Les structures

Énoncé du problème

- Nous avons vu que les tranches permettaient de regrouper des éléments **de même type**, qui étaient ensuite accessibles (pour être consultés et/ou modifiés) par leurs **indices**.

- Comment faire pour regrouper des éléments de types différents ?

- Par ailleurs, les indices sont un moyen peu lisible pour désigner un élément : ne serait-il pas possible d'affecter un nom à un élément ?

- Bref, pourrait-on disposer d'un moyen permettant de créer de nouveaux types, un peu comme les types d'entités que l'on manipule en BD ?

Exemple de besoins

- On voudrait pouvoir disposer d'un **type** permettant de représenter les coordonnées **x** et **y** d'un point du plan cartésien... Il n'existe pas déjà en Go.
    - Une tranche **coord** de deux entiers pourrait faire l'affaire, mais utiliser **coord[0]** et **coord[1]** pour désigner **x** et **y** serait peu lisible et sujet aux confusions.
- On voudrait pouvoir disposer d'un **type** permettant de représenter des personnes qui, elles-mêmes aurait un composant d'un **type** permettant de représenter une adresse postale... Ces types n'existent pas déjà en Go.
    - Une tranche ne peut pas convenir puisqu'il faut stocker des chaînes (pour le nom et le prénom), une adresse (d'un type qui reste à définir), etc.

Le mot-clé **struct**

- La solution à notre problème s'appelle **struct** : un mot-clé permettant de créer un **nouveau type** composé de champs ayant des types éventuellement différents (exactement comme les types d'entités vus en BD) :

```
type coords struct {      // un type pour représenter des coordonnées cartésiennes
    x, y int
}

type adr struct {         // un type pour représenter une adresse postale
    rue, cp, ville string
}

type personne struct {    // un type pour représenter une personne
   prenom, nom string
   genre          byte    // 'm' ou 'f'
   adresse        adr
}
```

Variables struct : création et initialisation

- Une fois que l'on a créé un type **struct**, on peut créer des variables ou des paramètres de ce type, le renvoyer à partir d'une fonction, créer des tranches de ce type, etc.

```
var (
   point1, point2  coords          // deux coordonnées
   moi             personne        // une personne
   classe          []personne      //  une collection de personnes
)
```

- On initialise une variable struct en utilisant les accolades et en fournissant des valeurs à chacun de ses champs **(qui valent "zéro" par défaut)** :

```
point1 = coords{1, 2}                  // x = 1, y = 2
point2 = coords{y:4, x:1}              // x = 1, y = 4

moi = personne{"Éric", "Jacoboni", adresse{"21 rue de la pompe", "31000", "Toulouse}}
classe = append(classe, moi)
```

Variables struct : complément sur l'initialisation

- Dans l'exemple précédent, on remarque qu'il **faut** préciser le type de struct lorsqu'on une variable simple : c'est logique puisque Go ne peut pas déduire ce type si on ne met que des accolades... Comment interpréter `point1 = {2, 3}` ??? `{2, 3}` pourrait être autre chose qu'une coordonnée...

- En revanche, **dans le cas d'une tranche de structures**, Go sait que cette tranche ne pourra contenir que des éléments de ce type de structure, il n'y a donc pas besoin de le préciser lorsqu'on l'initialise : on peut donc écrire :

  classe = []personne{**{**"Joe", "Dalton", **adresse**{"fort knox", "123", "texas"}**}**, {...}, ... }

- Mais, ici, il faut quand même préciser que le 3e champ est une adresse car Go ne peut pas le déduire... ça ne marche que pour les éléments des tranches.

Variables struct : accès aux champs

- On utilise la notation pointée pour accéder à un champ particulier :

```
if point1.x == 1 {
   point1.y = point2.y * 3              // point1.y vaut désormais 12
}

if moi.adresse.cp[:2] == "31" {    // j'habite en Haute-Garonne...
   ...
}

if classe[0].adresse.ville == "Toulouse" {  // j'habite à Toulouse...
  ...
}
```

- Un champ a toutes les propriétés de son type et peut être utilisé partout où son type peut l'être (par exemple, **x** et **y** de **coords** sont des **int**, donc on peut faire avec eux tout ce qu'on peut faire avec un **int**...)

Exemples de structures (1/2)

```
package main

import "fmt"

// Le type couple implémente un couple d'entiers
type couple struct {
	x, y int
}

// sumDiviseurs renvoie la somme des diviseurs propres de n
func sumDiviseurs(n int) int {
	res := 1	// 1 fait toujours partie des diviseurs propres
	lim := n / 2
	for x := 2; x <= lim; x++ {
		if n%x == 0 {
			res += x
		}
	}
	return res
}
```

```
// amis renvoie la liste des couples de nombres amis <= lim
func amis(lim int) []couple {
	res := []couple{}
	var b int	// évite une réallocation à chaque tour de boucle
	for a := 2; a <= lim; a++ {
		b = sumDiviseurs(a)
		if b > a && sumDiviseurs(b) == a {
			res = append(res, couple{a, b})
		}
	}
	return res
}

func main() {
	var limite int
	fmt.Print("Entrez une limite : ")
	fmt.Scan(&limite)
	fmt.Println(amis(limite))
}
```

Voir ce lien pour une version optimisée de sumDiviseurs

Exemples de structures (2/2)

```
package main

import "fmt"

type personne struct {
        id              int
        nom, prenom string
}

type employe struct {
        idPers int
        sal    float64
}

// searchPersonne recherche une personne par son id dans liste
func searchPersonne(id int, liste []personne) (personne, error) {
        for _, pers := range liste {
                if pers.id == id {
                        return pers, nil
                }
        }
        return personne{}, fmt.Errorf("Personne inconnue !")
}
```

```
// printEmployes effectue une "jointure" entre des employés et des personnes
// afin d'afficher les caractéristiques des employés
func printEmployes(from []employe, join []personne) {
        for _, employe := range from {
                pers, err := searchPersonne(employe.idPers, join)
                if err == nil {
                        fmt.Printf("%v %v\n", pers.prenom, pers.nom)
                } else {
                        fmt.Println(err)
                }
        }
}

func main() {
        personnes := []personne{{1, "Jacoboni", "Eric"},
                                {2, "Comparot", "Cathy"}, {3, "Adreit", "Françoise"}}
        employes := []employe{{1, 2500}, {2, 3000}, {42, 2500}, {3, 3500}}

        printEmployes(employes, personnes)
        fmt.Println()

        personnes = append(personnes, personne{42, "Isnard", "Stéphane"})
        printEmployes(employes, personnes)
}
```

Voir ce lien

Exercice

- Soient les trois relations suivantes (cf. cours de BD) :

  Produits(numProd : entier, libelleProd : chaine, prixProd : réel)
  Lignes_Commandes(numCommande : entier, numProd : entier, quantite : entier)
  Commandes(numCommande : entier, dateCommande : date)

- Créer les types Go permettant de représenter ces trois relations.

- Comment représenter en Go l'ensemble des produits et l'ensemble des commandes ? Donner un exemple et écrire un programme Go permettant d'afficher la liste des produits et la liste des commandes.