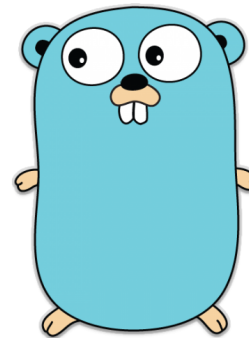


---

# Go



César Henrique Bernabé  
João Mario Silva

---

# Linguagens de Programação

---

# Introdução

---

- Go que nasceu nos escritórios do Google em 2007 como um projeto interno e em novembro de 2009 foi lançado como um projeto de código aberto.
- Inspirada em C, porém possui características como Orientação a Objetos e Coletor de Lixo.
- Última versão 1.4.2
- Não entra nas 50 linguagens mais utilizadas do ranking do TIOBE, mas em 2013 já esteve na 23 posição.



# Instalação

---

- Download e instruções de instalação estão disponíveis em <https://golang.org/> (inglês) ou em <http://www.golangbr.org/> (português).



# Executando um programa

---

Basta executar, a partir do terminal:

```
$go run programa.go
```

ou

```
$go build
```



# Hello World

---

```
package main

import "fmt"

func main() {
    fmt.Println("Olá, 世界")
}
```



# Comentários

---

- Comentários em Go são feitos da mesma forma que C, onde:
  - // comentário de uma única linha
  - /\*
    - \* comentário de múltiplas
    - \* linhas
    - \*/



# Palavras Reservadas em Go

---

<b>break</b>	<b>default</b>	<b>func</b>	<b>interface</b>	<b>select</b>
<b>case</b>	<b>defer</b>	<b>go</b>	<b>map</b>	<b>struct</b>
<b>chan</b>	<b>else</b>	<b>goto</b>	<b>package</b>	<b>switch</b>
<b>const</b>	<b>fallthrough</b>	<b>if</b>	<b>range</b>	<b>type</b>
<b>continue</b>	<b>for</b>	<b>import</b>	<b>return</b>	<b>var</b>



# Tipagem

---

- Go possui tipagem forte e estática, porém introduz uma forma curta de declaração de variáveis, baseada em inferência de tipos, isso evita redundância e produz códigos sucintos.





# Tipagem

---

```
//tipagem por inferência
```

```
x := 5
```

```
a, b := 8, 12
```

```
//tipagem estática
```

```
var y int
```

```
var k float64
```



# Valores Numéricos

---

- Go possui uma vasta gama de tipos numéricos:
  - **uint8, uint16, uint32, uint64** (inteiros sem sinal de 8, 16, 32 ou 64-bits)
  - **int8, int16, int32, int64** (inteiros (com sinal) de 8, 16, 32 ou 64-bits)
  - **float32, float64** (o conjunto de todos pontos flutuantes IEEE-754 32-bit)
  - **complex64, complex128** (o conjunto de todos números complexos de float32 partes reais e imaginárias)
  - **byte** (sinonimia para uint8), **rune** (sinonimia para int32)
  - **uint, int** (tanto 32 ou 64 bits)
  - **uintptr** (um inteiro sem sinal grande o suficiente para armazenar os bits não-interpretados de um ponteiro)



# Valores

---

- Strings
- Booleanos
- Arrays
- Structs
- Ponteiros
- Mapas
- Channel



# Arrays e Slices

---

Slice é uma abstração baseada em Array, onde:

- Arrays são listas com tamanho pré-determinado.
- Slices possuem tamanho dinâmico, podendo crescer indefinidamente.
- Quando usados como argumento ou retorno de funções, são passados por referência. Nos outros casos são passados como cópia.



# Arrays

---

- A definição de um array deve especificar seu tipo e tamanho:

```
var a [4]int
a[0] = 1
i := a[0]
// i == 1
```



# Slices

---

- Slices são ponteiros para segmentos de um array:

```
letters := []string{"a", "b", "c", "d"}
```

- Ou podem ser criados usando a função `make`, que possui a seguinte assinatura:

```
func make([]T, len, cap) []T
```

Onde `T` é o tipo dos elementos do slice.



