PET / ADS – Programa de Educação Tutorial

Instituto Federal de São Paulo –

Campus São Carlos

APOSTILA DE INTRODUÇÃO À PROGRAMAÇÃO GO

São Carlos – SP

2022

SUMÁRIO

INTRODUÇÃO

A linguagem de programação Go surgiu através da Google, em uma necessidade de suprir a demanda da busca por uma linguagem eficiente, performática e de fácil compilação. Foi a primeira linguagem *mainstream* criada após o surgimento dos processadores com múltiplos núcleos, tendo foco em programação concorrente, paralelismo e maximização.

Desenvolvida por Ken Thompson (inventor da linguagem B), Rob Pike (criador da UTF-8) e Robert Griesemer (programador do V8 da Google), o intuito era criar uma linguagem eficiente (boa performance, bom tempo de compilação, uma biblioteca padrão completa, atendimento multiplataforma e *garbage collection*) e fácil de usar (sendo compilada, com tipagem forte e estática, poucas palavras reservadas e popular).

É utilizada para serviços em grande escala envolvendo *web*, redes e *servers*, tendo foco no *backend*. Pode ser aplicada em *APIs*, *CLIs*, *microservices*, *libraries* e *frameworks*, processamento de dados, entre outros, baseando-se em serviços de *cloud* e orquestração de *containers*.

SOFTWARE

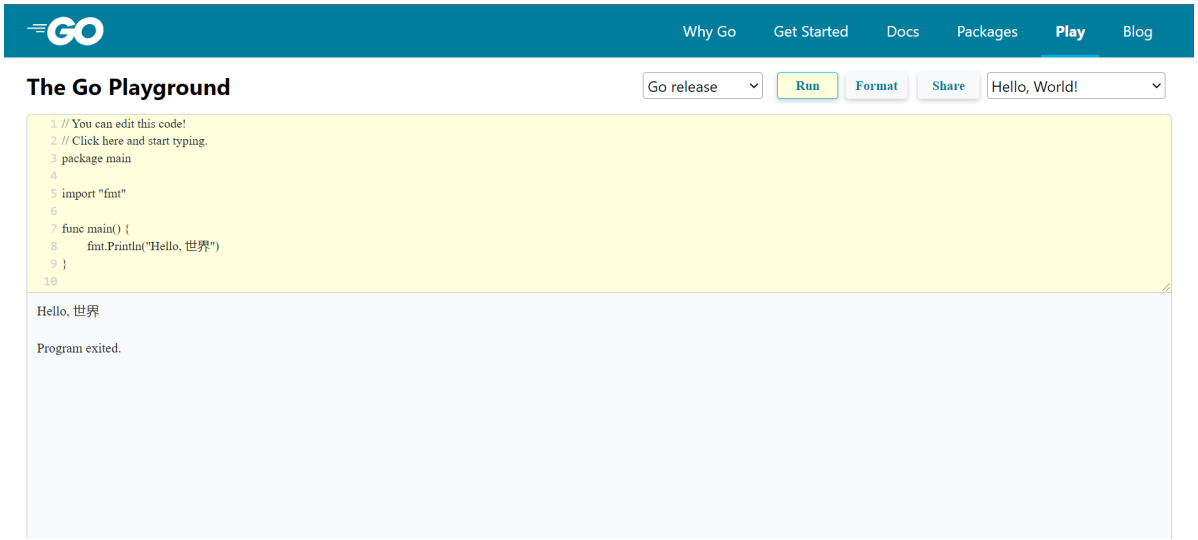
Neste início, não é necessário instalar um *software* específico para programar em Go. Ao invés disso, será utilizado um serviço *web* executado nos servidores golang.org: The Go Playground.

O Go Playground recebe um programa Go, examina, compila, vincula e executa o programa dentro de uma *sandbox* (medida de segurança que limita certas ações com o objetivo de evitar danos ao próprio programa) e retorna a saída.

