTAINER/RING

Liste circulaire :

```go
// Create a new ring of size 5
r := ring.New(5)

// Get the length of the ring
n := r.Len()

// Initialize the ring with some integer values
for i := 0; i < n; i++ {
  r.Value = i
  r = r.Next()
}

// Iterate through the ring and print its contents
r.Do(func(p interface{}) {
  fmt.Println(p.(int))
})
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json

© Copyright 2022 Zenika. All rights reserved

PACKAGE SORT

Trier une slice ne contenant pas un type basique :

```go
type Person struct {
    Name string
    Age  int
}

// ByAge implements sort.Interface for []Person based on the Age field.
type ByAge []Person
func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

func main() {
    people := []Person{
        {"Bob", 31},
        {"John", 42},
        {"Jenny", 26},
    }

    sort.Sort(ByAge(people))
    fmt.Println(people) // [{Jenny 26} {Bob 31} {John 42}]
}
```


© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json

© Copyright 2022 Zenika. All rights reserved

PACKAGE NET/HTTP (1/2)

Le package net/http rend très simple l'implémentation d'un client HTTP. Les requêtes Get, Head, Post et PostForm sont supportées :

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := io.ReadAll(resp.Body)

// ...

resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)

// ...

resp, err := http.PostForm("http://example.com/form",
    url.Values{"key": {"Value"}, "id": {"123"}})
```


© Copyright 2022 Zenika. All rights reserved

PACKAGE NET/HTTP (2/2)

La fonction `http.ListenAndServe` démarre un serveur HTTP.

Quand on spécifie `nil` comme `Handler` de `ListenAndServe`, le handler par défaut (`DefaultServeMux`) est utilisé.

Avec les fonctions `http.Handle` et `http.HandleFunc` il est possible de rajouter des handlers à `DefaultServeMux`.

```
http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

Le package `http` supporte le protocole `HTTP/2` de façon transparente.

© Copyright 2022 Zenika. All rights reserved

PLAN

- Package fmt
- Package strings
- Package io
- Packages container/...
- Package sort
- Package net/http
- Package encoding/json

© Copyright 2022 Zenika. All rights reserved

PACKAGE ENCODING/JSON

Décoder du JSON :

```go
type Person struct {
    FirstName    string    `json:"first_name"`
    LastName     string    `json:"last_name"`
    MiddleName   string    `json:"middle_name,omitempty"`
}

func main() {
    json_string := `
    {
        "first_name": "John",
        "last_name": "Smith"
    }`

    person := new(Person)
    json.Unmarshal([]byte(json_string), person)
    fmt.Println(person)

    new_json, _ := json.Marshal(person)
    fmt.Printf("%s\n", new_json)
}
// *Output*
// &{John Smith }
// {"first_name":"John","last_name":"Smith"}
```


© Copyright 2022 Zenika. All rights reserved

Lab 24

© Copyright 2022 Zenika. All rights reserved

Lab 25

© Copyright 2022 Zenika. All rights reserved

COMPILATION

© Copyright 2022 Zenika. All rights reserved

PLAN

- Contraintes de build
- File embedding
- Passer des arguments au linker

© Copyright 2022 Zenika. All rights reserved

CONTRAINTES DE BUILD (1/2)

Pour simplifier le support de plusieurs systèmes d'exploitation et architecture, on peut dire à Go de ne compiler certains fichiers que si certaines contraintes sont remplies.

On se sert pour cela de commentaires `//go:build .....` en début de fichier.

Exemple pour compiler un fichier uniquement si la cible est linux en architecture i386 :

```
//go:build linux && 386
```


© Copyright 2022 Zenika. All rights reserved

CONTRAINTES DE BUILD (2/2)

Il est également possible de définir des conditions personnalisées, en passant par des tags :

```
//go:build !quick_tests
```

```
go test --tags=quick_tests
```


© Copyright 2022 Zenika. All rights reserved

FILE EMBEDDING

On peut embarquer un fichier dans le binaire exécutable créé par le compilateur. C'est utile pour gérer plus facilement des assets.

```
import "embed"

//go:embed file.json
var jsonFile string

//go:embed directory/
var gs embed.FS
```

On peut embarquer des fichiers en tant que `string` ou `[]byte`, mais également une arborescence de file system.

© Copyright 2022 Zenika. All rights reserved

LINKER

Si certaines valeurs ne sont connues qu'au moment de la compilation plutôt que durant le développement, on peut les passer au linker :

```go
package main

import "fmt"

var Version = "development"

func main() {
    fmt.Println("Version:\t", Version)
}
```

```
go build -ldflags="-X 'main.Version=v1.0.0'"
```

C'est pratique pour embarquer des numéros de versions, ou une date de compilation.

© Copyright 2022 Zenika. All rights reserved

POUR ALLER PLUS LOIN

© Copyright 2022 Zenika. All rights reserved

RÉFÉRENCES ET LIENS (1/2)

- Go Playground
- Documentation officielle
- Le go tool
- A Tour of Go
- How to Write Go Code
- Effective Go
- GoDoc - Tous les packages Go
- Go Wiki
- Awesome Go

© Copyright 2022 Zenika. All rights reserved

RÉFÉRENCES ET LIENS (2/2)

- Le livre Go Bootcamp
- Le livre The Go Programming Language
- Le livre Go Cookbook
- Le groupe discussion gonuts
- La video de Russ Cox A Tour of The Go Programming Language
- L'interview des CastCodeurs de Francesc Campoy

© Copyright 2022 Zenika. All rights reserved

VEILLE

- Golang Weekly
- r/golang

Sur Twitter :

- @golang
- go news

© Copyright 2022 Zenika. All rights reserved