# Maps

---

Mapas seguem a estrutura *map[KeyType]ValueType*:

```
capt := map[string]string{
    "GO" : "Goiânia",
    "PB" : "João Pessoa",
    "ES" : "Vitória"
}
```

```
//com uso de make
m = make(map[string]int)

//mapas vazios
m = map[string]int{}

//ambas as formas tem o
//mesmo resultado!
```



# Range

---

- Slices, arrays e maps podem ser "fatiados" usando a palavra chave *range* (como em Python):

```
x := []int{0,1,2,3}

y := x[1:]

for i,v := range y { //range retorna
    fmt.Println(y[i]) //tanto o indice
    fmt.Println(v)    //quanto o valor
}
```





# Range\*

---

Obs.: Se o índice não for utilizado, deve-se usar o identificador vazio `_`

```
x := []int{0,1,2,3}

for _,v := range x {
    fmt.Println(v)
}
```



# Saída padrão

---

- Necessária importação da biblioteca fmt.

```
import (
    "fmt"
)

func main( ) {
    fmt.Println("O João é
um cara bem legal..")
}
```

Outras funções de fmt:

- //mesma sintaxe de C:  
fmt.Printf("Numero: %d", 5)
- fmt.Print("Nao insere a quebra de linha")



# Entrada Padrão

---

- Também são gerenciados pelo pacote fmt:

```
var i int
_, err := fmt.Scanf("%d", &i)
```



# Estruturas de Repetição

---

- If/Else
- For
- Switch
- ~~While~~



# If/Else

---

```
if num := 9; num < 0 {  
    fmt.Println(num, "é negativo")  
} else if num < 10 {  
    fmt.Println(num, "tem 1 dígito")  
} else {  
    fmt.Println(num, "tem vários dígitos")  
}
```

Obs.: uma instrução pode preceder condições; qualquer variável declarada nessa condição está disponível para todas as ramificações.



# For

---

Única condição (nesse caso funciona como While, Go não tem while):

```
i := 1
  for i <= 3 {
    fmt.Println(i)
    i = i + 1
  }
```



# For

---

Range (como em arrays, slices e maps):

```
for i, v := range y {  
    fmt.Println(v)  
    fmt.Println(y[i])  
}
```



# For

---

Várias condições (clássico):

```
for j := 7; j <= 9; j++ {  
    fmt.Println(j)  
}
```





# For

---

Sem condição:

```
for {  
    fmt.Println("loop")  
    break  
}
```



# Switch

---

## Switch Básico:

```
switch i {
    case 1:
        fmt.Println("um")
    case 2:
        fmt.Println("dois")
    case 3:
        fmt.Println("três")
}
```

## Cases separados por vírgulas:

```
switch time.Now().Weekday() {
    case time.Saturday, time.
    Sunday:
        fmt.Println("é final
de semana")
    default:
        fmt.Println("é dia de
semana")
}
```



# Blocos e Escopos

---

- Go possui suporte a blocos aninhados.
- Variáveis declaradas dentro de blocos são somente visíveis a este.



# Funções

---

```
func mais(a int, b int) int {  
    return a + b  
}
```

```
func main() {  
    res := mais(1, 2)  
    fmt.Println("1+2 =", res)  
}
```

Onde:

```
func nome_da_funcao (parametro  
int, entrada String) retorno {  
    corpo da funcao  
}
```

- Funções sem retorno apenas omite-se o tipo de retorno na definição da função, nessa caso elas são chamadas *procedimentos*.



# Funções - Múltiplos Retornos

---

```
func vals() (int, int) {  
    return 3, 7  
}  
  
func main() {  
    a, b := vals() //usando os dois valores de retorno  
    fmt.Println(a)  
    fmt.Println(b)  
    c := vals() //usando o primeiro valor de retorno  
    fmt.Println(c)  
}
```



# Funções - Múltiplos Retornos

---

```
func vals() (int, int) {  
    return 3, 7  
}  
  
func main() {  
    a, b := vals() //usando os dois valores de retorno  
    fmt.Println(a)  
    fmt.Println(b)  
    _, c := vals() //usando o segundo valor de retorno  
    fmt.Println(c)  
}
```



# Funções Variádicas

---

```
func soma(nums ...int) {  
    fmt.Print(nums, " ")  
    total := 0  
    for _, num := range nums  
    {  
        total += num  
    }  
    fmt.Println(total)  
}
```

```
func main() {  
    soma(1, 2)  
    soma(1, 2, 3)  
  
    nums := []int{1, 2, 3, 4}  
    soma(nums...)  
}
```



# Ponteiros

---

- Go suporta uso de ponteiros, porém não suporta aritmética entre eles.