Como limitações, o *software* é incapaz de utilizar algumas bibliotecas padrão, seu tempo inicia em 2009-11-10 23:00:00 UTC e há restrições quanto ao tempo de execução e usos da *CPU* e da memória.

INTERFACE

Ao abrir o site, a seguinte tela será exibida:



RUN

O botão Run faz com que o programa seja executado, exibindo a saída logo abaixo do código.

FORMAT

O botão Format corrige a indentação e a importação de funções de todo o código.

SHARE

O botão Share gera um *link* para que o código possa ser compartilhado. Entretanto, é necessário gerar um novo *link* para cada modificação.

PROGRAMA BASE

Seja qual for o propósito do seu programa, ao iniciá-lo irá se deparar com uma formatação padrão para todos os códigos em Go.

```
1 package main
2
3 // Pacotes e bibliotecas
4 import "..."
5
6 func main() {
7       // Programa principal
8 }
```

O conjunto de barras (//) é utilizado para comentar o código, não alterando sua funcionalidade.

OPERADORES

Os operadores são responsáveis por informar qual operação será feita em determinado trecho do programa. Existem quatro tipos diferentes de operadores: aritméticos, de comparação, lógicos e de atribuição.

ARITMÉTICOS

Realizam operações matemáticas comuns.

Tabela 1 - Operadores aritméticos

Fonte: Autores

OPERADOR	NOME	FUNÇÃO
+	Adição	Soma os valores
-	Subtração	Subtrai os valores
*	Multiplicação	Multiplica os valores
/	Divisão	Divide os valores
%	Módulo	Retorna o resto da divisão de dois valores

++	Incremento	Aumenta o valor em 1
--	Decremento	Diminui o valor em 1

DE COMPARAÇÃO

Realizam comparação.

Tabela 2 - Operadores de comparação

Fonte: Autores

OPERADOR	NOME
==	Igual
!=	Diferente
>	Maior
<	Menor
>=	Maior ou igual
<=	Menor ou igual

LÓGICOS

Operações lógicas possuem como retorno valores booleanos, sendo utilizados em estruturas condicionais.

Tabela 3 - Operadores lógicos

Fonte: Autores

OPERADOR	NOME	FUNÇÃO
&&	E	Verifica se ambos valores são adequados para a condição
||	Ou	Verifica se algum dos valores é adequado para a

		condição
!	Negação	Retorna o resultado inverso

DE ATRIBUIÇÃO

Atribuem valores para variáveis.

Tabela 4 - Operadores de atribuição

Fonte: Autores

OPERADOR	NOME	FUNÇÃO
=	Atribuição	Atribui um novo valor
+=	Adição e atribuição	Soma o valor existente com o novo
-=	Subtração e atribuição	Subtrai o valor novo do existente
*=	Multiplicação e atribuição	Multiplica o valor existente com o novo
/=	Divisão e atribuição	Divide o valor existente pelo novo
%=	Módulo e atribuição	Retorna o resto da divisão do valor existente pelo novo

VARIÁVEIS

Variáveis são pequenos espaços de memória, utilizados para armazenar e manipular dados. Em Go, os tipos de variáveis mais usadas são int (armazena números inteiros), float32 (armazena números flutuantes), string (armazena texto) e bool (armazena booleanos). Cada variável pode armazenar apenas um tipo de dado.

NOMENCLATURA

Os nomes das variáveis devem ser inicializados com uma letra, mas podem possuir números também. Lembre-se de que a linguagem diferencia maiúsculas e minúsculas.

DECLARAÇÃO

Diferentemente de outras linguagens de programação, é preciso ou não declarar o tipo de cada variável, dependendo de onde ocorre a declaração, pelos diferentes níveis de escopo existentes.

Para variáveis de uso global, conhecidas como variáveis *package-level scope*, é necessária a declaração fora de um *code block*, utilizando a palavra reservada var, informando o tipo e, se desejar, o valor com o operador de atribuição (=).

