Le langage Go

2ème partie

Rob Pike
r@google.com

Traduction en français, adaptation
xavier.mehaut @gmail.com

(Version de Juin 2011)

Sommaire de la journée

Exercices

des questions?

Types composite

structures, tableaux, slices, dictionnaires

Méthodes

plus seulement réservées aux structures

Interfaces

Google

Exercices

Des questions?

Google

Tableaux

Google

Tableaux (arrays)

Les tableaux en Go sont quelque peu différents de ceux en C ; se rapprochent plus des tableaux en Pascal. (les slices, le prochain sujet, se comportent un peu plus comme des tableaux C).

```
var ar [3]int
```

déclare `ar` comme étant un tableau de 3 entiers, initialisés à zéro.

La taille fait partie du type.

La fonction native `len` renvoie la taille :

```
len(ar) == 3
```

Google

Les tableaux sont des valeurs et non des pointeurs

Les tableaux sont des valeurs, pas des pointeurs implicites comme en C.
Vous pouvez néanmoins récupérer l'adresse d'un tableau, attribuer un pointeur à un tableau (par exemple pour le passer efficacement sans recopie à une fonction) :

```
func f(a [3]int) { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }
func main() {
   var ar [3] int
   f(ar)     // passe  une copie de ar
   fp(&ar)   // passe  un pointeur sur ar
}
```

Sortie écran (Print et consors reconnaissent les tableaux):

```
 [0 0 0]
&[0 0 0]
```

Google

Les littéraux de type tableau

Tous les types composite possèdent une syntaxe similaire pour créer des valeurs. Dans le cas des tableaux, cela ressemble à :

Tableau de 3 entiers:

```
[3]int{1, 2, 3}
```

Tableau de 10 entiers, les trois premières cases étant seuls initialisées à des valeurs autres que zéro:

```
[10]int{ 1, 2, 3}
```

Vous ne voulez pas borner votre tableau? Utilisez `...` Pour initialiser sa longueur:

```
[...]int{1, 2, 3}
```

Vous ne voulez pas tout initialiser? Utilisez les paires `key:value` :

```
[10]int{2:1, 3:1, 5:1, 7:1}
```

Google

Pointeurs sur des littéraux de type tableau

Vous pouvez obtenir l'adresse d'un littéral de type tableau afin de récupérer un pointeur sur une instance nouvellement créée :

```
func fp(a *[3]int) { fmt.Println(a) }
func main() {
   for i := 0; i < 3; i++ {
     fp(&[3]int{i, i*i, i*i*i})
   }
}
```

Sortie écran:

```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```

Google

Les slices

Google

Les slices : définition

Une slice (tranche) est une référence vers une section d'un tableau.

Les slices sont habituellement plus utilisées que des tableaux purs.
Une slice ne coûte pas grand chose en espace mémoire et en performances (plus d'information sur le sujet à venir).

Un type slice ressemble à un type tableau sans mention de la taille :

```
var a []int
```

La fonction native `len(a)` *retourne le nombre d'éléments de la slice.*

On crée une slice en découpant un tableau ou une autre slice :

```
a = ar[7:9]
```

Les positions valides de `a` *seront alors 0 et 1;* `len(a)==2`.

Google

Raccourcis pour les slices

Quand on crée une slice, le premier index est par défaut à 0 :

`ar[:n]` signifie *`ar[0:n]`.*

Le second index est par défaut la `len(tableau ou slice)`:

`ar[n:]` *signifie `ar[n:len(ar)]`.*

Enfin pour créer une slice d'un tableau :

`ar[:]` *signifie `ar[0:len(ar)]`.*

Google

Un slice référence un tableau

Conceptuellement:

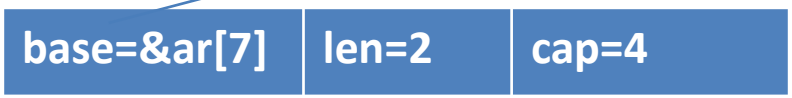
```
type Slice struct {
  base *elemType // pointeur sur le 0ème élément
  len int        // nb. d'éléments dans la slice
  cap int        // num. d'éléments disponibles
}
```

Tableau:

ar:

Slice:

a=ar[7:9]:



Google

Construire une slice

Les littéraux de type slice ressemblent à des littéraux de type tableau sans la taille :

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

Ce qui se passe, c'est la création d'un tableau de longueur 5 suivie de la création d'une slice qui y fait référence.

Nous pouvons également allouer une slice (et le tableau sous-jacent) avec la fonction native `make` *:*

```
var s100 = make([]int, 100) // slice: 100 ints
```

Pourquoi `make` *et pas* `new`*? Parce nous avons besoin de fabriquer une slice et pas seulement d'allouer de la mémoire. Notez que* `make([]int, 10)` *retourne* `[]int` *tandis que* `new([]int)` *retourne* `*[]int`*.*

Utilisez `make` *pour créer des slices, des maps, et des channels (plus tard dans le cours)*

Google

La capacité d'une slice

Une slice fait référence à un tableau sous-jacent, ainsi il peut y avoir des éléments à l'exprémité de la slice qui sont présents dans le tableau.

La fonction native `cap` *(capacity) indique jusqu'à quelle taille la slice pourrait encore grossir s'il le fallait.*

Après

```
    var ar = [10]int{0,1,2,3,4,5,6,7,8,9}
    var a = ar[5:7] // référence vers le sous-tableau{5,6}
```

`len(a)` *est 2 et* `cap(a)` *est 5. On peut "reslicer":*

```
    a = a[0:4] // référence vers le sous-tableau {5,6,7,8}
```

`len(a)` *est désormais 4 mais* `cap(a)` *est toujours 5.*

Google

Retailler une slice

Les slices peuvent être utilisées pour faire grossir un tableau.