# Ponteiros

---

```
func zeroval(ival int) {
    ival = 0
}
func zeroptr(iptr *int) {
    *iptr = 0
}
```

```
func main() {
    i := 1
    fmt.Println("inicial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    fmt.Println("ponteiro:", &i)
}
```



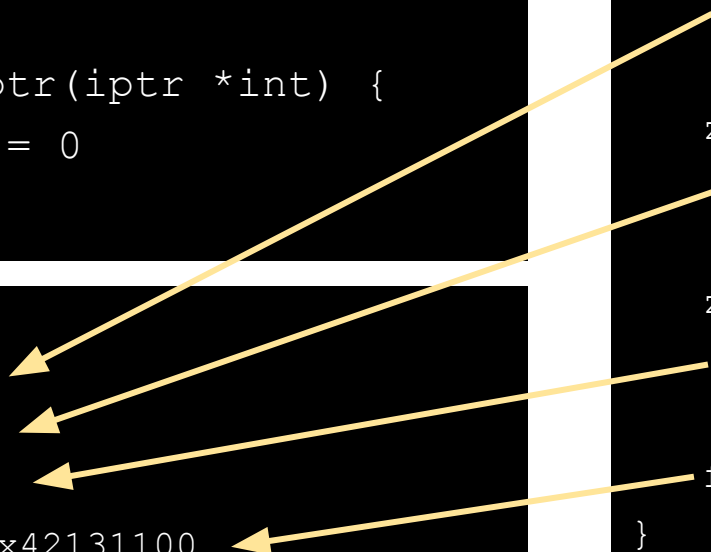
# Ponteiros

---

```
func zeroval(ival int) {  
    ival = 0  
}  
func zeroptr(iptr *int) {  
    *iptr = 0  
}
```

```
Saida:  
inicial: 1  
zeroval: 1  
zeroptr: 0  
ponteiro: 0x42131100
```

```
func main() {  
    i := 1  
    fmt.Println("inicial:", i)  
    zeroval(i)  
    fmt.Println("zeroval:", i)  
    zeroptr(&i)  
    fmt.Println("zeroptr:", i)  
    fmt.Println("ponteiro:", &i)  
}
```



# Visibilidade

---

A visibilidade em pacotes é definida pela primeira letra do nome da função ou variável global:

- Se primeira letra for **maiúscula**, então é função/variável **pública**.
- Se primeira letra for **minúscula**, então é função/variável **privada**.



# Orientação a Objeto

---

A orientação a objetos em Go é simulada por meio de structs. Isso será discutido nos próximos slides.



# Sistema de Tipos

---

- Go não permite herança, mas os tipos implementam interfaces implicitamente. Na prática, considera-se essa implementação implícita como *duck typing*.
  - *Duck typing*: se faz “quack” como um pato, e anda como um pato, então provavelmente é um pato.
- 



# Interfaces

---

Baseadas na implementação de structs, interfaces em Go são definidas como coleções de assinaturas de métodos.



# Interfaces

---

```
type geometria interface {
    area() float64
}
type quadrado struct {
    largura, altura float64
}
type círculo struct {
    raio float64
}
```

```
func (q quadrado) area()
float64 {
    return q.largura *
        s.altura
}
func (c círculo) area()
float64 {
    return math.Pi * c.raio
        * c.raio
}
```



# Interfaces

---

```
func medir(g geometria) {  
    fmt.Println(g)  
    fmt.Println(g.area())  
}
```

```
func main() {  
    q := quadrado{largura:  
3, altura: 4}  
  
    c := círculo{raio: 5}  
  
    medir(q)  
    medir(c)  
}
```





# Interfaces

---

- É importante destacar que todas as funções previamente mencionadas foram escritas no mesmo arquivo. A tipagem de cada “objeto” foi feita dinamicamente. Tivemos então o chamado **POLIMORFISMO**
- Go não dá suporte a excessões.



