Interfaces

Uma interface é um tipo de dados, assim como uma **struct**, que serve para declarar um **conjunto de métodos** que um objeto deve implementar.

Além dos métodos declarados na própria **struct** (ou no tipo, como já vimos) podemos acrecentar os métodos de uma interface. Assim, podemos separar objetos de acordo com seu comportamento.

Já que não temos o conceito de **herança** precisamos de uma maneira para generalizar o comportamento de objetos similares. Por exemplo, o que todos os **veículos** possuem em comum?

- Ligar: TurnOn
- Desligar: TurnOff
- Mover: Move
- Parar: Stop

Podemos criar uma interface que abstraia esses comportamentos. Veja nosso **exemplo**:

```
// Vehicle this represents a vehicle behavior
type Vehicle interface {
    TurnOn() bool
    TurnOff() bool
    Move(direction int, speed int) bool
    Stop()
}
```

Uma interface é um tipo, portanto, podemos criar variáveis a partir dela:

```
var newVehicle Vehicle
fmt.Println(newVehicle, reflect.TypeOf(newVehicle))
...
<nil> <nil>
```

Como a interface é apenas um conjunto de métodos, ela não possui valores estáticos e, se não inicializada com um objeto compatível, o resultado é **nil** (nulo).

Uma interface pode assumir como valores quaisquer objetos que a implementem. Em nosso exemplo: **Car** e **Truck** implementam a interface **Vehicle**. Como eu sei disso? Bem, olhe o **código-fonte** e note que ambas as **structs** declaram TODOS os métodos de **Vehicle**, portanto, elas implementam esta interface.

```
var newVehicle Vehicle
newVehicle = Car{"Dodge"}
newTruck := Truck{"GMC"}
newVehicle = newTruck
```

Isso só é possível se as structs implementarem TODOS os métodos da interface.

Podemos utilizar quaisquer métodos da interface em qualquer objeto, e o método invocado será o declarado na **struct**, ou seja, **polimorfismo** implícito:

```
newVehicle = Car{"Dodge"}
newVehicle.TurnOn()
...
Turning the car on
...
newTruck := Truck{"GMC"}
newVehicle = newTruck
...
Turning the Truck on
```

Neste exemplo, o método **TurnOn()** invocado é sempre o do tipo concreto do objeto, já que a interface não possui implementação dos métodos.

Empty interface

Todos os tipos implementam a **empty interface** ou **interface{}**, por exemplo:

```
var aVehicle interface{}
fmt.Println(aVehicle, reflect.TypeOf(aVehicle))
...
<nil> <nil>
```

Uma variável ou um argumento do tipo **interface{}** pode assumir qualquer objeto, já que todos a implementam:

```
aVehicle = newTruck
fmt.Println(aVehicle, reflect.TypeOf(aVehicle))
...
{GMC} main.Truck
```

Um objeto pode implementar mais de uma interface, por exemplo, todo objeto **Car** também implementa a interface **Gas**:

```
// Gas this represents a gasoline fueled object
type Gas interface {
    FillUp()
}
...
// FillUp this method Fills the tank of a Car
func (c Car) FillUp() {
```

```
    fmt.Println("Filling the car's tank")
}
```

Quando temos uma variável **interface** podemos verificar quais interfaces o objeto apontado por ela implementa. Para isto, usamos a sintaxe: `i.(Type)`:

```
newVehicle = newCar
v2, a2 := newVehicle.(Car)
fmt.Println(v2, a2)
v3, a3 := newVehicle.(Gas)
fmt.Println(v3, a3)
...
{Mustang} true
{Mustang} true
```

A sintaxe **i.(Type)** testa se o objeto apontado por uma **interface** implementa outra interface, e retorna o valor e um **bool** indicando a resposta.

Ponter vs value receiver

Já vimos que um método pode ter um **pointer** ou um **value** receiver. Por exemplo, este método recebe um valor, portanto, nada pode modificar no objeto:

```
// FillUp this method Fills the tank of a Car
func (c Car) FillUp() {
    fmt.Println("Filling the car's tank")
}
```

Porém, na lição **L04** que podemos ter métodos cujo receiver é um ponteiro:

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

Isso ocorre porque o método **register()** altera propriedades do objeto **Course**. Quando temos métodos de objetos, tanto faz o tipo de receiver. Mas, quando usamos interfaces, podemos ter problemas. Veja o código abaixo:

```
// Computer this represents a Computer behavior
type Computer interface {
    TurnOn() bool
    TurnOff() bool
}
```

```
// Laptop this struct represents a Laptop
type Laptop struct {
    model string
    on    bool
}

// TurnOn this method turns a laptop on
func (l *Laptop) TurnOn() bool {
    fmt.Println("Turning the laptop on")
    l.on = true
    return true
}

// TurnOff this method turns a laptop off
func (l *Laptop) TurnOff() bool {
    fmt.Println("Turning the laptop off")
    l.on = false
    return true
}
...
var c Computer = Laptop{"Asus", false}
```

O código resultaria em erro. Na última linha veríamos um erro do tipo: "Laptop does not implement Computer (TurnOff method has pointer receiver)". Quando usamos métodos de **struct** tanto faz o tipo de receiver, mas, quando queremos usar com interfaces, temos que saber isso. Se um método tiver um **pointer** receiver, temos que usar a notação de ponteiro:

```
var c Computer = &Laptop{"Asus", false}
fmt.Println(c)
c.TurnOn()
```

Desafio

Pegue o desafio do capítulo anterior e declare os métodos de **Course** em uma interface.