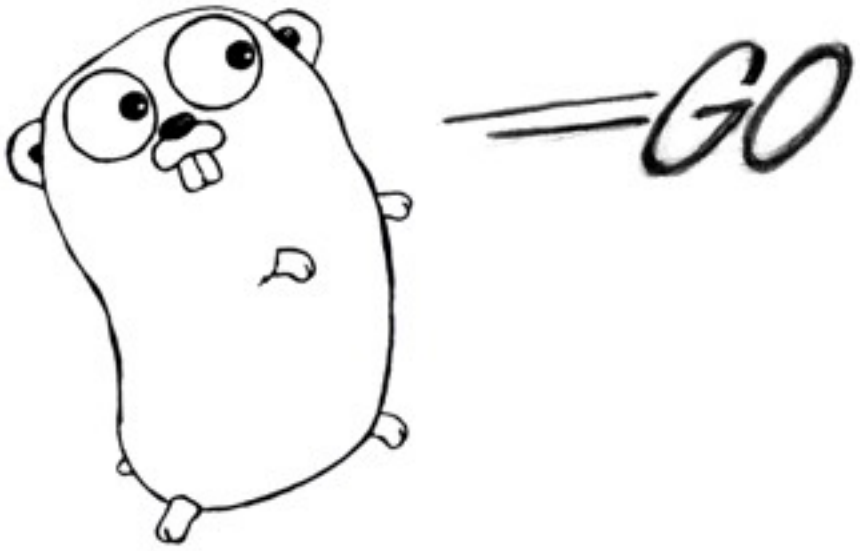

# Linguagens de Programação

**Professor:**
Vítor Souza

**Grupo:**
Gustavo Tavares
Linicker Harison
Rodrigo Barcellos

# Seminário



# Histórico e Paradigma

- Criad apelo Google
- Lançada em novembro de 2007
- Código livre em 2009
- Primeira versão estável em 2012 (Go1)
- Linguagem compilada (não usa VM ou interpretador)
- Foco na eficiência, legibilidade e concorrência
- Possui coletor de lixo

# Por que foi criada?

- Criada parare solver problemas do Google com:
  - C++, Python e Java
  - Muitas linhas de código
  - Sistema distribuído
  - Uma única árvore (um makefile na raiz para compilar tudo)
  - Númeroe norme de máquinas (concorrência)
  -
- O desenvolvimento no Google pode ser lento, mas é eficiente!

- Objetivos:
  - Ter eficiência em seus softwares
  - Ter um código simples de construir
  - Poder fazer softwares confiáveis

# Tipos de dados

- Go é fortemente tipada

# Tipos de dados

## Primitivos

```
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // pseudônimo para uint8

rune // pseudônimo para int32
     // representa um ponto de código Unicode

float32 float64

complex64 complex128
```

# Tipos de dados

# Compostos

## Structs

- A *struct* é uma coleção de campos.
- Uma declaração **type** define um novo tipo de dado.
- Os campos de uma *struct é* acessado através de um ponto.

```
type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    // Imprime {1 2}
    fmt.Println(v)
    v.X = 4
    // Imprime 4
    fmt.Println(v.X)
}
```

```
type Vertex struct {
    X, Y int
}

var (
    p = Vertex{1, 2}  // tem tipo Vertex
    q = &Vertex{1, 2} // tem tipo *Vertex
    r = Vertex{X: 1}  // Y:0 é implicito
    s = new(Vertex)   // X:0 e Y:0 (*Vertex)
)

func main() {
    //Imprime {4 2} &{3 2} {1 0} &{0 0}
    fmt.Println(p, q, r, s)
}
```

# Tipos de dados

# Compostos

## Ponteiros

- Go tem ponteiros, mas não há aritmética de ponteiros.
- Campos struct podem ser acessados através de um ponteiro struct.

```
type Vertex struct {
    X int
    Y int
}

func main() {
    p := Vertex{1, 2}
    q := &p //var q *Vertex = &p
    q.X = 1e2
    // Imprime {100 2}
    fmt.Println(p)
}
```

