LES STRUCTURES ET LES MÉTHODES DANS LE LANGAGE DE PROGRAMMATION GO

C'est quoi une structure

Jusqu'ici pour définir une variable pouvant contenir plusieurs données on utilisait les tableaux. Le problème avec les tableaux c'est qu'ils vous obligent à utiliser le même type dans tout le tableau , d'ailleurs il nous le fait bien savoir si on tente de rentrer un autre type.

```
package main

import "fmt"

func main() {
    var tableau [2]int
    fmt.Println(tableau)

    tableau[0] = 5 // fonctionne bien car la valeur est du même type que notre tablea
    tableau[1] = "text"
}
```

Erreur :

```
cannot use "text" (type string) as type int in assignment
```

Pour résoudre ce problème on peut entre autres utiliser les structures. Une structure est tout simplement un type de données disponibles sur Go qui est défini par l'utilisateur et qui vous permet de **combiner des éléments de données de différents types**.

Déclarer une structure

Pour déclarer une structure, vous devez utiliser les mots-clés `type` et `struct`.

```
type Nom_de_ma_structure struct {
    nom_var_1 type;
    nom_var_2 type;
    ...
    nom_var_3 type;
}
```

- Le mot-clé `struct` indique à votre compilateur qu'il s'agit d'une structure
- Le mot-clé `type` permet d'associer un nom à votre structure
- Entre les accolades vous placez vos **attributs** (vos variables)

Vous pouvez ensuite utiliser votre structure comme un type variable, exemple :

```
package main

import "fmt"

func main() {

    // déclaration de la structure nommée Personnage
    type Personnage struct {
        nom string
        age int
    }

    var perso Personnage // initialisation d'une variable de type Personnage
    fmt.Println(perso)
}
```

Résultat :

```
{ 0}
```

Vous pouvez surcharger les valeurs par défaut comme suit :

```
package main

import "fmt"

func main() {

    // déclaration de la structure nommée Personnage
    type Personnage struct {
        nom string
        age int
    }

    perso := Personnage{"Hatim", 20} // surchargement des valeurs par défaut
    fmt.Println(perso)
}
```

Résultat :

```
{Hatim 20}
```

Comme résultat nous avons les valeurs par défaut des attributs contenues dans la structure `Personnage`

Accéder et Modifier les attributs de votre structure

Pour accéder et modifier votre attribut de votre structure, nous utiliserons un opérateur d'accès `.` (un point).

```
package main

import "fmt"

func main() {
    type Personnage struct {
        nom string
        age int
    }

    perso := Personnage{"Hatim", 20}
    fmt.Println(perso)
```

```
    // accès juste à l'attribut age de notre structure Personnage
    fmt.Println("je ne veux afficher que l'age du personnage => ", perso.age)

    perso.age = 23 // modification de l'age
    fmt.Println("3 ans plus tard ...", perso.age)
}
```

Résultat :

```
{Hatim 20]
je ne veux afficher que l'age du personnage =>  20
3 ans plus tard ... 23
```

L'intérêt des structures

L'intérêt d'une structure comme son nom l'indique est de mieux **structurer vos variables** et de les **regrouper** dans un seul type pour ainsi leur donner un sens.

Grâce aux structures il est possible de représenter une chose matérielle ou immatérielle qu'elle soit réelle ou fictive avec des caractéristiques (les attributs) et de lui associer des actions sous forme de code Go. Un exemple sera plus parlant :

Imaginons le scénario suivant :

Vous êtes développeur de jeu vidéo et vous avez décidé de créer votre jeu avec le langage de programmation go. Vous commencez alors par créer votre personnage basique et pour ça vous avez besoin de déclarer :

- la vie du personnage
- la puissance du personnage
- le nom du personnage
- Savoir si le personnage est mort ou pas