```
var sl = make([]int, 0, 100) // len 0, cap 100

func appendToSlice(i int, sl []int) []int {
   if len(sl) == cap(sl) { error(...) }
   n := len(sl)
   sl = sl[0:n+1] // étend d'une longueur de 1
   sl[n] = i
   return sl
 }
```

Ainsi la longueur de `sl` est toujours le nombre d'éléments mais il grossit avec le besoin. Ce style de programmation, propre à Go, est très économe.

Google

Les slices sont légères

Sentez vous libres d'allouer et de retailler des slices comme vous l'entendez. Elles sont légères; pas de besoin d'allocation. Souvenez-vous que ce sont des références, ainsi le stockage sous-jacent peut être modifié.

Par exemple, les I/O utilisent des slices, pas des compteurs:

```
func Read(fd int, b []byte) int
var buffer [100]byte
for i := 0; i < 100; i++ {
   // remplit le buffer d'un octet à la fois
   Read(fd, buffer[i:i+1]) // pas d'allocation ici
}
```

Scinder un buffer:

```
header, data := buf[:n], buf[n:]
```

Les chaînes de caractères peuvent être également slicées avec la même efficacité.

Google

Les dictionnaires (maps)

Google

Les dictionnaires (Map)

Les dictionnaires sont un autre type de référence. Ils sont déclarés de la façon suivante :

```
var m map[string]float64
```

Ceci déclare un dictionnaire indexé avec une clef de type `string` *et une valeur de type* `float64`*.*

C'est analogue à ce que l'on peut trouver en C++

```
type *map<string,float64> (note the *).
```

Etant donné un dictionnaire `m`*,* `len(m)` *retourne le nombre de clefs.*

Google

Création d'un dictionnaire

Comme avec une slice, une variable de type dictionnaire ne se réfère à rien; vous devez y mettre quelque chose dedans avant de l'utiliser.

Trois façons de le faire :

1) Littéral: liste de `clef:valeur`

```
m = map[string]float64{"1":1, "pi":3.1415}
```

2) Création

```
m = make(map[string]float64) // make pas new
```

3) Affectation

```
var m1 map[string]float64
m1 = m  //m1 et m se réfèrent maintenant à la même map
```

Google

Indexation d'un dictionnaire

(Les prochains exemples utilisent tous

```
m = map[string]float64{"1":1, "pi":3.1415}
```

)

Accède à un élément comme à une valeur; si pas présente, renvoie une valeur zéro :

```
un := m["1"]
zéro := m["pas présent"] // mets zéro à 0.0.
```

Stocke un élément :

```
m["2"] = 2
m["2"] = 3
```

Google

Test de l'existence

Pour tester si une clef est pésente dans un dictionnaire, nous pouvons utiliser une affectation multiple, la forme "comma ok" :

```
m = map[string]float64{"1":1, "pi":3.1415}
var value float64
var present bool
value, present = m[x]
```

or idiomatiquement

```
value, ok := m[x] // la forme "comma ok (virgule ok)"
```

Si `x` est présent dans le dictionnaire, positionne le booléen à `true` et la valeur récupère la valeur associé à la clef donnée. Si non, positionne le booléen à `false` et renvoie la valeur zero pour le type considéré.

Google

Suppression

La suppression d'une entrée se fait de la manière suivante :

```
m = map[string]float64{"1":1.0, "pi":3.1415}
var keep bool
var value float64
var x string = f()
m[x] = v, keep
```

Si `keep` est true, l'affectation de `v` dans le dictionnaire s'effectue bien; si `keep` est false, l'entrée du dictionnaire pour la clef `x` est supprimée.

```
m[x] = 0, false    // supprime l'entrée pour x
```

Google

For et range

La boucle `for` possède une syntaxe spéciale dans le cas d'une itération sur un tableau, d'une slice ou d'un dictionnaire:

```
m := map[string]float64{"1":1.0, "pi":3.1415}
for key, value := range m {
  fmt.Printf("key %s, value %g\n", key, value)
}
```

S'il n'y a qu'une seule variable, on ne récupère que la clef :

```
for key = range m {
  fmt.Printf("key %s\n", key)
}
```

Les variables peuvent être affectées ou déclarées en utilisant l'opérateur `:=`.
Pour les tableaux et les slices, le `range` renvoie l'index et la valeur.

Google

Range sur une chaîne de caractères

Un `for` utilisant un `range` sur une chaîne boucle sur les élements Unicode et non sur les octets. (Utiliser `[]byte` pour les octets, ou utiliser le `for` standard). La chaîne est considérée comme devant contenir des caractères en UTF-8.

La boucle

```
s := "[\u00ff\u754c]"
for i, c := range s {
  fmt.Printf("%d:%q ", i, c) // %q pour'quoted'
}
```

affiche `0:'[' 1:'ÿ' 3:'□ ' 6:']'`

Si un UTF-8 est erroné, le caractère est renvoyé est U+FFFD et l'index avance d'un seul octet.

Google

Les structures

Google

Les structures (struct)

Les structures vont vous sembler familières ; ce sont de simples déclarations de champs .

