

Métodos e interfaces em Go

André Murbach Maidl

Escola Politécnica

Pontifícia Universidade Católica do Paraná

Antes de tudo

- Os slides sobre a linguagem Go funcionam apenas como guias para as aulas.
- Não esqueça de consultar a documentação da linguagem e das APIs das suas bibliotecas:
<http://golang.org/pkg/>.
- Se tiver alguma dúvida entre em contato com o professor.
- Lembre-se também que tanto o *Google* quanto o *DuckDuckGo* são seus amigos! 😊

Objetivos da aula

- Entender como definir e usar métodos e interfaces em Go.
- Compreender exemplos de uso de algumas interfaces comuns:
 - Stringer;
 - error;
 - Reader;
 - Handler.

Métodos

- Em Go podemos definir métodos para estruturas, apesar da linguagem não incluir classes.
- Diferente de Java e C++, em Go precisamos definir explicitamente o *receiver* (*this/self*) de um método.
- Isto é, o *receiver* (*this/self*) de um método deve ser informado entre a palavra chave **func** e o nome do método.

Exemplo

- No exemplo a seguir, o método `Ponto2String` tem um *receiver* do tipo `Ponto` chamado `p`.

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func (p Ponto) Ponto2String () string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func main () {
    p := Ponto{0.0, 5.0}
    fmt.Println(p.Ponto2String())
}
```

Métodos são funções

- Note que métodos são apenas funções que possuem um *receiver* como argumento.

Exemplo

- Note que a definição de `Ponto2String` como uma função normal não afeta a funcionalidade.

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func Ponto2String (p Ponto) string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func main () {
    p := Ponto{0.0, 5.0}
    fmt.Println(Ponto2String(p))
}
```

Métodos e tipos não estruturados

- Podemos definir métodos para qualquer tipo de dado que declaramos em um pacote.
- Isto é, os métodos não são exclusivos das estruturas.
- No entanto, não podemos definir métodos para tipos que foram declarados em outros pacotes.

Exemplo

- Note que precisamos declarar um sinônimo para o tipo **string** se quisermos declarar métodos que operam sobre strings.

```
package main

import "fmt"

type MyString string

func (s MyString) Print () {
    fmt.Printf("MyString = %s\n", s)
}

func main () {
    s := MyString("teste 1 2 3")
    s.Print()
}
```

Métodos e ponteiros

- Geralmente usamos ponteiros no tipo do *receiver*.
- Primeiro, queremos evitar copiar valores nas chamadas de métodos, pois isso seria ineficiente com estruturas grandes.
- Segundo, queremos que os métodos possam alterar o valor apontado pelo *receiver*.

Exemplo

- No exemplo a seguir, o método Move usa o *receiver* p, do tipo *Ponto, para alterar o seu conteúdo.

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func (p Ponto) Ponto2String () string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func (p *Ponto) Move (x, y float64) {
    p.x += x
    p.y += y
}

func main () {
    p := Ponto{0.0, 5.0}
    p.Move(10.0, 5.0)
    fmt.Println(p.Ponto2String())
}
```

Métodos e indireção (&)

- Funções que recebem um ponteiro como argumento devem receber uma referência:

```
var p Ponto = Ponto{0.0, 5.0}  
Move(p, 10.0, 5.0)    /* Erro */  
Move(&p, 10.0, 5.0)    /* OK */
```

- No entanto, métodos que recebem um ponteiro como o *receiver* podem receber tanto uma cópia quanto uma referência:

```
var p Ponto = Ponto{0.0, 5.0}  
p.Move(10.0, 5.0)      /* OK */  
var r *Ponto = &p  
r.Move(5.0, 10.0)      /* OK */
```

Métodos e indireção (&)

- Funções que recebem um ponteiro como argumento devem receber uma referência:

```
var p Ponto = Ponto{0.0, 5.0}
Move(p, 10.0, 5.0)    /* Erro */
Move(&p, 10.0, 5.0)    /* OK */
```

- No entanto, métodos podem receber um ponteiro como o receptor, tanto uma cópia que

Açúcar sintático de Go:

```
p.Move(10.0, 5.0)
==
(&p).Move(10.0, 5.0)
```

```
var p Ponto = Ponto{0.0, 5.0}
p.Move(10.0, 5.0)    /* OK */
var r *Ponto = &p
r.Move(5.0, 10.0)    /* OK */
```

Métodos e indireção (*)

- Funções que recebem um valor como argumento devem receber uma cópia:

```
var p Ponto = Ponto{0.0, 5.0}
fmt.Println(Ponto2String(p))    /* OK */
fmt.Println(Ponto2String(&p))  /* Erro */
```

- No entanto, métodos que recebem um valor como o *receiver* podem receber tanto uma cópia quanto uma referência:

```
var p Ponto = Ponto{0.0, 5.0}
fmt.Println(p.Ponto2String())  /* OK */
var r *Ponto = &p
fmt.Println(r.Ponto2String())  /* OK */
```

Métodos e indireção (*)

- Funções que recebem um valor como argumento devem receber uma cópia:

```
var p Ponto = Ponto{0.0, 5.0}
fmt.Println(Ponto2String(p))    /* OK */
fmt.Println(Ponto2String(&p))  /* Erro */
```

- No entanto, métodos que recebem um valor como o *receiver* podem receber uma cópia quanto uma referência

```
var p Ponto = Ponto{0.0, 5.0}
fmt.Println(p.Ponto2String())
var r *Ponto = &p
fmt.Println(r.Ponto2String()) /* OK */
```

Açúcar sintático de Go:

`r.Ponto2String()`

`==`

`(*r).Ponto2String()`

O que usar: cópia ou referência?

- Relembrando: geralmente vamos usar ponteiros (passagem por referência) no tipo do *receiver* pelos seguintes motivos:
 1. Queremos evitar copiar valores nas chamadas de métodos, pois isso seria ineficiente com estruturas grandes.
 2. Queremos que os métodos possam alterar o valor apontado pelo *receiver*.

Exemplo

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func (p *Ponto) Ponto2String () string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func (p *Ponto) Move (x, y float64) {
    p.x += x
    p.y += y
}

func main () {
    p := Ponto{0.0, 5.0}
    fmt.Println("Antes de mover p =", p.Ponto2String())
    p.Move(10.0, 5.0)
    fmt.Println("Depois de mover p =", p.Ponto2String())
}
```

Interfaces

- Um tipo interface é definido por um conjunto de métodos.
- Um tipo implementa uma interface se ele implementa todos os métodos dessa interface.
- A vantagem de termos interfaces implícitas é podermos separar a declaração e a implementação em pacotes diferentes, ou seja, um não depende do outro.

A interface `Stringer`

```
type Stringer interface {  
    String() string  
}
```

- A interface `Stringer` é declarada pelo pacote `fmt`.
- Sendo assim, podemos definir a sua implementação em nossos pacotes para podermos descrever como imprimir os nossos tipos.

Exemplo

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func (p Ponto) String () string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func (p *Ponto) Move (x, y float64) {
    p.x += x
    p.y += y
}

func main () {
    p := Ponto{0.0, 5.0}
    fmt.Println("Antes de mover p =", p)
    p.Move(10.0, 5.0)
    fmt.Println("Depois de mover p =", p)
}
```

A interface error

```
type error interface {  
    Error() string  
}
```

- Go declara por padrão a interface `error`, a qual podemos usar para descrever como imprimir os erros que os nossos pacotes geram.
- Podemos usar o tipo `error` para sinalizar quando uma função terminou sua execução sem sucesso.
- Nesse caso, podemos testar se o erro é `nil` ou não, para decidirmos se a função terminou ou não com sucesso.

Exemplo

```
package main

import "fmt"

type ErroDivisaoPorZero int

func (e ErroDivisaoPorZero) Error () string {
    return "divisão por zero"
}

func idiv (x int, y int) (int, int, error) {
    if y == 0 {
        return 0, 0, ErroDivisaoPorZero(x)
    } else {
        return x / y, x % y, nil
    }
}

func main () {
    x, y := 9, 2
    q, r, err := idiv(x,y)
    if err == nil {
        fmt.Printf("%d/%d = (%d,%d)\n", x, y, q, r)
    } else {
        fmt.Println("Erro:", err)
    }
}
```

Exemplo

```
package main

import "fmt"

type ErroDivisaoPorZero int

func (e ErroDivisaoPorZero) Error () string {
    return "divisão por zero"
}

func idiv (x int, y int) (int, int, error) {
    if y == 0 {
        return 0, 0, ErroDivisaoPorZero(x)
    } else {
        return x / y, x % y, nil
    }
}

func main () {
    x, y := 9, 0
    q, r, err := idiv(x,y)
    if err == nil {
        fmt.Printf("%d/%d = (%d,%d)\n", x, y, q, r)
    } else {
        fmt.Println("Erro:", err)
    }
}
```

Qual é o resultado se trocarmos o 2 por 0?

A interface Reader

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

- O pacote `io` define a interface `io.Reader`, a qual podemos usar para implementar leitores de *streams* de dados.
- A função `Read` preenche `p` e retorna o número de dados preenchidos mais um erro.
- Podemos também usar o tipo `Reader` para modificar uma *stream* de dados.

Exemplo

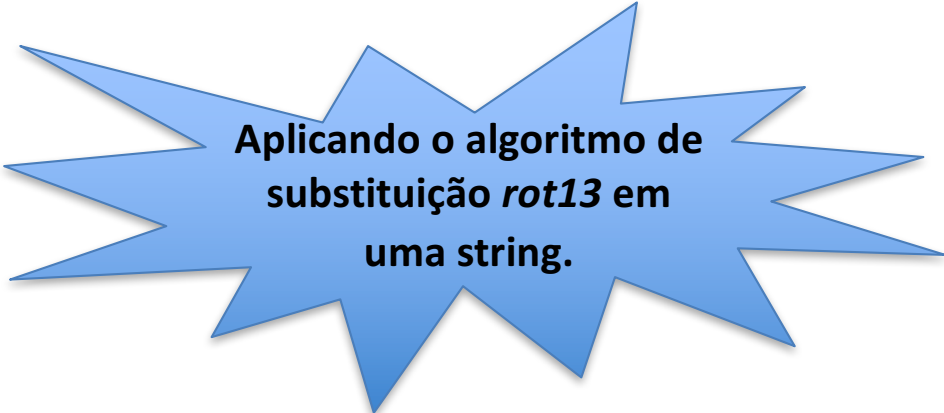
```
package main