# Erros

---

- Go comunica erros através de um valor explícito, separado:

```
func f1(arg int) (int, error) {  
    if arg == 42 {  
        return -1, errors.New("não pode trabalhar com 42")  
    }  
    return arg + 3, nil  
}
```



# Erros

---

- Por convenção, erros são o último parâmetro de retorno de uma função e possuem tipo próprio (*error*).
- Também é possível usar 0 no lugar de *nil*.



# Closures

---

- Go suporta funções anônimas, que podem formar closures.
- Funções anônimos são úteis quando se quer definir uma função em linha sem ter de nomeá-la.



# Closures

```
func intSeq() func() int {  
    i := 0  
    return func() int {  
        i += 1  
        return i  
    }  
}
```

```
func main() {  
    nextInt := intSeq()  
  
    fmt.Println(nextInt())  
    fmt.Println(nextInt())  
    fmt.Println(nextInt())  
  
    newInts := intSeq()  
    fmt.Println(newInts())  
}
```



# Closures

```
func intSeq() func() int {  
    i := 0  
    return func() int {  
        i += 1  
        return i  
    }  
}
```

```
func main() {  
    nextInt := intSeq()  
  
    fmt.Println(nextInt())  
    fmt.Println(nextInt())  
    fmt.Println(nextInt())  
}
```

```
$ go run closures.go
```

```
1  
2  
3  
1
```



# Concorrência

---

*“Do not communicate by sharing memory,  
share memory by communicating.”*



# Concorrência

---

- Em Go, a concorrência é expressa através de Goroutines, e a comunicação entre “processos” é feita através de canais (memória compartilhada).





# Goroutines

---

- Uma Goroutine é uma *thread* leve de execução.
- Para criar uma Goroutine basta chamar a função com a palavra chave *go* como prefixo

```
Func funcao() { ... }
```

```
go funcao()
```



# Goroutines

---

- O escalonador de Go não tira proveito do hardware.
- Go só usa 1 core de CPU, a não ser que esse numero seja alterado usando a SVC `runtime.GOMAXPROCS(NCPUS)`.



# Canais

---

- Canais são pipes que conectam goroutines concorrentes. É possível enviar valores para os canais de uma goroutine e receber esses valores em outra goroutine.



# Canais

---

```
func main() {
    //cria um novo canal
    mensagens := make(chan string)

    //escreve do canal
    go func() { mensagens <- "ping" }()

    //le do canal
    msg := <-mensagens
    fmt.Println(msg)
}
```



# Sincronização de Canais

---

- A comunicação por um canal é síncrona (bloqueante).
- Sincronização é feita com uso de ferramentas da biblioteca "sync".
- Funções da biblioteca "sync" podem ser chamadas com a criação de uma variável do tipo `WaitGroup`.



# Canais

---

```
func main() {  
    var controle sync.WaitGroup  
  
    //indica que uma nova goroutine devera ser sincronizada  
    controle.Add(1)  
  
    go executar(&controle)  
  
    //espera até que todas goroutines estejam finalizadas  
    controle.Wait()  
}
```



# Avaliação da Linguagem

---

Critério	C	Java	Go
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	S	Go possui vários pacotes que estendem suas funções, como exemplo o pacote HTTP, que fornece funcionalidades de manipulações de endereços web
Aprendizado	Não	N	
Eficiência	Sim	P	



# Avaliação da Linguagem

---

Critério	C	Java	Go
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Não
Aprendizado	Não	N	
Eficiência	Sim	P	

Assim como em C (e ao contrário de Java), Go não apresenta tratamento de exceções.





# Avaliação da Linguagem

---

Critério	C	Java	Go
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Não
Aprendizado	Não	Não	Sim
Eficiência	Sim	Parcial	Sim

Comparada a C e Java, Go é mais simples de aprender (não usa aritmética de ponteiros, não obriga o tratamento de exceções, etc).