```
var p struct {
  x, y float64
}
```

De manière plus commune:

```
type Point struct {
  x, y float64
}
var p Point
```

Les `struct` permettent au programmeur de définir les bornes mémoire.

Google

Les structs sont des valeurs

Les structs sont des valeurs et `new(aStructType)` retourne un pointeur sur une valeur zéro (mémoire initialisée à zéro).

```
type Point struct {
  x, y float64
}
var p Point
p.x = 7
p.y = 23.4
var pp *Point = new(Point)
*pp = p
pp.x = Pi // sucre syntaxique pour(*pp).x
```

Il n'existe pas de notation -> pour les pointeurs sur structure.
Go vous fournit l'indirection pour vous.

Google

Construction des structs

Les structures sont des valeurs ainsi vous pouvez en créer initialisées à zéro (zéro ne signifiiant pas forcément 0 maisla valeur par défaut d'un type donné) juste en la déclarant.

```
var p Point // valeur initialisée à zéro, ici x et y =0
```

Vous pouvez aussi en allouer une avec `new`.

```
pp := new(Point) // allocation idiomatique
```

Les littéraux struct ont la syntaxe attendue.

```
p = Point{7.2, 8.4}
p = Point{y:8.4, x:7.2}
pp = &Point{7.2, 8.4} // idiomatique
pp = &Point{} // également idiomatique; == new(Point)
```

Comme avec les tableaux, prendre l'adresse d'un littéral struct donne l'adresse d'une valeur nouvellement créée.

Ces exemples sont des constructeurs.

Google

Exportation des types et champs

Les champs (et méthodes - à venir) d'une structure doivent commencer avec une majuscule pour être visibles à l'extérieur du package.

Type et champs privés :

```
type point struct { x, y float64 }
```

Typeet champs exportés :

```
type Point struct { X, Y float64 }
```

Type exporté avec un mix des champs:

```
type Point struct {
    X, Y float64   // exporté
    name string    // pas exporté
}
```

Vous pouvez même avoir un type privé avec des champs exportés. Exercice : quand est-ce utile?

Google

Champs anonymes

A l'intérieur d'une structure, vous pouvez déclarer des champs, comme une autre structure, sans leur donner un nom de champ.

*C'est ce que l'on appelle des champs **anonymes** et ils se comportent comme si les structures internes étaient simplement insérées ou "embarquées" vu de l'extérieur.*

Ce mécanisme simple fournit une façon de dériver quelques unes ou la totalité de votre implémentation d'un autre type ou de types.

Un exemple à suivre pour illustrer le propos.

Google

Un champ struct anonyme

```
type A struct {
   ax, ay int
}
type B struct {
   A
   bx, by float64
}
```

`B` agit comme si il possédait 4 champs, *`ax`, `ay`, `bx`, et `by`. C'est presque comme si `B` était `{ax, ay int; bx, by float64}`.*

Néanmoins, les littéraux comme B doivent être remplis dans le détail:

```
b := B{A{1, 2}, 3.0, 4.0}
fmt.Println(b.ax, b.ay, b.bx, b.by)
```

Affiche 1 2 3 4

Google

Les champs anonymes ont le type comme nom

Mais c'est plus riche qu'une simple interpolation de champs :
`B` *aussi a un champ* `A`*.*

Le champ anonyme ressemble à un champ dont le nom est son type.

```
b := B{A{ 1, 2}, 3.0, 4.0}
fmt.Println(b.A)
```

affiche `{1 2}`*.*

Si `A` *venait d'un autre package, le champ s'appelerait également A:*

```
import "pkg"
type C struct { pkg.A }
...
c := C {pkg.A{1, 2}}
fmt.Println(c.A) // pas c.pkg.A
```

Google

Champs anonymes de n'importe quel type

N'importe quel type nommé, ou pointeur sur celui-ci, peut être utilisé dans un type anonyme et peut apparaître à n'importe quel endroit de la structure.

```
type C struct {
  x float64
  int
  string
}
c := C{3.5, 7, "bonjour"}
fmt.Println(c.x, c.int, c.string)
```

affiche `3.5 7 bonjour`

Google

Conflits et masquage

S'il existe deux champs avec les même noms (éventuellement un nom de type dérivé), les règles suivantes s'appliquent :

1) *Un champ externe masque un champs interne. Ceci fournit une façon de surcharger un champs ou une méthode.*

2) *Si le même nom apparait deux fois à un même niveau, c'est une erreur si le nom est utilisé par le programme. (si pas utilisé, pas de problème).*

Pas de règles pour résoudre les ambiguïtés; doit être corrigé par le programmeur.

Google

Exemples de conflits

```
type A struct { a int }
type B struct { a, b int }
type C struct { A; B }
var c C
```

Utiliser `c.a` est une erreur: est-ce `c.A.a` ou `c.B.a`?