# Tipos de dados

# Compostos

## Slices (arrays)

- Um slice aponta para uma matriz de valores e também inclui um comprimento.
- []Té um slice com elementos do tipo T.
- Slices podem ser fatiados gerando novos slices que apontam para a mesma matriz

```
func main() {
    p := []int{2, 3, 5, 7, 11, 13}
    //p == [2 3 5 7 11 13]
    fmt.Println("p ==", p)
    for i := 0; i < len(p); i++ {
        fmt.Printf("p[%d] == %d\n",
            i, p[i])
    }
}
```

# Tipos de dados

# Compostos

## Maps (dicionários)

- Inserir ou atualizar um elemento no map:

```
m[key] = elem
```

- Recuperar um elemento:

```
elem = m[key]
```

- Excluir um elemento:

```
delete(m, key)
```

- Testar que uma chave está presente com dois valores:

```
elem, ok = m[key]
```

# Definição de variáveis

- A instrução **var** declara uma lista de variáveis
- O tipo deve ser passado.

```
package main

import "fmt"

var x, y, z int
var c, python, java bool

func main() {
    fmt.Println(x, y, z, c, python, java)
}
```

# Definição de variáveis

- A declaração **var** pode incluir inicializadores, um por variável. Se um inicializador está presente, o tipo pode ser omitido; a variável terá o tipo do inicializador.

```
package main

import "fmt"

var x, y, z int = 1, 2, 3
var c, python, java = true, false, "no!"

func main() {
    fmt.Println(x, y, z, c, python, java)
}
```

# Definição de variáveis

- **Dentro de uma função** a instrução de atribuição curta := pode ser utilizada em lugar de uma declaração var com o tipo implícito.
- Fora de uma função cada estrutura começa com uma palavra-chave e não é possível usar o :=

```
package main

import "fmt"

func main() {
    var x, y, z int = 1, 2, 3
    c, python, java := true, false, "no!"

    fmt.Println(x, y, z, c, python, java)
}
```

# Declaração de Constantes

- Constantes são declaradas como variáveis, mas com a palavra-chave const.
- Constantes podem serseqüências de caracteres, booleanos, ou valores numéricos.

```
package main

import "fmt"

const Pi = 3.14

func main() {
    const World = "世界"
    fmt.Println("Hello", World)
    fmt.Println("Happy", Pi, "Day")

    const Truth = true
    fmt.Println("Go rules?", Truth)
}
```

# Declaração de Constantes

## Constantes Numéricas

- As constantes numéricas são valores de alta precisão.
- Uma constante sem tipo tem o tipo necessário para o seu contexto.

```
package main

import "fmt"

const (
    Big   = 1 << 100
    Small = Big >> 99
)

func needInt(x int) int { return x*10 + 1 }
func needFloat(x float64) float64 {
    return x * 0.1
}

func main() {
    fmt.Println(needInt(Small))   // 21
    fmt.Println(needFloat(Small)) // 0.2
    fmt.Println(needFloat(Big))   // 1.2676506002282295e+29
    // Erro: constant Big é maior que int (overflow)
    //fmt.Println(needInt(Big))
}
```

# Keywords

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

# Operadores

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| + | & | += | &= | && | == | != | ( | ) |
| - | \| | -= | \|= | \|\| | < | <= | [ | ] |
| * | ^ | *= | ^= | <- | > | >= | { | } |
| / | << | /= | <<= | ++ | = | := | , | ; |
| % | >> | %= | >>= | -- | ! | ... | . | : |
| | &^ | | &^= | | | | | |

# Pacotes

- Cada programa Go é composto de pacotes.
- Programas começam rodando pelo pacote main.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Happy", math.Pi, "Day")
}
```

# Gerenciamento de memória

- Utiliza coletor de lixo
- Mecanismo de Parallel Mark and Sweep

# Expressões e Comandos

## Comandos Condicionais

### If

- O if básico parece com o que o de C ou Java fazem, exceto que as ( ) se foram (não são nem mesmo opcionais) e os { } são obrigatórios.

