Les bases du langage Go
Partie 4 : structures, méthodes, interfaces

Initiation au développement

BUT informatique, première année

Ce TP a pour but de vous présenter les bases du langage Go. C'est ce langage qui sera utilisé pour réaliser tous les exercices de ce cours d'initiation au développement. Il n'y a pas de prérequis particuliers en dehors du contenu des TP précédents (les bases du langage Go partie 1, partie 2 et partie 3).

Dans tout ce TP un texte avec cet encadrement est un texte contenant une information importante, il faut donc le lire avec attention et veiller à en tenir compte.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

Même si la plupart des exercices de ce TP peuvent vous sembler très faciles, faites les tous sérieusement : cela vous permettra de retenir la syntaxe du langage Go et d'être ainsi plus à l'aise lors des prochains TPs.

Si vous souhaitez avoir un autre point de vue sur l'apprentissage du langage Go, vous pouvez consulter le tutoriel officiel *A Tour of Go*[1] ou le site *Go by Example*[2] en faisant toute fois attention au fait que ces sites sont plutôt faits pour les personnes ayant déjà une bonne expérience de la programmation.

1. `https://tour.golang.org/`
2. `https://gobyexample.com/`

1 Types structurés

Les types de base (int, float, string, bool, etc) ne sont pas toujours suffisants pour représenter toutes les données qu'on voudrait manipuler. On a vu qu'on peut réunir des éléments d'un même type au sein de tableaux ([]int, []float, etc) ou de maps (map[string]int par exemple), mais il pourrait être intéressant de réunir des éléments de types différents, ou d'avoir des structures plus complexes.

Le mot clé struct permet de définir des nouveaux types, constitués de différents *champs* qui peuvent chacun être d'un type différent. Le programme 1 donne un exemple de création d'un nouveau type cuple qui contient deux champs de type int, nommés first et second.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c cuple
    fmt.Println(c)
}
```

Programme 1 – struct.go

Récupérez le fichier struct.go sur MADOC et exécutez-le. Quelle est la valeur par défaut d'une variable de type cuple ? Quel doit être la valeur par défaut d'une variable d'un type structuré quelconque ?

Il est bien sûr possible de fixer soit même les valeurs des champs d'une variable d'un type structuré lorsqu'on la crée. Le programme 2 montre comment faire cela.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c1 cuple = cuple{first: 1, second: 2}
    var c2 cuple = cuple{first: 1}
    var c3 cuple = cuple{second: 2}
    fmt.Println(c1, c2, c3)
}
```

Programme 2 – struct2.go

Récupérez le fichier struct2.go sur MADOC et exécutez-le. Essayez de donner d'autres valeurs aux champs. Que se passe-t-il lorsqu'on ne définit pas les valeurs de tous les champs ? (quelle est la valeur des autres champs ?)

On n'est pas obligé de donner les noms des champs lorsqu'on fixe la valeur d'une variable d'un type structuré (dans le programme ci-dessus on pourrait écrire var c1 cuple = cuple{1, 2}), mais on doit alors obligatoirement définir tous les champs et les définir dans l'ordre où ils sont déclarés dans la structure.

Enfin, on peut accéder indépendamment aux différents champs d'une variable d'un type structuré, et les modifier. Le programme 3 montre comment faire cela.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c, c.first, c.second)

    c.first = 3
    fmt.Println(c, c.first, c.second)
}
```

Programme 3 – struct3.go

Récupérez le fichier struct3.go sur MADOC et exécutez-le. Modifiez le programme pour changer la valeur du champs second et testez-le à nouveau.

Quand on définit un type structuré dans un paquet, si le nom du type commence par une majuscule, alors ce type sera utilisable dans d'autres paquets (à condition d'importer tout ce qu'il faut). De même, les noms des champs ne seront visibles dans d'autres paquets que s'ils commencent par des majuscules. Ainsi, on peut avoir un type dont certains champs sont accessibles à l'extérieur du paquet où il est définit, mais d'autres champs ne sont accessibles que dans ce paquet. Ceci permet de n'exporter que les éléments qui sont pertinents.

Avec type on peut aussi donner des noms à des types déjà existants, par exemple type myint int définit un type myint qui correspond au type int. Ceci sera utile en lien avec la partie suivante sur les méthodes.

2 Méthodes

On appelle *méthode* une fonction particulière qui est fortement associée à un type. Vous en avez déjà rencontré, ce sont les fonctions comme Close() que nous utilisions pour fermer un fichier f en écrivant f.Close(). En pratique, on pourrait remplacer toute méthode par une fonction classique mais les méthodes permettent de mieux structurer le code, en associant des fonctionnalités à des types.

Pour définir une méthode on fait comme pour une fonction normale, mais le type sur lequel elle doit s'appliquer est indiqué avant le nom de la fonction. Le programme 4 donne un exemple de déclaration et d'utilisation d'une méthode.

Dans ce programme on définit une méthode add qui s'applique sur une variable de type cuple, sans prendre de paramètres supplémentaires, et qui retourne un entier (qui est la somme des deux champs

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func (c cuple) add() int {
    return c.first + c.second
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c.add())
}
```

Programme 4 – methode.go

de la variable sur laquelle on applique la méthode).