```
type D struct { B; b float64 }
var d D
```

Utiliser `d.b` est OK: il s'agit de `float64`, et non de `d.B.b`

On peut accéder à `b` via `D.B.b`.

Google

Méthodes

Google

Méthodes sur les structures

Go *ne possède pas de classes, mais vous pouvez attacher des méthodes à n'importe quel type. Les méthodes sont déclarées de manière séparée des types, comme des fonctions avec un récepteur explicite.*

```
type Point struct { x, y float64 }
// une méthode sur *Point
func (p *Point) Abs() float64 {
  return math.Sqrt(p.x*p.x + p.y*p.y)
}
```

NB:

récepteur explicite (pas de `this` *automatique), dans le cas du type* `*Point` *utilisé dans la méthode.*

Google

Méthodes sur des valeurs struct

Une méthode ne requière pas un pointeur comme récepteur.

```
type Point3 struct { x, y, z float64 }
// une méthode sur Point3
func (p Point3) Abs() float64 {
  return math.Sqrt(p.x*p.x + p.y*p.y +
  p.z*p.z)
}
```

Ceci est une peu cher payé, parce que Point3 sera toujours passé par valeur à la méthode et non par pointeur, mais c'est valide en Go.

Invocation d'une méthode

Juste comme vous y attendez.

```
p := &Point{ 3, 4 }
fmt.Print(p.Abs()) // affichera 5
```

Un exemple sans structure:

```
type IntVector []int
func (v IntVector) Sum() (s int) {
  for _, x := range v { //  identificateur blank!
    s += x
  }
  return
}

fmt.Println(IntVector{1, 2, 3}.Sum()
```

Google

Les règles de base des méthodes

Les méthodes sont attachées à un type nommé, disons `Foo` *et y sont liées statiquement.*

Le type d'un récepteur dans une méthode peut être soit `*Foo` *ou* `Foo`*. Vous pouvez avoir des méthodes* `Foo` *et des méthodes* `*Foo`*.* `Foo` *lui-même ne peut pas être un type pointeur bien que les méthodes puissent avoir un récepteur de type* `*Foo`*.*

Le type `Foo` *doit être défini dans le même package que celui de ses méthodes.*

Google

Pointeurs et valeurs

Go déréférence automatiquement les valeurs pour vous à l'invocation des méthodes.

Par exemple, même si une méthode possède un type récepteur `*Point`*, vous pouvez l'invoquer sur une valeur de type* `Point`*.*

```
p1 := Point{ 3, 4 }
fmt.Print(p1.Abs())     // sucre syntaxique pour
                        //(&p1).Abs()
```

De manière similaire, si des méthodes sont sur `Point3` *vous pouvez utiliser une valeur de type* `*Point3`*:*

```
p3 := &Point3{ 3, 4, 5 }
fmt.Print(p3.Abs())     // sucre syntaxique pour
                        //(*p3).Abs()
```

Google

Méthodes sur des champs anonymes

Naturellement, quand un champ anonyme est embarqué dans une structure, les méthodes de ce type sont embarquées de la même façon – en effet, il hérite des méthodes.

Ce mécanisme offre une façon simple d'émuler certains effets du sous-classage et de l'héritage.

Google

Exemple de champ anonyme

```
type Point struct { x, y float64 }
func (p *Point) Abs() float64 { ... }

type NamedPoint struct {
  Point
  name string
}

n := &NamedPoint{Point{3, 4}, "Pythagoras"}
fmt.Println(n.Abs()) // prints 5
```

Google

Surcharger une méthode

La surcharge fonctionne comme avec les champs.

```
type NamedPoint struct {
  Point
  name string
}
func (n *NamedPoint) Abs() float64 {
  return n.Point.Abs() * 100.
}

n := &NamedPoint{Point{3, 4}, "Pythagoras"}
fmt.Println(n.Abs()) // prints 500
```

Bien entendu vous pouvez avoir des champs multiples anonymes avec des types variés – une simple version de l'héritage multiple. Les règles de résolution des conflits rendent les choses simples néanmoins.

Google

Un autre exemple

Une utilisation encore plus irrésistible des champs anonymes.

```
type Mutex struct { ... }
func (m *Mutex) Lock() { ... }

type Buffer struct {
  data [100]byte
  Mutex // ne nécessite pas d'être le premier dans le buffer
}
var buf = new(Buffer)
buf.Lock() // == buf.Mutex.Lock()
```

Notez que le récepteur de `Lock` (l'adresse de) est le champ `Mutex`, pas la structure environnante (contraste avec le sous-classage et les mix-in Lisp)

Google

D’autres types

Les méthodes ne sont pas spécifiques aux structures. Elles peuvent être définies pour n’importe quel type (non pointeur).

Le type doit être défini néanmoins dans votre package. Vous ne pouvez donc pas écrire une méthode pour le type `int` mais vous pouvez déclarer un nouveau type `int` et lui attribuer des méthodes.