```
package main

import (
    "fmt"
    "math"
)

func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}

func main() {
    //1.4142135623730951 2i
    fmt.Println(sqrt(2), sqrt(-4))
}
```

# Expressões e Comandos

## Comandos Condicionais

### If

- A instrução **if** pode começar com uma breve declaração antes de executar a condição.
- Variáveis declaradas pela instrução são válidas somente no escopo até o final do **if**.

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    //fmt.Printf(v) -- undefined: v
    return lim
}
```

# Expressões e Comandos

## Comandos Iterativos

## For

- Go tem apenas uma estrutura de laço, o **for**.
- O for básico parece com o que o de C ou Java fazem, exceto que as ( ) se foram (não são nem mesmo opcionais) e os { } são obrigatórios.
- O **while** do C é escrito com **for** em Go.

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum) // 45
}
```

```
package main

import "fmt"

func main() {
    sum := 1
    //for ; sum < 1000;
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum) // 1024
}
```

# Expressões e Comandos

## Comandos Iterativos

## For

- O range do laço for itera sobre uma *slice* ou *map*.

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

- Pode-se ignorar o índice ou o valor atribuindo _.

```
for i := range pow {
    pow[i] = 1 << uint(i)
}
for _, value := range pow {
    fmt.Printf("%d\n", value)
}
```

# Modularização

## Funções

- A função pode ter zero ou mais argumentos.
- O tipo vem após o nome da variável (facilita a leitura).
- Quando dois ou mais consecutivos parâmetros da função nomeados compartilhar um tipo,pode-se omitir o tipo de todos, menos o último.

```
//Adicione dois parâmetros do
//tipo int e retorne um int.
func add(x int, y int) int {
    return x + y
}
```

```
func add(x, y int) int {
    return x + y
}
```

# Modularização

## Funções

- Uma função pode retornar qualquer número de resultados

```
func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

- O resultados retornados podem ser nomeados

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

# Modularização

## Funções

- Funções também são valores.

```
func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }

    fmt.Println(hypot(3, 4))
}
```

# Modularização

## Funções

- Funções também tem closures completos.

```
func adder() func(int) int {
    //Cada chamada de adder tera
    //sua propria sum internamente
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}
```

# Go Não Possui Classes

- Go não possui classes ou objetos. Para fazer programas orientados a objetos deve-se utilizar de artifícios disponibilizado pela linguagem, como:
  - Estruturas, como classes
  - Interfaces, para herança.

# Métodos

- Go não tem classes. No entanto, pode-se definir métodos em com tipo *struct*.
- O método receptor aparece em sua lista de argumentos entre a própria palavra-chave *func* e o nome do método.

```
type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    fmt.Println(v.Abs())
}
```

# Métodos

- Pode-se definir métodos em qualquer tipo definido dentro do package, não apenas em structs.
- Não se pode definir um método em um tipo de outro pacote, ou em um tipo básico.

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

# Métodos

## Ponteiro receptor

- Métodos podem ser associados a um tipo nomeado ou um ponteiro para um tipo nomeado
- Evita copiar o valor em cada chamada de método
- O método passa a modificar o valor que seu receptor aponta

```
type Vertex struct {
    X, Y float64
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    // Imprime: &{15 20} 25
    fmt.Println(v, v.Abs())
}
```

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    // Imprime: &{3 4} 5
    fmt.Println(v, v.Abs())
}
```

# Interfaces

- Um tipo de interface é definida por um conjunto de métodos.
- Um valor de tipo de interface pode conter qualquer valor que implementa esses métodos.

# Interfaces

```
type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // MyFloat implementa Abser
    a = &v // *Vertex implementa Abser
//  a = v  -- Vertex, nao implementa Abser

    fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

# Polimorfismo

## Coerção

- Não possui polimorfismo de coerção.
- Conversões de tipo devem ser explicitadas pelo programador.
- Exceção para constantes, que caso não tenham tipo explicitamente definido, é tipada baseada no contexto que é utilizada.

```
func identidade(a int8){
    fmt.Printf("%d",a);
}

func main() {
    var a int16 = 5
    identidade(int8(a))
//  identidade(a) -- Erro!
}
```

