LES INTERFACES DANS LE LANGAGE DE PROGRAMMATION GO

Définition

Une interface est un ensemble de signatures de méthodes qu'une structure peut **implémenter**. Par conséquent, l'interface définit le comportement d'une structure.

Le but principal de l'interface consiste à ne fournir que des **signatures de méthodes** composées :

- Du nom de la méthode
- Des arguments d'entrée
- Des types de retour

Il appartient à la structure de déclarer les méthodes signées dans l'interface et de les implémenter

Démonstration step by step

Déclarer une interface

Pour déclarer une interface nous avons besoin du mot-clé `type` qui permet d'associer un nom à votre interface et du du mo-clé `interface` qui indique à votre compilateur qu'il s'agit d'une interface.

```
package main
```

```
import (
    "fmt"
)

type Forme interface { // création de L'interface Forme
    Air() float64 // signature de la méthode Air()
    Perimetre() float64 // signature de la méthode Perimetre()
}

func main() {
    var f Forme // Initialisation de l'interface Forme
    fmt.Println(f)
}
```

Comme vous pouvez le voir dans une interface on ne met que des signatures de méthodes sans aucun attributs. Pour le moment ces méthodes ne sont pas encore utilisées car pour l'instant je vous montre juste comment déclarer une interface.

Résultat :

```
nil
```

D'après le résultat ci-dessus, nous pouvons voir que la valeur de notre interface est `nil` car il n'existe aucune structure qui implémente l'interface `Forme`.

Utilisation d'une interface

Notre interface `Forme` possède deux signatures :

- `Air()` qui retourne un type `float64`
- `Perimetre()` qui retourne un type `float64`

La structure qui va implémenter l'interface `Forme` doit obligatoirement implémenter les méthodes `Air()` et `Perimetre()` signées dans l'interface `Forme`.

```
package main
```

```
import (
    "fmt"
)

type Forme interface {
    Air() float64
    Perimetre() float64
}

type Rectangle struct {
    largeur  float64
    longueur float64
}

/*
Pour implémenter une interface dans Go, il suffit
d'implémente toutes les méthodes de l'interface. Ici on
implémente la méthode Air() de l'interface Forme.
 */
func (r Rectangle) Air() float64 {
 return r.largeur * r.longueur
}

/*
On implémente la méthode Perimetre() de l'interface Forme
*/
func (r Rectangle) Perimetre() float64 {
 return 2 * (r.largeur * r.longueur)
}


func main() {
    var f Forme
    f = Rectangle{5.0, 4.0} // affectation de la structure Rectangle à l'interface Fo
    r := Rectangle{5.0, 4.0}
    fmt.Println("Type de f :", f)
    fmt.Printf("Valeur de f : %v\n", f)
    fmt.Println("Air du rectangle r :", f.Air())
    fmt.Println("f == r ? ", f == r)
}
```

Il est tout à fait possible d'affecter à notre interface `Forme` la structure `Rectangle` puisque la structure `Rectangle` implémente l'interface `Forme`.

Résultat :

```
Type de f : {5 4}
Valeur de f : {5 4}
Air du rectangle r : 20
f == r ?  true
```

Sur le résultat on peut remarquer que `f == r ?` est à `true` car dans mon exemple `f` et `r` ont le même type et la même valeur.

Je vais intentionnellement supprimer la méthode `Air()` de la structure `Rectangle` pour ainsi vous prouvez qu'il est vraiment **obligatoire** d'implémenter toutes les méthodes de l'interface `Forme`.

```
package main

import (
 "fmt"
)

type Forme interface {
 Air() float64
 Perimetre() float64
}

type Rectangle struct {
 largeur  float64
 longueur float64
}

/* Pas de méthode Air */

func (r Rectangle) Perimetre() float64 {
 return 2 * (r.largeur * r.longueur)
}

func main() {
 var f Forme
 f = Rectangle{5.0, 4.0}
 r := Rectangle{5.0, 4.0}
 fmt.Println("Type de f :", f)
 fmt.Printf("Valeur de f : %v\n", f)
 fmt.Println("Air du rectangle r :", f.Air())
 fmt.Println("f == r ? ", f == r)
}
```

Erreur :

```
.cannot use Rectangle literal (type Rectangle) as type Forme in assignment:
Rectangle does not implement Forme (missing Air method)
invalid operation: f == r (mismatched types Forme and Rectangle)
```

L'erreur ci-dessus indique clairement que, pour implémenter une interface avec succès, vous devez implémenter toutes les méthodes déclarées par l'interface et notre compilateur nous le fait bien savoir on nous marquons qu'il manque la méthode `Air()` *"missing Air method"*.

On va dans cet exemple créer deux structures `Cercle` et `Rectangle` qui implémente l'interface `Forme` :

```
package main

import (
 "fmt"
 "math"
)

type Forme interface {
 Air() float64
 Perimetre() float64
}

/*----------------Début Structure Rectangle----------------*/
type Rectangle struct {
 largeur  float64
 longueur float64
}

func (r Rectangle) Air() float64 {
 return r.largeur * r.longueur
}

func (r Rectangle) Perimetre() float64 {
 return 2 * (r.largeur * r.longueur)
}
/*----------------Fin Structure Rectangle----------------*/


/*----------------Début Structure Cercle----------------*/
type Cercle struct {
 rayon float64
}

func (c Cercle) Air() float64 {
 return math.Pi * c.rayon * c.rayon
}
func (c Cercle) Perimetre() float64 {
 return 2 * math.Pi * c.rayon
}
```

```
/*----------------Fin Structure Cercle-------------------*/



//Fonction qui prend compte un paramètre de type  Cerle ou Rectangle
func AirPerimetrePresentation(f Forme) {
 fmt.Println("- Air :", f.Air())
 fmt.Println("- Perimetre :", f.Perimetre())
}

func main() {
 var r Forme = Rectangle{5.0, 4.0}
 var c Forme = Cercle{5.0}

 fmt.Println("Cercle :")
 AirPerimetrePresentation(r)
 fmt.Println("\nrectangle :")
 AirPerimetrePresentation(c)
}
```

Résultat :

```
Cercle :
- Air : 20
- Perimetre : 40

rectangle :
- Air : 78.53981633974483
- Perimetre : 31.41592653589793
```

Le résultat ci-dessus nous montre que l'interface `Forme` peut être déclaré en tant que `Cercle` ou `Rectangle`

Conclusion

Les interfaces sont une sorte de contrat, ils permettent de créer des comportements génériques: si plusieurs structures doivent obéir à des comportements particuliers, alors on créé une interface décrivant ces comportements. Ces structures devront ainsi obéir strictement aux signatures de l'interface, sans quoi la compilation ne se fera pas.

Grâce à eux on peut obtenir un code beaucoup plus claire et facile à maintenir.