```
type Jourint

var nomsjour= []string {
   "Lundi", "Mardi", "Mercredi", ...
}

func (jour Jour) String() string {
   return nomsjour[jour]
}
```

Google

D’autres types (suite)

Maintenant voici un type semblable à une énumération qui sait comment s’afficher lui-même :

```
const (
  Lundi Jour= iota
  Mardi
  Mercredi
  // ...
)
var jour= Mardi
fmt.Printf("%q", jour.String())
// affiche “Mardi"
```

Google

Print comprend les méthodes String

Au moyen de techniques divulguées sous peu, `fmt.print` et ses amies peuvent identifier les valeurs qui implémentent la methode `String` comme définie pour la type `Jour`. De telles valeurs sont automatiquement formatées en invoquant la méthode.

Ainsi:

```
fmt.Println(0, Monday, 1, Tuesday)
```

Affiche `0 Lundi 1 Mardi`.

Définissez une méthodes `String` pour vos types et ils s'afficheront de manière sympathique sans plus de travail.

Google

Visibilité des champs et des méthodes

Révision

Go est très différent du C++ dans le domaine de la visibilité.

Les règles de Go sont les suivantes:

1. *Go possède une visibilité* `package` *(C++ a une visibilité fichier).*
2. *L'orthographe détermine ce qui est exporté ou local (pub/priv).*
3. *Les structs dans un même package ont un plein accès aux champs et méthodes des autres types.*
4. *Un type local peut exporter ses champs et méthodes*
5. *Pas de vrai sous-classage, ni notion de "protégé"*
6. *Ces simples règles fonctionnent très bien en pratique, pas de sur-specification déconnectée du réel*

Google

Interfaces

Google

Regardez avec attention

Nous allons jeter un coup d'oeil sur l'aspect le plus inhabituel de Go:

Les interfaces

Laissez vos préjugés svp sous le paillasson!

Google

Introduction

Jusqu'à présent, tous les types que nous avons examinés étaient concrets : ils implémentaient quelque chose.

*Il y a un type supplémentaire à considérer encore : **l'interface**. C'est complétement abstrait. Il n'implémente RIEN. Il spécifie à la place un ensemble de propriétés qu'une implémentation doit fournir, comme en java en quelque sorte.*

Mais par rapport à java, le concept de « la valeur de type interface » en Go est tout à fait nouveau!

Google

Définition d’une interface

Le mot "interface" est un peu surchargé en Go : il y a le concept d’une interface, il y a un type interface, et ensuite il y a des valeurs de ce type.

Premièrement, le concept.

Définition:

Une interface est un ensemble de méthodes.

Pour définir à gros grain, les méthodes implémentées par un type concret comme une structure forment l’interface de ce type.

Google

Exemple

Nous avons vu ce type simple auparavant :

```
type Point struct { x, y float64 }
func (p *Point) Abs() float64 { ... }
```

L'interface du type `Point` *possède la méthode :*

```
Abs() float64
```

Ce n'est pas

```
func (p *Point) Abs() float64
```

parce que l'interface fait abstraction du récepteur.

Nous embarquons `Point` *dans un nouveau type* `NamedPoint`*;*
`NamedPoint` *possède la même interface.*

Google

Le type interface

Un type `interface` est une spécification d'une interface, un ensemble de méthodes implémentées par d'autres types.

Ci-dessous une interface simple avec une seule méthode :

```
type AbsInterface interface {
  Abs() float64 // le recepteur est implicite
}
```

Ceci est la définition de l'interface implémentée par `Point`, où, dans notre terminologie, `Point` *implémente* `AbsInterface` *également,* `NamedPoint` *et* `Point3` *implementent* `AbsInterface`

Les méthodes sont décrites à l'intérieur de la déclaration de l'interface.

Google

Un exemple

```
type MonFloat float64

func (f MonFloat) Abs() float64 {
    if f < 0 { return float64(-f) }
    return f
}
```

`MonFloat` ***implemente*** *`AbsInterface`* ***même si*** `float64` ***ne le fait pas.***

(NB: *`MonFloat`* ***n'est pas le la conversion (boxing) de*** *`float64`****; sa représentation est identique à*** *`float64`****.)***

Google

De plusieurs vers plusieurs

Une interface peut être implémentée par un nombre arbitraire de types. `AbsInterface` *est implémenté par n'importe quel qui possède une méthode avec une signature* `Abs() float64`*, nonobstant le fait que le type puisse avoir d'autres méthodes.*

Un type peut implémenter un nombre arbitraire d'interfaces. `Point` *en implémente ainsi au moins deux :*

```
type AbsInterface interface { Abs() float64 }
type EmptyInterface interface { }
```

Tout type implémente une interface vide `EmptyInterface`*. Ce qui va nous être pratique dans le futur…*

Google

Valeur de type interface

Une fois qu'une variable est déclarée avec un type interface, elle peut enregistrer n'importe quelle valeur qui implément cette interface.