# Polimorfismo

## Sobrecarga

- Possui sobrecarga em seus operadores.
- Não possui sobrecarga de funções.

```
func identidade(a int8){
    fmt.Printf("%d",a);
}

// Erro!
//func identidade(a int16){
//    fmt.Printf("%d",a);
//}

func main() {

    var a,b int8 = 1,2
    var x,y float32 = 3.2,5.8

    a = a + b //a = somaInt8(a,b)
    x = x + y //x = somaFloat32(x,y)

    // Imprime: 3 9.000000
    fmt.Printf("%d %f",a,x);
}
```

# Polimorfismo

## Paramétrico

- Não possui polimorfismo paramétrico.
- Solução: utilizar uma interface que aceite todos os tipo desejados.

# Polimorfismo

## Inclusão

- Não possui polimorfismo de inclusão.
- Todo o tipo que implementa um conjunto de métodos definidos em uma interface, satisfaz esta interface.

# Polimorfismo

Inclusão

```
type Veiculo interface{
    andar(km int)
}

type Carro struct{
    passageiros int
}

type Caminhao struct{
    carga int
}

func (c *Carro) andar(km int){
    fmt.Printf("Carro andando %d km, com %d passageiros\n",km,c.passageiros)
}

func (c *Caminhao) andar(km int){
    fmt.Printf("Caminhao andando %d, com %dKg de carga\n",km,c.carga)
}

func andar(v Veiculo, km int){
    v.andar(km)
}

func main() {
    carro := &Carro{5}
    caminhao := &Caminhao{2000}
    andar(carro,10)
    andar(caminhao,20)
}
```

# Sequenciadores

- Desvios incondicionais
- Escapes
- Exceções

# Desvios Incondicionais

```
I: Println("Digite a dimensao: (N)");

var tam int = ReadInt();
if ( tam < 20 && tam > 4) {
    goto L;
}

Println("N fora do limite [4,20].");
goto I;

L: geraTabuleiro(tam);
```

- Para fazer desvios incondicionais em go usa-se a palavra reservada **goto.**
- Não se pode fazer desvios incondicionais para pontos externos do programa, somente dentro do mesmo escopo.

# Escape

```
for i := 0; i < len(tabuleiro); i++ {
    for j := 0; j < len(tabuleiro[i]); j++ {

        if (tabuleiro[i][j] == 9) {
            Println("Achou bomba na linha ", i+1);
            break;
        }
    }
}
```

- Go utiliza a palavras reservadas break e continue como formas de escape.
- Escape não rotulado apenas finaliza a iteração corrente.

# Escape

# Rotulado

```
busca:
for i := 0; i < len(tabuleiro); i++ {
    for j := 0; j < len(tabuleiro[i]); j++ {

        if (tabuleiro[i][j] == 9) {
            Println("Achou bomba na linha ", i+1);
            break busca;
        }
    }
}
```

- Para fazer um escape rotulado utiliza-se a palavra break mais o rótulo.

# Tratamento de exceções

- Não utiliza blocos try/catch.
- As funções podem possuir múltiplos retornos,assim, quando necessário, deve-se incluir um retorno do tipo **error**.
- O programador é responsável por tratar o retorno e verificar se a variável de erro foi preenchida.