# Avaliação da Linguagem

---

Critério	C	Java	Go
Aplicabilidade	Sim	Parcial	
Confiabilidade	Não	Sim	
Aprendizado	Não	Não	
Eficiência	Sim	Parcial	Parcial

Efficiency Partial: Control of resources against goroutines



# Avaliação da Linguagem

---

Critério	C	Java	Go
Portabilidade	Não	Sim	Sim
Método de projeto	Estruturado	Sim	Sim
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Sim	Sim	Parcial

Possível compilar o mesmo código em vários sistemas diferentes.



# Avaliação da Linguagem

---

Critério	C	Java	Go
Portabilidade	Não	Sim	Sim
Método de projeto	Estruturado	OO	Estruturado e OO "simulado"
Evolutibilidade	Não	Sim	Parcial
Reusabilidade	Sim	Sim	

Justamente por possuir OO simulada, consideramos Evolutibilidade em Go como Parcial.



# Avaliação da Linguagem

---

Critério	C	Java	Go
Portabilidade	Não	Sim	Parcial
Método de projeto	Estruturado	OO	Reusabilidade é feita com o uso de Interfaces
Evolutibilidade	Não	Sim	
Reusabilidade	Parcial	Sim	Parcial



# Avaliação da Linguagem

---

Critério	C	Java	Go
Integração	Sim	Parcial	Parcial
Custo	Depende da aplicação	Depende da ferramenta	Pode ser linkado com C ou C++



# Avaliação da Linguagem

---

Critério	C	Java	Go
Integração	Sim	Parcial	Parcial
Custo	Depende da aplicação	Depende da ferramenta	Depende da ferramenta

Go é Open Source!



# O Trabalho

---

- Implementação para o sistema de gerenciamento da Revista EngeSoft, o sistema aceita submissões de artigos, com cadastro de autores, revisores e notas para artigos, então exibe um relatório sobre a edição.





# O Trabalho - Estrutura Artigos

---

```
type Artigo struct{
    titulo string
    contato Autor
    listaAutores[] Autor
    listaRevisores[] Revisor
    media float64
    revisoesEnviadas int
}
func (art *Artigo) AdicionaAutor(autor Autor){
    art.listaAutores = append(art.listaAutores, autor)
}
```



# O Trabalho - Estrutura Autor

---

```
type Edicao struct {
    volume, numero int
    dataPublicacao time.Time
    tema string
    chefe Revisor
    artigos []Artigo
    codArtigos map[int]int
}
```



# O Trabalho - Estrutura Edicao

---

```
type Edicao struct {  
    volume, numero int  
    dataPublicacao time.Time  
    tema string  
    chefe Revisor  
    artigos []Artigo  
    codArtigos map[int]int  
}
```



# O Trabalho - Estrutura Revisor

---

```
type Revisor struct {  
    nome string  
    email string  
    senha int  
    instituicao string  
    endereco string  
    temas []string  
    artigosRevisados int  
    notasAtribuidas float64  
}
```



# O Trabalho - Funções - Sorting

---

```
type ByName []Revisor

func (a ByName) Len() int { return len(a) }

func (a ByName) Swap(i, j int) { a[i], a[j] = a[j], a[i] }

func (a ByName) Less(i, j int) bool {
return a[i].nome < a[j].nome }
```



# O Trabalho - Funções - Leitura CSV

---

```
func readCSVFile(fileName string, fieldNumber int)(rowData
[][]string, err error){
    file, err := os.Open(fileName)
    reader := csv.NewReader(file)

    reader.Comma = ';'
    rowData, err = reader.ReadAll()

    return rowData, nil
}
```



# O Trabalho - Funções - Escrita Arq

---

```
func escreverArquivo(nomeArquivo, conteudo string) {  
    d1 := []byte(conteudo)  
    err := ioutil.WriteFile(nomeArquivo, d1, 0644)  
    check(err)  
}
```



# Referências

---

- <http://golang.org/>
  - <http://www.golangbr.org/>
  - <http://goporexemplo.golangbr.org/>
  - Concorrência e sincronização na linguagem Go - Daniel Alfonso Gonçalves de Oliveira
  - Livro “Programando em Go” - Caio Filipini.
-



# Obrigado

---