```
var ai AbsInterface

pp := new(Point)
ai = pp                         // OK: *Point possède  Abs()
ai = 7.                         // erreur à la compilation:
                                // float64 ne possède pas Abs()
ai = MonFlottant(-7.)           // OK: MonFlottant possède Abs()

ai = &Point{ 3, 4 }
fmt.Printf(ai.Abs())            // appel de méthode
```

affiche 5.

NB: `ai` n'est pas un pointeur! C'est une valeur interface.

Google

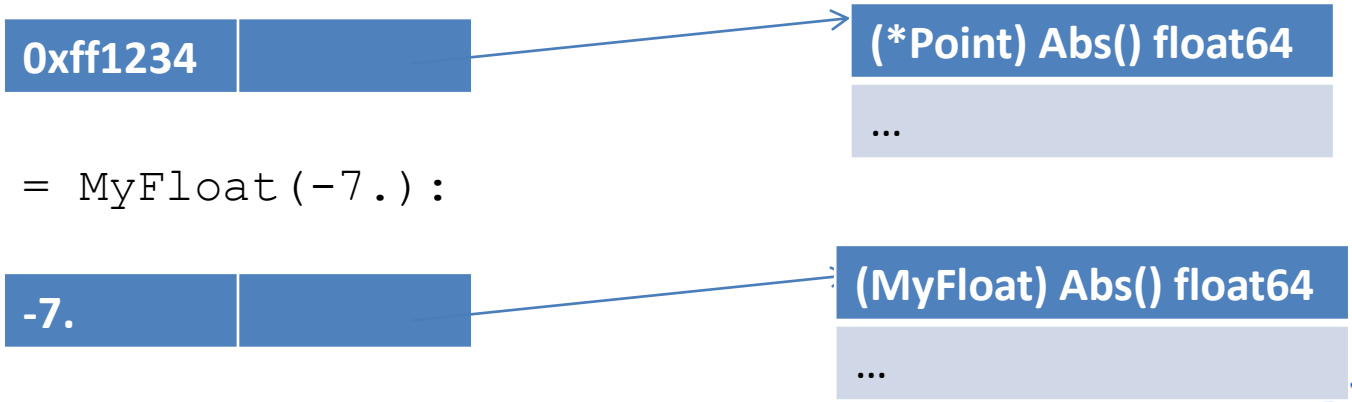
En mémoire

`ai` *n'est pas un pointeur! C'est une structure de données multi-mots.*

`ai`

A différents instant, `ai` *possède un type et une valeur différente*

```
ai = &Point{3,4} (un *Point à l'adresse 0xff1234):
```



Google

Trois choses importantes

1. *Les interfaces définissent un ensemble de méthodes. Elles sont pures et abstraites : pas d'implémentation, pas de champs de données. Go possède une claire séparation entre interface et implementation.*

2. *Les valeurs de type interface sont juste des valeurs. Elles contiennent n'importe quelles valeurs concrètes qui implémentent toutes les méthodes définies dans l'interface. Cette valeur concrète peut ou non être un pointeur.*

3. *Les types implémentent les interfaces en possédant juste ses méthodes. Ils ne doivent pas déclarer ce qu'elles font? Par exemple, tout type implémente une interface vide* `interface{}`.

Google

Exemple : io.Writer

Ci-après la signature concrète de `fmt.Fprintf`:

```
func Fprintf(w io.Writer, f string, a ... interface{})
(n int, error os.Error)
```

Elle n'écrit pas dans un fichier, mais écrit vers quelque chose du type `io.Writer`*, qui est défini dans le package `io`:*

```
type Writer interface {
  Write(p []byte) (n int, err os.Error)
}
```

`Fprintf` peut par conséquent être utilisé pour écrire sur n'importe quel type qui possède une méthode Write *`Write`*, ce qui inclue les fichiers, les pipes, les connexions réseau, etc…

Google

I/O bufferisées

... Un buffer(tampon) en écriture vient du package `bufio`*:*

```
type Writer struct { ... }
```

`bufio.Writer` *implémente la méthode canonique* `Write`*.*

```
func (b *Writer) Write(p []byte) (n int, err
os.Error)
```

C'est aussi une "factory": donnez lui un `io.Writer`*, et il retournera un* `io.Writer` *bufferisé de la forme d'un* `bufio.Writer`*:*

```
func NewWriter(wr io.Writer) (b *Writer, err
os.Error)
```

Et bien entendu, `os.File` *implémente* `Writer` *également.*

Google

Mettons tout ça ensemble

```
import (
   "bufio"; "fmt"; "os"
)
func main() {
    // non bufferisé
    fmt.Fprintf(os.Stdout, "%s, ", "bonjour")
    // bufferisé: os.Stdout implémente io.Writer
    buf := bufio.NewWriter(os.Stdout)

    fmt.Fprintf(buf, "%s\n", "monde!")
    buf.Flush()
}
```

La bufferisation fonctionne avec n'importe quoi qui `Writes`.

Ressemble presque au pipes Unix, n'est-ce pas?

La composabilité est très puissante; voir le package `crypto`.

Google

Autres interface publiques dans io

Le package `io` possède :

```
Reader
Writer
ReadWriter
ReadWriteCloser
```

Ce sont des interfaces stylées mais évidentes : elles capturent les fonctionnalités de n'importe quoi implémentant les fonctions listées dans leurs noms.

C'est pourquoi nous pouvons avoir un package d'io bufferisées avec une implémentation séparée des io elle-mêmes : si les deux acceptent et fournissent des valeurs interfaces.

Google

Comparaison

En termes C++, un type `interface` *est comme une classe abstraite pure spécifiant les méthodes mais n'implémentant aucune d'elles.*

En termes java, une interface ressemble en effet plus au concept d'interface Java.

Mais, en Go il y a une différence majeure : un type n'a pas besoin de déclarer l'interface qu'il implémente, ni besoin d'en hériter. Si le type possède les méthodes, c'est qu'il implémente l'interface.

D'autres différences apparaîtront plus clairement plus tard.

Les champs anonymes fonctionnent également