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  // var nome tipo = valor
7  var inteiro int = 2022
8  var flutuante float32
9  var texto string = "PET - ADS"
10 var booleano bool
11
12 func main() {
13       // Programa principal
14 }
```

Por padrão, ao rodar o programa, será atribuído o valor zero correspondente a cada tipo em variáveis sem atribuição de valor.

Tabela 5 - Valor zero

Fonte: Autores

TIPO	VALOR ZERO
int	0
float64	0.0
string	“”
bool	false
map	nil (nulo)

Para variáveis de uso local, somente declaradas dentro de *code blocks*, pode-se utilizar o operador curto de declaração (:=).

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  func main() {
7          // nome := valor
8          inteiro := 2022
9          flutuante := 20.10
10         texto := "PET - ADS"
11         booleano := true
12 }
```

Este operador identifica o tipo da variável, não sendo necessário indicá-lo, portanto, somente pode ser utilizado na declaração da variável. Para alterar o valor posteriormente, utilize o operador de atribuição (=).

Também é possível declarar simultaneamente diversas variáveis, mesmo que sejam de tipos diferentes.

```go
package main

// Pacotes e bibliotecas
import "..."

// var (
//     nome tipo = valor
//     nome tipo
//)
var (
    inteiro   int = 2022
    flutuante float32
    texto     string = "PET - ADS"
    booleano  bool
)

func main() {
    // nome, nome, ... := valor, valor, ...
    inteiro, flutuante, texto, booleano := 2022, 20.10, "PET - ADS", true
}
```

CONVERSÃO DE TIPOS

Como as variáveis são estáticas, em determinadas ocasiões é necessário convertê-las. Para isso, basta escrever o tipo desejado e o valor entre parênteses, não esquecendo de armazenar caso necessário.

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  func main() {
7          flutuante := 20.22
8
9          // nome := tipo(valor)
10         inteiro := int(flutuante)
11 }
```

CRIANDO SEU PRÓPRIO TIPO

A declaração de um tipo próprio, restrita ao espaço fora de *code blocks*, consiste na palavra reservada `type` seguida do nome desejado e do tipo base.

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  // type nome tipo
7  type PET int
8
9  func main() {
10         // Programa principal
11 }
```

DESCARTE DE VALOR

Alguns métodos retornam diversas informações quando chamados, mas nem sempre todas serão úteis para o

programa. Com isso, utilizamos o sublinhado (_) como nome de variável quando precisamos descartar um dado.

PACOTE FMT

Para utilizar os pacotes, é necessário importá-los em seu programa.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6       // Programa principal
7 }
```

Ao chamar uma função, deve-se utilizar o formato:

nomeDoPacote.nomeDaFunção(parâmetros)

O pacote FMT é responsável pela relação de entrada e saída.

SAÍDA

FORMATAÇÃO DO TEXTO

Ao utilizar um comando de print, é possível formatar sua saída de alguns modos.

Tabela 6 - Formatação do print

Fonte: Autores

COMANDO	FUNÇÃO
\n	Quebra a linha
\t	Insere um tab

MODOS DE PRINT

Tabela 7 - Tipos de print

Fonte: Autores

COMANDO	FUNÇÃO
`fmt.Println(parâmetros)`	Print dos parâmetros com `\n` no final
`fmt.Print(parâmetros)`	Print dos parâmetros sem `\n` no final
`fmt.Sprint(parâmetros)`	Retorna a concatenação dos parâmetros
`fmt.Printf(parâmetros)`	Com `%v` mostra o valor da variável Com `%T` mostra o tipo da variável

Você está pronto para os exercícios de Nível #1!
Pratique antes de continuar

ESTRUTURAS CONDICIONAIS

Estruturas condicionais são utilizadas para determinar se uma condição é verdadeira ou não e então realizar uma ação. Na linguagem Go, utilizamos `if` e `switch` para realizar essas operações.

ESTRUTURA CONDICIONAL SIMPLES

Nessa estrutura, uma condição é avaliada e, caso seja verdadeira, o código contido em seu bloco é executado.

Uma estrutura condicional composta possui mais do que uma condição. Geralmente, quando nossa condição `if` não é verdadeira, é desejado que outro comando seja executado. Nesse cenário, temos o `else`, que será executado caso a condição seja falsa.

