Modularização: Structs

Nas linguagens de programação de hoje em dia, há o conceito de **Orientação a objetos**, permitindo que declaremos **classes** e instanciemos **objetos** a partir delas. **Go** não tem esse conceito.

Go não tem o conceito de classes

Go permite criarmos **Types** (tipos de dados) complexos. Estes tipos são representados pela **interface Type** (veremos mais adiante).

Um **Type** é uma **struct** (um tipo complexo) como este:

```
package main

import (
    "fmt"
    "reflect"
)

type Student struct {
    name string
}

func main() {
    var newStudent Student
    newStudent.name = "John Doe"
    fmt.Println(newStudent, reflect.TypeOf(newStudent))
}
...
{John Doe} main.Student
```

Abra o projeto s13_typesample.

Criamos uma **struct** chamada **Student** contendo apenas um membro: O nome do estudante. Gostaria de chamar a atenção para o uso de letra maiúscula no nome **Student** pois permite que o tipo seja exportado para outros pacotes. Tudo o que você declarar com letra maiúscula no escopo do pacote, será exportado para outros pacotes, mas é preciso usar a declaração completa (com **var**).

Depois de executarmos, vemos o conteúdo do objeto **Student** e o seu nome de tipo, que é o nome do pacote e o nome da struct.

Ponteiros e memória

Go tem ponteiros. Podemos obter o endereço de uma variável e armazená-lo em um ponteiro com o operador "&":

```
pst := &newStudent
fmt.Println(pst.name, reflect.TypeOf(newStudent))
```

```
...
John Doe main.Student
```

O ponteiro **pst** passa a apontar para o objeto (localização na memória) da variável **newStudent**. Em outras linguagens, como C++, precisaríamos *desreferenciar* explicitamente utilizando o operador "*":

```
fmt.Println((*pst).name, reflect.TypeOf(newStudent))
```

Quando declaramos uma variável usando uma **struct** criamos nova área de memória para ela:

```
var other Student
other.name = "Other Student"
fmt.Println(pst.name, other.name)
...
John Doe Other Student
```

Como podemos ver, cada variável tem sua própria área de memória. Na verdade, é um ponteiro para uma área de memória criada.

E podemos inicializar uma struct usando um **literal struct** que contenha os valores na mesma ordem dos tipos da struct:

```
p := Student{"Pamela"}
fmt.Println(p.name)
```

Métodos

Podemos criar **funções** dentro de **structs**, mais conhecidas como **métodos** em outras linguagens de programação. Por exemplo, se quiséssemos criar uma *classe* **Course** (curso) que contenha uma descrição e a lista de estudantes, faríamos assim em Java:

```
class Course {
        String description;
        List<Student> students;
}
```

Em **Go** um array pode ser de um tipo **struct**, mas os arrays não estáticos! Seu tipo é declarado junto com seu tamanho! Só **slices** são dinâmicos. Então, como declaramos a **struct**?

```
// Course This is a course
type Course struct {
    description string
```

```
    students    []Student
}
```

*(Antes de toda **struct** coloque um comentário com o nome e a algum bla-bla-bla, caso contrário, o compilador vai ficar te perturbando)*

Criei uma variável **students** que é um **slice** (um vetor sem declaração de tamanho) e vou criar um método para adicionar novos estudantes:

```
func (c *Course) register(s Student) {
    c.students = append(c.students, s)
}
```

Um **método** é uma função que recebe um **receiver** (antes do seu nome). Um **receiver** é um inicializador da função e é o que a trasforma em um método. O **receiver** pode ser o valor ou um ponteiro para uma instância da **struct**. Se quisermos apenas usar as propriedades da **struct**, informamos o **receiver** como uma variável comum, porém, se quisermos modificar alguma propriedade da **struct**, informamos como **ponteiro**.

Neste caso, o método **register()** recebe um **ponteiro** para um **Course** (note o asterisco), portanto, ele pode modificar qualquer propriedade do **Course**. E ele o faz aumentando o **slice de Students** com o novo **Student**.

Agora, vejamos como usamos o **Course** e como registramos novos alunos:

```
engineering := Course{"Engineering", make([]Student, 0)}
engineering.register(newStudent)
engineering.register(p)
fmt.Println(engineering)
...
{Engineering [{John Doe} {Pamela}]}
```

Herança

Go não tem o conceito de herança, onde uma classe herda as propriedades e métodos de outra. Ela tem o conceito de composição, no qual uma classe pode incluir outra e acessar as propriedades desta. Por exemplo, vamos criar uma modalidade de curso EAD:

```
// EadCourse This is another course type
type EadCourse struct {
    course  Course
    website string
}
...
ead := EadCourse{Course{"New EAD", make([]Student, 0)},
"http://eadcourse"}
```

```
ead.course.register(newStudent)
fmt.Println(ead)
...
{ {New EAD [{John Doe}]}  http://eadcourse }
```

Métodos em outros tipos

Podemos declarar métodos em outros tipos que criamos. Por exemplo, vou criar um método para arredondar um **float64** em um determinado número de casas decimais:

```
// NewFloat float with steroids
type NewFloat float64

func (nf NewFloat) roundBy(places float64) NewFloat {
    nplaces := math.Pow(10.00, places)
    return NewFloat(math.Round(float64(nf)*nplaces) / nplaces)
}
...
var n NewFloat = 5.0293019384
fmt.Println(n.roundBy(2))
fmt.Println(n.roundBy(3))
...
5.03
5.029
```

Como podem ver, eu criei um método para uma classe **NewFloat** derivada de **float64**.

Desafio

Crie um método para remover alunos de um curso.

Dica: Use **slice** para remover alunos (append(slice[:i], slice[i+1:]...)) e use um **for** para descobrir o índice (i) do aluno a ser removido.