```
type LockedBufferedWriter struct {
    Mutex // has Lock and Unlock methods
    bufio.Writer // has Write method
}

func (l *LockedBufferedWriter) Write(p []byte)
    (nn int, err os.Error) {
    l.Lock()
    defer l.Unlock()
    return l.Writer.Write(p) // inner Write()
}
```

`LockedBufferedWriter` ***implémente*** *`io.Writer`* ***, mais aussi à travers le*** *`Mutex`* ***anonyme***

```
type Locker interface { Lock(); Unlock() }
```

Google

Exemple : service HTTP

```
type Handler interface {
   ServeHTTP(ResponseWriter, *Request)
}
```

Ceci est l'interface définie par le package serveur HTTP. Pour déclarer un serveur HTTP, il faut définir un type qui implémente cette interface et se connecter au serveur (détails omis).

```
type Counter struct {
   n int // on pourrait juste dire type Counter int
}
func (ctr *Counter) ServeHTTP(w http.ResponseWriter,
   req *http.Request) {
   fmt.Fprintf(w, "counter = %d\n", ctr.n)
   ctr.n++
}
```

Google

Une fonction (type) qui est implémente un serveur HTTP

```
func notFound(w http.ResponseWriter, req *http.Request){
   w.SetHeader("Content-Type", "text/plain;" +
   "charset=utf-8")
   w.WriteHeader(StatusNotFound)
   w.WriteString("404 page not found\n")
}
```

Maintenant définissons un type qui implémente `ServeHTTP`*:*

```
type HandlerFunc func(http.ResponseWriter, *http.Request)
   func (f HandlerFunc) ServeHTTP(w http.ResponseWriter,
   req *http.Request) {
   f(w, req) // the receiver's a func; call it
}


var Handle404 = HandlerFunc(notFound)
```

Google

Containers et interface vide

Aperçu de la mise en œuvre des vecteurs *(en pratique, on tend à utiliser des slices brutes à la place, mais c'est informatif):*

```
type Element interface {}

// Vector est le container lui-même.
type Vector []Element

// At() retourne le i'ème élément.
func (p *Vector) At(i int) Element {
  return p[i]
}
```

Les Vectors peuvent contenir n'importe quoi parce tout type implémente l'interface vide par défaut

Google

Assertions

Une fois que vous avez mis quelque chose dans `Vector`*, il est stocké comme une valeur de type interface. On a besoin de la convertir pour récupérer l'élément original : on utilise pour cela une* `assertion`*. Syntaxe :*

```
interfaceValue.(typeToExtract)
```

Echouera si le type est mauvais – mais nous verrons cela dans le prochain slide.

```
var v vector.Vector
v.Set(0, 1234.)                 // stocké comme une valeur interface
i := v.At(0)                    // récupéré comme interface{}
if i != 1234. {}                // erreur à la compilation
if i.(float64) != 1234. {}      // OK
if i.(int) != 1234 {}           // erreur à l'exécution
if i.(MyFloat) != 1234. {}      // err: pas MyFloat
```

Les assertions ne sont levées qu'à l'exécution . Le compilateur rejette les assertions qui sont sûres d'échouer.

Google

Conversion d'interface vers une autre interface

Jusqu'à présent, nous avons déplacé des valeurs concrètes de ou vers des valeurs interface, mais les valeurs interface qui contiennent les méthodes appropriées peuvent aussi être converties.

En effet c'est le même schéma que de convertir (boxing) une valeur de type `interface` *pour extraire la valeur concrète sous-jacente, puis de la reconvertir pour avoir un nouveau type* `interface`*.*

Le succès de la conversion dépend de la valeur sous-jacente, pas du type d'`interface` *originel.*

Google

Exemple de conversion interface vers interface

soit:

```
var ai AbsInterface
type SqrInterface interface { Sqr() float64 }
var si SqrInterface
pp := new(Point) // say *Point has Abs, Sqr
var empty interface{}
```

Elles sont toutes OK :

```
empty = pp           // tout satisfait vide
ai = empty.(AbsInterface) // valeur sous-jacente
                     // implémente Abs()
                     // (échec d'excution sinon)
si = ai.(SqrInterface) // *Point possède Sqr()
                     // même si AbsInterface non
empty = si           // *Point implémente un ensemble vide
                     // Note: statiquement vérifiable
                     // ainsi l'assertion n'est pas nécessaire
```

Google

Tester avec des assertions

On peut utiliser les assertions "comma ok" pour tester une valeur de type.

```
elem := vector.At(0)
if i, ok := elem.(int); ok {
  fmt.Printf("int: %d\n", i)
} else if f, ok := elem.(float64); ok {
  fmt.Printf("float64: %g\n", f)
} else {
  fmt.Print("type inconnu\n")
}
```

Google

Tester avec un type switch

Syntace spéciale:

```
switch v := elem.(type) { // litéral
  keyword "type"
  case int:
   fmt.Printf("est int: %d\n", v)
  case float64:
   fmt.Printf(" est float64: %g\n", v)
  default:
   fmt.Print(" type inconnu\n")
}
```

Google

Est-ce que v implémente m()?

Pour aller un peu plus loin, testons si une valeur implémente une méthode.