```go
package main

import "fmt"

func main() {
        nome := "Ada"

        if nome == "Ada" {
                fmt.Println("O nome é Ada")
        }
}
```

Nesse caso, para ser verdadeiro e executar o código contido, o nome precisa ser Ada. Abaixo temos a saída produzida pelo código acima.

```
O nome é Ada
```

ESTRUTURA CONDICIONAL COMPOSTA

INSTRUÇÕES ELSE IF

A instrução else if (senão se) será utilizada quando estivermos analisando mais de dois resultados possíveis. Deve ser colocada entre um if e um else.

```go
package main

import "fmt"

func main() {
        nome := "Ada"
        if nome == "Carol" {
                fmt.Println("O seu nome é Carol")
        } else if nome == "Frances" {
                fmt.Println("O seu nome é Frances")
        } else {
                fmt.Println("O seu nome não é Carol ou Frances.")
        }
}
```

```
O seu nome não é Carol ou Frances.
```

Nesse exemplo, a variável nome estava armazenando Ada. Como não atendeu as condições de nenhuma estrutura condicional, o bloco de código contido no else foi executado.

INSTRUÇÕES IF ANINHADAS

Podemos utilizar as instruções if aninhadas para verificarmos uma condição secundária que só deve ser executada caso a condição primária seja verdadeira.

```
package main

import "fmt"

func main() {
        nome := "Carol"
        sobrenome := "Shaw"
        if nome == "Carol" {

                if sobrenome == "Shaw" {
                        fmt.Println("O seu nome é Carol Shaw")
                } else {
                        fmt.Println("O seu nome é Carol")
                }
        } else if nome == "Frances" {
                fmt.Println("O seu nome é Frances")
        } else {
                fmt.Println("O seu nome não é Carol ou Frances.")
        }
}
```

```
O seu nome é Carol Shaw
```

No exemplo, verificamos se o nome é Carol: caso a condição seja verdadeira, verificamos o sobrenome, mas se a condição for falsa, o sobrenome não é verificado. Note que a verificação primária nesse caso é a do nome e a secundária do sobrenome.

ESTRUTURAS CONDICIONAIS COM SWITCH

O funcionamento do switch é muito similar ao if, pois ele avalia as condições do bloco e executa apenas se for verdadeiro. Caso nenhuma condição seja verdadeira, o default será executado.

```go
package main

import "fmt"

func main() {
	nome := "Ada"
	switch nome {
	case "Carol":
		fmt.Println("O seu nome é Carol.")
	case "Frances":
		fmt.Println("O seu nome é Frances.")
	default:
		fmt.Println("O seu nome não é Carol ou Frances.")
	}
}
```

```
O seu nome não é Carol ou Frances.
```

Inicializamos o switch seguido da variável nome, isso faz com que todas as condições sejam comparadas a esta variável. Nesse caso, como nenhuma condição foi atendida, o bloco contido no default foi executado.

ESTRUTURA DE REPETIÇÃO

A estrutura de repetição é utilizada para executar uma mesma sequência de comandos várias vezes, normalmente dependem de uma condição ou do número de vezes que precisar ser repetido, essas estruturas são conhecidas também como laços. Na linguagem Go, utilizamos apenas o for.

REPETIÇÃO COM INICIALIZAÇÃO, CONDIÇÃO E EXECUÇÃO

```go
package main

import (
        "fmt"
)

func main() {
        for x :=0; x < 10; x ++{
                fmt.Println(x)
        }
}
```

Primeiro fazemos a inicialização da variável(x := 0), depois temos a condição de parada da repetição (x < 10) e por último realizamos um comando a cada fim de repetição(x++). Note que todos são separados por ponto e vírgula (;).

Em cada repetição estamos dando print no valor de x. Abaixo temos a saída produzida pelo código.