import "io"
import "os"
import "strings"

type rot13Reader struct {
    r io.Reader
}

func (self *rot13Reader) Read(p []byte) (n int, err error) {
    n, err = self.r.Read(p)
    for i, b := range(p) {
        if b <= 'Z' && b >= 'A' {
            p[i] = (b - 'A' + 13) % 26 + 'A'
        } else if b >= 'a' && b <= 'z' {
            p[i] = (b - 'a' + 13) % 26 + 'a'
        }
    }
    return n, err
}

func main() {
    s := strings.NewReader("Qrpvsenfgr b frterqb!\n")
    r := rot13Reader{s}
    io.Copy(os.Stdout, &r)
}
```



Aplicando o algoritmo de
substituição *rot13* em
uma string.

A interface Handler

```
type Handler interface {  
    ServeHTTP(w ResponseWriter, r *Request)  
}
```

- O pacote `net/http` define a interface `http.Handler`, a qual podemos usar para implementar servidores HTTP.
- Por exemplo, podemos implementar um servidor HTTP que imprime “*Olá, Mundo!*” no navegador.

Exemplo

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

type String string

func (s String) ServeHTTP (w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Olá, Mundo!\n")
}

func main () {
    var s String
    err := http.ListenAndServe("localhost:8080", s)
    if err != nil {
        log.Fatal(err)
    }
}
```

Exercício 1 (0,1)

- No programa a seguir, o que acontece se tirarmos o `*` na declaração do método `Move`?

```
package main

import "fmt"

type Ponto struct {
    x, y float64
}

func (p *Ponto) Ponto2String () string {
    return fmt.Sprintf("{x = %.1f, y = %.1f}", p.x, p.y)
}

func (p *Ponto) Move (x, y float64) {
    p.x += x
    p.y += y
}

func main () {
    p := Ponto{0.0, 5.0}
    p.Move(10.0, 5.0)
    fmt.Println(p.Ponto2String())
}
```

Exercício 2 (0,4)

- Considere a seguinte estrutura:

```
type Arvore struct {  
    info int  
    esq *Arvore  
    dir *Arvore  
}
```

- Complete o programa da próxima transparência para que implemente os métodos Pares, Folhas e Igual, os quais operam sobre o tipo Arvore.
- Execute a **main** da próxima transparência para testar os métodos implementados.

Exercício 2 (0,4)

```
package main
import "fmt"
type Arvore struct {
    info int
    esq *Arvore
    dir *Arvore
}
/* retorna a quantidade de nós que armazenam números pares */
func (a *Arvore) Pares () int {
    return 0
}
/* retorna a quantidade de folhas de uma árvore binária */
func (a *Arvore) Folhas () int {
    return 0
}
/* compara se duas árvores binárias são iguais */
func (a *Arvore) Igual (b *Arvore) bool {
    return false
}
func main () {
    a1 := &Arvore{4, nil, nil}
    a2 := &Arvore{2, nil, a1}
    a3 := &Arvore{5, nil, nil}
    a4 := &Arvore{6, nil, nil}
    a5 := &Arvore{3, a3, a4}
    a := &Arvore{1, a2, a5}
    b := &Arvore{7, a2, a5}
    fmt.Println(a.Pares() == 3)
    fmt.Println(a.Folhas() == 3)
    fmt.Println(a.Igual(a) == true)
    fmt.Println(a.Igual(b) == false)
}
```

Exercício 3 (0,1)

- Modifique o exemplo da transparência sobre a interface `Handler` para adicionar o tipo a seguir:

```
type Struct struct {  
    Saudacao string  
    Fulano string  
}
```

- Defina o funcionamento do método `ServeHTTP` sobre o tipo `Struct` para que imprima `Saudacao` seguido de `Fulano`.
- Continua na próxima transparência...

Exercício 3 (0,1)

- Use a função `http.Handle` para tratar os dois *handlers* que você tem no seu servidor:
 - Um deve ser `"/string"`, para continuar imprimindo "Olá, Mundo!" quando acessamos o endereço <http://localhost:8080/string>
 - O outro deve ser `"/struct"`, para imprimir o conteúdo de uma estrutura (que pode ser estática) quando acessamos o endereço <http://localhost:8080/struct>
- Use o navegador para testar a sua solução.