- L'inventaire du personnage

Pour assouvir votre besoin on va stocker ces variables dans une structures nommée `Personnage`

```
package main

import "fmt"

// création de notre structure Personnage
type Personnage struct {
    nom        string
    vie        int
    puissance  int
    mort       bool
    inventaire [3]string
}

func main() {
    var p1 Personnage // initialisation de ma structure Personnage

    /*
        Initialisation des attributs de mon personnage p1
    */
    p1.nom = "magix"
    p1.vie = 100
    p1.puissance = 20
    p1.mort = false
    p1.inventaire = [3]string{"potion", "bâton", "poison"}

    fmt.Println("Vie du personnage", p1.nom, ":", p1.vie)
    fmt.Println("Puissance du personnage", p1.nom, ":", p1.puissance)

    if p1.mort {
        fmt.Println("Vie du personnage", p1.nom, "est mort")
    } else {
        fmt.Println("Vie du personnage", p1.nom, "est vivant")
    }

    fmt.Println("\nLe personnage", p1.nom, "possède dans son inventaire :", p1.vie)

    for _, item := range p1.inventaire {
        fmt.Println("-", item)
    }

}
```

Résultat :

```
Vie du personnage magix : 100
Puissance du personnage magix : 20
Vie du personnage magix est vivant

Le personnage magix possède dans son inventaire : 100
- potion
- bâton
- poison
```

Structures et pointeurs

Il est possible d'utiliser les pointeurs sur des structures pour modifier directement les valeurs des attributs de notre structure dans une fonction autre que la fonction `main()`.

On va reprendre l'exemple précédent et déclarer une fonction qui permet de surcharger les valeurs par défaut de votre personnage en combinant structures et pointeurs.

```
package main

import "fmt"

type Personnage struct {
    nom        string
    vie        int
    puissance  int
    mort       bool
    inventaire [3]string
}

func main() {
    // initialisation de ma structure Personnage
    var p1 Personnage

    valeurParDefaut(&p1)

    fmt.Println("Vie du personnage", p1.nom, ":", p1.vie)
    fmt.Println("Puissance du personnage", p1.nom, ":", p1.puissance)

    if p1.mort {
        fmt.Println("Vie du personnage", p1.nom, "est mort")
```

```
    } else {
        fmt.Println("Vie du personnage", p1.nom, "est vivant")
    }

    fmt.Println("\nLe personnage", p1.nom, "possède dans son inventaire :", p1.vie)

    for _, item := range p1.inventaire {
        fmt.Println("-", item)
    }

}

/*
    Déclaration d'une fonction utilisant comme paramètre
    un pointeur de structure
*/
func valeurParDefaut(p1 *Personnage) {
    p1.nom = "inconnu"
    p1.vie = 50
    p1.puissance = 10
    p1.mort = false
    p1.inventaire = [3]string{"vide", "vide", "vide"}
}
```

Résultat :

```
Vie du personnage inconnu : 50
Puissance du personnage inconnu : 10
Vie du personnage inconnu est vivant

Le personnage inconnu possède dans son inventaire : 50
- vide
- vide
- vide
```

Information

Pas besoin d'utiliser l'astérisque `*` pour accéder ou modifier une valeur de la structure pointée.

Les méthodes

Une méthode n'est rien d'autre que le nom qu'on donne à une fonction avec un récepteur défini (ici le récepteur est notre structure). Pour faire simple les méthodes sont littéralement des **fonctions liées à votre structure** c'est-à-dire que ce sont des fonctions qui ne peuvent être appelées que par votre structure.

Je vais dans cet exemple déclarer une méthode nommée `affichage()` liée à la structure `Personnage` :

```
package main

import (
    "fmt"
)

type Personnage struct {
    nom        string
    vie        int
    puissance  int
    mort       bool
    inventaire [3]string
}

func main() {
    // initialisation de ma structure Personnage
    var p1 Personnage
    var p2 Personnage

    valeurParDefaut(&p1)
    valeurParDefaut(&p2)

    p1.nom = "barbare"
    p2.nom = "magicien"

    p1.affichage()
    p2.affichage()
}

// déclaration de ma méthode affichage() lié à ma structure Personnage
func (p Personnage) affichage() {
    fmt.Println("--------------------------------------------------")
    fmt.Println("Vie du personnage", p.nom, ":", p.vie)
    fmt.Println("Puissance du personnage", p.nom, ":", p.puissance)

    if p.mort {
        fmt.Println("Vie du personnage", p.nom, "est mort")
    } else {
        fmt.Println("Vie du personnage", p.nom, "est vivant")
```

```
    }

    fmt.Println("\nLe personnage", p.nom, "possède dans son inventaire :", p.vie)

    for _, item := range p.inventaire {
        fmt.Println("-", item)
    }
}

/*
@description : Initialise des valeurs par défaut pour un personnage

@return: void
*/
func valeurParDefaut(p1 *Personnage) {
    p1.nom = "inconnu"
    p1.vie = 50
    p1.puissance = 10
    p1.mort = false
    p1.inventaire = [3]string{"vide", "vide", "vide"}
}
```