Récupérez le fichier methode.go sur MADOC et testez-le.

Une méthode peut bien sûr modifier la valeur de la variable à laquelle elle s'applique. Cependant, comme d'habitude, pour cela il faut utiliser un pointeur (le passage d'arguments se fait toujours par valeur). Le programme 5 donne un exemple de cela.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func (c *cuple) exchange() {
    c.first, c.second = c.second, c.first
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c)
    c.exchange()
    fmt.Println(c)
}
```

Programme 5 – methode2.go

Dans ce programme on définit une méthode exchange qui échange les valeurs des deux champs (first et second) d'une variable de type cuple.

Récupérez le fichier methode2.go sur MADOC et testez-le. Que remarquez-vous sur la manière dont on appelle la méthode exchange ?

On ne peut définir des méthodes que sur des types qui sont définit au sein du même paquet. Ainsi, vous ne pouvez par exemple pas définir une méthode sur des entiers (int) mais vous pouvez redéfinir un type qui correspond aux entiers et une méthode sur ce nouveau type.

3 Mise en pratique

On suppose qu'on souhaite gérer les notes de différentes promotions d'étudiants, calculer les moyennes des étudiants et les classer. Pour cela on va définir un type pour nos étudiants et un type pour une promo (qui contiendra plusieurs étudiants). On définira ensuite un certain nombre de méthodes sur les étudiants et sur les promos.

Définir un type Etudiant comprenant un nom, un prénom et une liste de notes.

Définir une méthode Moyenne qui s'applique à un étudiant et calcule sa moyenne.

Définir une méthode AjouteNote qui s'applique à un étudiant et ajoute une note dans sa liste de notes.

Définir un type Promotion comprenant un nom de promo et une liste d'étudiants.

Définir une méthode Classer qui s'applique à une promotion et classe les étudiants en fonction de leur moyenne.

4 Pour aller plus loin : interfaces

Les interfaces permettent de définir des fonctions génériques, qui vont s'appliquer à plusieurs types d'éléments, à condition que ces types partagent un ensemble de méthodes (qui seront utilisées pour les manipuler).

Nous avons déjà utilisé des interfaces sans le savoir quand nous avons créé des bufio.Reader. Ceux-ci peuvent être créés à partir de n'importe quelle variable d'un type qui met en œuvre l'interface io.Reader, c'est-à-dire qui a une méthode Read.

Une interface est en fait simplement une liste de signatures de méthodes avec un nom. On dit qu'un type qui possède toutes ces méthodes implante l'interface. Dès qu'une fonction prend en paramètre le type de l'interface on peut utiliser à la place tout type qui implante l'interface. Le programme 6 montre un exemple de définition et d'implantation d'interface.

Ce programme définit une interface Forme qui décrit une forme générale avec un périmètre. Sur cette interface, une fonction est définie, permettant de comparer les périmètres de deux formes.

Ensuite, deux implantation de l'interface forme sont proposées : par le type Carre et par le type Cercle.

Dans le main on voit ensuite qu'on peut comparer les périmètres de carres et de cercles, en utilisant les fonctions qui ont été définies à partir de l'interface Forme.

Récupérez le fichier interface.go sur MADOC et testez-le.

```go
package main

import (
    "fmt"
    "math"
)

// Interface
type Forme interface {
    Perimetre() float64
}

func PlusGrandPerimetre(f1, f2 Forme) bool {
    return f1.Perimetre() > f2.Perimetre()
}

// Implantation pour les carrés
type Carre struct {
    Cote float64
}

func (c Carre) Perimetre() float64 {
    return 4 * c.Cote
}

// Implantation pour les cercles
type Cercle struct {
    Rayon float64
}

func (c Cercle) Perimetre() float64 {
    return 2 * math.Pi * c.Rayon
}

func main() {
    var ca Carre = Carre{Cote: 5}
    var ce Cercle = Cercle{Rayon: 2.5}
    fmt.Println(PlusGrandPerimetre(ca, ce))
}
```

Programme 6 – interface.go

Implantez l'interface Forme une troisième fois par un type Rectangle qui représente des rectangles. Testez votre implantation.

Sur le modèle de Perimetre, ajoutez une méthode Aire dans les méthodes demandées par l'interface Forme ainsi qu'une fonction PlusGrandeAire qui l'utilise. Mettre à jour les implantations de Carre, Cercle et Rectangle pour prendre en compte cette nouvelle méthode. Testez.