```
type Stringer interface { String() string }
if sv, ok := v.(Stringer); ok {
  fmt.Printf("implémente String(): %s\n",
  sv.String()) // note: sv pas v
}
```

C'est ainsi que `Print`, etc., vérifie si un type peut être affiché ou pas.

Google

Réflexion et …

Il existe un package (`reflect`) qui vous permet d'examiner des valeurs pour découvrir leur type. Trop riche pour être décrite ici mais `Printf`, etc., l'utilise pour analyser ses arguments.

```
func Printf(format string, args ...interface{})
(n int, err os.Error)
```

A l'intérieur de `Printf`, les `args` deviennent une slice du type spécifié, i.e. `[]interface{}`, et `Printf` utilise le package réflexion pour dépaqueter chaque élément pour analyser son type.

Google

Réflexion et Print

Commé résultat, `Printf` et consors connaissent les types réels de leurs arguments. Parce qu'ils savent si les arguments sont non signés ou longs, il n'y a pas de `%u` ou `%ld`, seulement `%d`.

C'est ainsi que `Print` et `Println` peuvent afficher les arguments proprement sans une instruction de formatage .

Il y a aussi un format `%v` ("value") qui permet une sortie écran agréable par défaut pour n'importe quelles valeurs de n'importe quel type pour `Printf`.

```
fmt.Printf("%v %v %v %v", -1, "bonjour",
[]int{1,2,3}, uint64(456))
```

affiche `-1 bonjour [1 2 3] 456`.

En fait, `%v` est identique au formatage effectuée par `Print et Println`.

Google

Fonctions « variadiques »

Google

Fonctions « variadiques »: …

Les listes de paramètres à longueur variable sont déclarées avec la syntaxe suivante `...T`*, où* `T` *est le type des arguments individuels. De tels arguments doivent être le dernier de la liste d'arguments. A l'intérieur d'une fonction, l'argument variadique possède implictement le type* `[]T`*.*

```
func Min(args ...int) int {
  min := int(^uint(0)>>1) // int le plus large possible
  for _, x := range args { // args possède le type[]int
   if min > x { min = x }
  }
  return min
}
fmt.Println(Min(1,2,3), Min(-27), Min(), Min(7,8,2))
```

affiche `1 -27 2147483647 2`

Google

Slices dans les variadiques

L'argument devient une slice. Qu'arrive t il si vous voulez passer la slice comme argument directement? Utilisez `...` *Lors de l'appel (fonctionne seulement avec les variadiques.)*

Se rappeler que : `func Min(args ...int) int`

Les deux invocations retournent `-2`:

```
Min(1, -2, 3)
slice := []int{1, -2, 3}
Min(slice...) // ... Transforme une slice en arguments
```

Ceci est une erreur de type :

```
Min(slice)
```

Parce que la slice ets de type `[]int` *tandis que les arguments* `Min` *doivent être individuellement* `int`*. Le* `...` *est obligatoire.*

Google

Printf en erreur

Nous pouvons utiliser l'astuce ... avec `Printf` (ou une de ses variantes) afin de créer un handler d'erreur personnalisé.

```
func Errorf(fmt string, args ...interface{}) {
   fmt.Fprintf(os.Stderr, "MyPkg: "+fmt+"\n", args...)
   os.Exit(1)
}
```

Nous pouvons l'utiliser ainsi:

```
if err := os.Chmod(file, 0644); err != nil {
   Errorf("couldn't chmod %q: %s", file, err)
}
```

Sortie standard (qui inclut un retour charriot):

```
MyPkg: couldn't chmod "foo.bar": permission denied
```

Google

Append

La fonction native `append`*, qui est utilisée pour faire grandir la taille d'une slice, est variadique. Elle possède en effet cette signature :*

```
append(s []T, x ...T) []T
```

où une slice et `T` *est son type élement. Il retourne une slice avec l'élément* `x` *ajouté à* `s`*.*

```
slice := []int{1, 2, 3}
slice = append(slice, 4, 5, 6)
fmt.Println(slice)
```

affiche `[1 2 3 4 5 6]`

Si possible, `append` *fait grandir la slice en taille.*

Google

Ajouter une slice

Si vous désirez ajouter une slice complète, plutôt que des élements individuels, nous utilisons de nouveau … lors de l'appel.

```
slice := []int{1, 2, 3}
slice2 := []int{4, 5, 6}
slice = append(slice, slice2...)
// … est nécessaire
fmt.Println(slice)
```

Cet exemple aussi affiche `[1 2 3 4 5 6]`

Google

Exercices

Google

Exercice : jour 2

Regarder le package `http`*.*

Ecrire un serveur HTTP qui présente des pages dans un système de fichier, mais passablement transformé , peut-être rot13, peut-être quelque chose d'un peut plus imaginatif. Pouvez-vous rendre substituable la tranformation? Pouvez-vous L'appliquer à votre programme de suite Fibonacci?

Google

Prochaine leçon

- Concurrence et communication

Google