Résultat :

```
-------------------------------------------------
Vie du personnage personnage 1 : 50
Puissance du personnage personnage 1 : 10
Vie du personnage personnage 1 est vivant

Le personnage personnage 1 possède dans son inventaire : 50
- vide
- vide
- vide
-------------------------------------------------
Vie du personnage personnage 2 : 50
Puissance du personnage personnage 2 : 10
Vie du personnage personnage 2 est vivant

Le personnage personnage 2 possède dans son inventaire : 50
- vide
- vide
- vide
```

Méthodes et pointeurs

On va combiner les pointeurs avec les méthodes .

Le but principal de cette manipulation ?

Ça va nous permettre de modifier directement les valeurs des attributs de notre structure depuis une méthode. Pour le même exemple je vais créer une méthode `Init()` qui va initialiser les valeurs des attributs de ma structure `Personnage`.

```
package main
```

```
import (
    "fmt"
)

type Personnage struct {
    nom        string
    vie        int
    puissance  int
    mort       bool
    inventaire [3]string
}

func main() {
    // initialisation de ma structure Personnage
    var p1 Personnage
    var p2 Personnage

    p1.Init("barbare", 200, 20, false, [3]string{"épée", "bouclier", "armure"})
    p2.Init("magicien", 100, 40, false, [3]string{"potions", "poisons", "bâton"})
    p1.affichage()
    p2.affichage()
}

/*
@description : Surcharge des valeurs par defaut

@return: void
*/
func (p *Personnage) Init(nom string, vie int, puissance int, mort bool, inventaire [
    p.nom = nom
    p.vie = vie
    p.puissance = puissance
    p.mort = mort
    p.inventaire = inventaire
}

/*
@description : Affiche des informations sur un personnage

@return: void
*/
func (p Personnage) affichage() {
    fmt.Println("------------------------------------------------")
    fmt.Println("Vie du personnage", p.nom, ":", p.vie)
    fmt.Println("Puissance du personnage", p.nom, ":", p.puissance)

    if p.mort {
        fmt.Println("Vie du personnage", p.nom, "est mort")
    } else {
        fmt.Println("Vie du personnage", p.nom, "est vivant")
    }

    fmt.Println("\nLe personnage", p.nom, "possède dans son inventaire :", p.vie)
```

```
    for _, item := range p.inventaire {
        fmt.Println("-", item)
    }
}
```

Résultat :

```
-------------------------------------------------
Vie du personnage barbare : 200
Puissance du personnage barbare : 20
Vie du personnage barbare est vivant

Le personnage barbare possède dans son inventaire : 200
- épée
- bouclier
- armure
-------------------------------------------------
Vie du personnage magicien : 100
Puissance du personnage magicien : 40
Vie du personnage magicien est vivant

Le personnage magicien possède dans son inventaire : 100
- potions
- poisons
- bâton
```

Conclusion

Go n'est pas un langage de programmation purement orienté objet. Les structures et les méthodes sont concepts qui peuvent être implémentés à l'aide de Go et qui se rapprochent de la POO (programmation orientée objet).

Quand nous parlerons de packages personnalisés, on commencera à s'approcher de plus en plus des classes qu'on peut retrouver dans d'autres langages de programmation (C++, Java, Python ...).

Mon but dans ce chapitre est de vous montrer que les structures comme son nom l'indique permettent de mieux structurer notre code.

Par exemple on a réussi à transformer un scénario sous forme de texte en programme écrit Go.

Je vous conseille de reprendre mon exemple sur les personnages et de rajouter des attributs et des méthodes (`Attaquer()`, `Soigner()`, etc ...) sur votre structure Personnage et pourquoi pas recréer le tp précédent à base de structures.