Structures & Pointeurs

Définition

> Simplement
> Type personnalisé représentant une collection de champs

Syntaxe

```
type <NomStruct> struct {
	var1 int
	var2 string
	var3 float64
}
```

Exemple

```
type User struct {
	Name string
	Email string
	Age int
}
```

Déclaration

Il y a 3 types de déclaration possible

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

```
func main() {
  // 1
	var p1 Person
	p1.Name = "Bob"
	p1.Addr.city = "Lyon"
	
	// 2
	p2 := Person{"Paul","Abibi"}

	// 3
	p3 := Person{Name: "Swann"}
}
```

Règles

- Une structure ne peut contenir que des variables
- La règle de visibilité de package s'applique pour :
  - la structure elle-même
  - les variables de la structure

Exercice player

- Définir 2 structure :
  - Avatar
    - Url
  - Player
    - Name
    - Age
    - Avatar
    - password
- Le mot de passe doit avoir un scope privé.
- Créer une fonction de création d’un player, qui ne prend que son nom en argument et initialise la structure avec ce dernier, ainsi qu’un mot de passe par défaut.

Solution

```
type Avatar struct {
	Url string
}
```

```
type Player struct {
	Name     string
	Age      int
	Avatar   Avatar
	password string
}

func New(name string) Player {
	return Player{
		Name:     name,
		password: "defaultpassword",
	}
}
```

Embedded struct

```
type Avatar struct {
	Url string
}

type Player struct {
	Name     string
	Avatar   Avatar
}
```

Un Player a un Avatar

Parfois, on veut exprimer un autre type de relation → Un XXX est un YYY

Dans d'autres langages, cette relation est exprimée par l'héritage

Go préfère la composition avec l'embedded struct :

```
type Avatar struct {
	Url string
}
```

```
type Player struct {
	Name     string
	Avatar // Pas de nom de variables
}

var p Player
p.Url = "https://photodemoi.jpg"
```

Dans ce code, Avatar est embarqué dans le type Player

Player est un Avatar

Receiver function

Grâce à ce fonctionnent, on peut enfin reproduire quasiment à l'identique le comportement d'un objet tel qu'en Java par exemple.

```
type User struct {
	Name string
}
func (u User) SayHello() {
	fmt.Printf("Hello %v!\n", u.Name)
}
```

- **u** est un value receiver pour la méthode SayHello()
- Les champs de **u** sont accessibles pour la fonction
- C'est bien beau ça, mais à quoi ça nous sert ?

```
func main() {
    u := User{"Paul"}
    u.SayHello()
}
```

Lorsqu'on utilise cette technique, notre struct User passé en argument est en réalité "copiée" pour la méthode.

Conséquence : Une méthode avec un value receiver ne peut pas modifier la structure originale. Cela peut permettre de favoriser l'immutabilité en renvoyant une nouvelle instance de notre structure avec les propriétés mise à jour !

Exercice rectangle

- Définir une structure rectangle qui contient
  - Longueur
  - Largeur
- Créer 3 receiver functions pour cette structure :
  - `Area()` : renvoie l'air du rectangle
  - `String()` : Affiche les informations de la structure bien formatées
  - `DoubleSize()` : Renvoie une nouvelle structure du rectangle avec sa taille doublée

> Astuce
> Définir la receiver function `String()` d'une structure viendra surcharger l'affichage par défaut de cette dernière !

Solution

```
package main

import (
	"fmt"
)

type Rect struct {
	Width, Height int
}

func (r Rect) Area() int {
	return r.Width * r.Height
}

func (r Rect) String() string {
	return fmt.Sprintf("Rect ==> width=%v, height=%v", r.Width, r.Height)
}

func (r Rect) DoubleSize() Rect {
	r.Width *= 2
	r.Height *= 2
	return r
}
```

```
func main() {
	r := Rect{2, 4}
	fmt.Printf("Rect area=%v\n", r.Area())
	fmt.Println(r)

	r2 := r.DoubleSize()
	fmt.Println("r", r)
	fmt.Println("r2", r2)
}
```

Pointeurs

En Go, lorsque qu'on passe un paramètre à une fonction, on passe en réalité une copie de cette dernière,

Les pointeurs en Go fonctionnent presque comme en C, à l'exception que nous n'avons pas à gérer l'allocation et la libération de la mémoire.

```
x := -42
s := "Bob"
p := &x // Création d'un pointer vers la variable x
i := *p // Déréférencement de p pour récupérer la valeur de x
```

Manipulations

```
func UpdateVal(val string) {
	val = "value"
}

func UpdatePtr(ptr *string) {
	*ptr = "pointer"
}

func main() {
	i := 1
	var p *int = &i
```

```
	fmt.Printf("i=%v\n", i)
	fmt.Printf("p=%v\n", p)
	fmt.Printf("*p=%v\n", *p)
	fmt.Println("---------------")

	s := "Paul"
	sPtr := &s
	s2 := *sPtr
	fmt.Println("String pointer")
	fmt.Printf("*s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("---------------")

	*sPtr = "Clément"
	fmt.Println("Dereference and update")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("---------------")

	UpdateVal(s)
	fmt.Println("Func UpdateVal()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("---------------")

	UpdatePtr(&s)
	UpdatePtr(sPtr)
	fmt.Println("Func UpdatePtr()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("---------------")
}
```

Pointer Receiver

Comme dit plus tôt, les paramètres de fonctions sont des copies des objets originaux.

Ça vaut aussi pour les fonctions `value receiver` sur les structures. Elles ne peuvent donc que faire de la lecture simple, et ne peuvent pas modifier la structure d'origine.

Grâce aux pointeurs, on peut régler ce problème et le couplant à des `receiver functions`.

```
type Post struct {
	Title     string
	Text      string
	published bool
}

func (p Post) Headline() string {
	return fmt.Sprintf("%v - %v", p.Title, p.Text[:50])
}

func (p *Post) Publish() {
	p.published = true
}

func (p *Post) Unpublish() {
	p.published = true
}

func main() {
	p := Post{
		Title: "Go release",
		Text: "Go is a programming language...",
	}

	fmt.Println(p.Headline())

	fmt.Printf("Post published? %v\n", p.Published())
	p.Publish()
	fmt.Printf("Post published? %v\n", p.Published())
}
```

Enfin, si on souhaite créer une structure directement sous la forme d'un pointeur, on peut faire autrement que :

```
p := Post{
    Title: "Go release",
    Text: "Go is a programming language...",
}
pointer := &p
```

Comme ceci :

```
pythonPost := &Post{
    Title: "Python Intro",
    Text: "Python is an interpreted high-level programming language",
}
```

`pythonPost` est un pointeur.

Revision #1
Created 2022-05-09 19:17:49 UTC by Noé Larrieu-Lacoste
Updated 2022-05-09 19:20:03 UTC by Noé Larrieu-Lacoste