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

# Tratamento de exceções

```
package main

import (
    "fmt"
    "time"
)

type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s",
        e.When, e.What)
}

func run() error {
    return &MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```

# Persistência de Memória

```
oldfile := "text.txt"
newfile := "text2.txt"
doc, _ := ioutil.ReadFile(oldfile)
ioutil.WriteFile(newfile,doc,0644)
```

- Variáveis persistentes em go são do tipo FILE

# Concorrência

- Goroutines
- Canais
- Select
  - Selection Default

# Goroutines

- Uma *goroutine* é um segmento leve e gerenciado pelo runtime de Go.
- go f(x, y, z) // Inicia uma nova execução goroutine
- f(x, y, z) // A avaliação de f, x, y, e z acontece na goroutine corrente e para a execução de f acontece em uma goroutine nova.
- Goroutines executam no mesmo espaço de endereço, para que o acesso à memória compartilhada seja sincronizada.

# Goroutines

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

# Canais

- Canais são um conduto tipado através do qual você pode enviar e receber valores com o operador de canal, <-.
- ch<- v // v envia para o canal ch.
- v:= <-ch // Recebe do ch, e atribui o valor de v (os dados fluem na direção da seta).
- Como maps e slices, os canais devem ser criados antes de se usar:
- ch:=make(chanint)
- Por padrão, enviam e recebem bloco até o outro lado estar pronto. Isso permite que goroutines sincronizem sem bloqueios explícitos ou variáveis de condição.

# Canais

```
package main

import "fmt"

func sum(a []int, c chan int) {
    sum := 0
    for _, v := range a {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

# Canais

# Bufferizados

- Os canais podem ser *bufferizados*. Fornecendo o tamanho do buffer como o segundo argumento para make para inicializar um canal bufferizado:
- ch:=make(chanint, 100)
- Envia para um bloco de canais bufferizados apenas quando o buffer está cheio. Recebe bloco quando o buffer está vazio.

# Canais

# Bufferizados

```
package main

import "fmt"

func main() {
	c := make(chan int, 2)
	c <- 1
	c <- 2
	fmt.Println(<-c)
	fmt.Println(<-c)
}
```

# Canais

# Range e Close

- Um remetente pode dar **close** em um canal para indicar que os valores não serão mais enviados. Receptores podem testar se um canal foi fechado através da atribuição de um segundo parâmetro para a expressão de recepção: depois
- v, ok := <-ch
- ok é false se não há mais valores a receber e o canal está fechado.
- O laço for i := range c recebe valores do canal repetidamente até que seja fechado.
- **Nota:** Apenas o remetente deve fechar um canal, nunca o receptor. O envio em um canal fechado irá causar um pânico.
- **Outra nota**: Canais não são como arquivos, você geralmente não precisa fechá-los. O encerramento só é necessário quando o receptor precisa saber que não há mais valores chegando, como para terminar um laço range.

# Canais

## Range e Close

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

```
0
1
1
2
3
5
8
13
21
```

# Select

- A instrução select permite uma espera na goroutine sobre as operações de comunicação múltiplas.
- O bloco select aguarda até que um de seus cases possam executar, então ele executa esse case. Ele escolhe um ao acaso se vários estiverem prontos.

# Select

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

```
0
1
1
2
3
5
8
13
21
34
quit
```

# Select

## Selection Default

- O case default em um select é executado se nenhum outro caso está pronto.
- Utilize um case default para tentar um enviar ou receber sem bloqueio:
- Exemplo:

```
select{
        casei := <-c:
        // usei
        default: // recebendo c bloquearia
        }
```

# Select

## Selection Default

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

```
    .
    .
tick.
    .
    .
tick.
    .
    .

tick.
    .
    .
tick.
    .
    .
tick.
BOOM!
```

# Curiosidades

- cgo, Go importando o C todo pra ele
- Playgound Go, uma forma de testar/criar programas simples á complexos direto do site da GO
- No site da GO existe um passo-a-passo que ensina por meio de texto mais o Playground Go todos os conceitos da linguagem (em português inclusive)
- Possui várias IDEs (ou plugins para IDEs) como por exemplo: Eclipse, NetBeans, LiteIDE, Zeus, etc

# Conclusão

- Go é uma linguagem completa porém brilha quando o assunto é concorrência.
- Prioriza eficiência e legibilidade
- Sacrifica principalmente a confiabilidade

# Fontes

- http://golang.org/
- http://www.infoq.com/presentations/Go-Google