```
0
1
2
3
4
5
6
7
8
9
```

REPETIÇÃO HIERÁRQUICA

Na repetição hierárquica temos *loops* dentro de *loops*, então para a execução de um ciclo completo do *loop* externo é necessário executar os *loops* internos.

No nosso exemplo temos um for que percorre as horas e dentro dele temos um for que percorre os minutos, então para cada valor de hora percorremos os minutos.

```
package main

import (
        "fmt"
)

func main() {

        for horas := 0; horas <= 2; horas++ {
                fmt.Println("Hora:", horas)
                for x := 0; x <= 15; x++ {
                        fmt.Print(" ", x)
                }
                fmt.Println("\n")
        }
}
```

```
Hora: 0
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Hora: 1
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Hora: 2
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

REPETIÇÃO CONDICIONAL

Nesse tipo de repetição não temos os parâmetros de inicialização e execução posterior no for, ou seja, temos apenas a condição, esse tipo de repetição substitui o *while*. A cada novo ciclo de repetição é verificada a condição, e, enquanto ela não retornar *false*, continuará sendo executada.

```go
package main

import (
        "fmt"
)

func main() {
        x := 0

        for x < 5{
                x++
                fmt.Println("X vale : ", x)
        }
}
```

```
X vale :  1
X vale :  2
X vale :  3
X vale :  4
X vale :  5
```

Observe que enquanto o valor de x não era maior que 5, a repetição prosseguiu normalmente, isso porque a condição estava sendo atendida.

REPETIÇÃO INFINITA

Esse tipo de repetição ocorre quando iniciamos um laço de repetição sem que seja possível atender a condição de parada, ou até mesmo sem uma condição. Esse tipo de repetição deve ser usada com break ou continue, para evitar que o código fique preso nesse trecho para sempre.

BREAK

O comando break termina o laço de repetição em que foi declarado.

No exemplo abaixo, o programa mostrará apenas uma vez "Ao infinito e além" e encerrará, porque utilizamos o break.

```go
package main

import (
        "fmt"
)

func main() {
        for {
                fmt.Println("Ao infinito e além.")
                break
        }
}
```

```
Ao infinito e além.
```

CONTINUE

Do mesmo modo que podemos finalizar um laço de repetição, podemos apenas pular um ciclo.

Imagine que você precisa mostrar somente os números pares, isso pode ser feito pulando os ciclos de repetição. Verificamos se o resto da divisão é diferente de zero: caso seja verdadeiro, pulamos o próximo ciclo. No caso, pulamos o print: desse modo, apenas os valores pares serão mostrados.

```go
package main

import (
        "fmt"
)

func main() {
        for i := 0; i < 10; i++ {
                if i % 2 != 0 { // resto da divisão diferente de 0, continua
                        continue
                }
        fmt.Println(i)
        }
}
```

```
0
2
4
6
8
```

Você está pronto para os exercícios de Nível #2!
Pratique antes de continuar

AGRUPAMENTO DE DADOS

ARRAYS

Arrays armazenam diversos valores do mesmo tipo em uma única variável, que podem ser acessados através de um índice que inicia em zero.

Normalmente não são utilizados em Go, por possuírem uma quantidade finita de elementos (a indicada na declaração). Caso seja inicializado sem todos os valores preenchidos, será atribuído valor zero aos índices vazios.

DECLARAÇÃO

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  // Variável global vazia
7  // var nome [quantidade]tipo
8  var arrayGlobalVazio [10]int
9
10 // Variável global com conteúdo
11 // var nome = [quantidade]tipo {valor1, valor2, ...}
12 var arrayGlobalComConteudo= [2]bool{true, false}
13
14 func main() {
15       // Variável local vazia
16       // nome [quantidade]tipo
17       arrayLocalVazio := [10]string{}
18
19       // Variável local com conteúdo
20       // nome = [quantidade]tipo {valor1, valor2, ...}
21       arrayLocalComConteudo := [2]float32{20.10, 20.22}
22 }
```

MÉTODOS

Tabela 8 - Métodos do array

Fonte: Autores

SINTAXE	FUNÇÃO
`nome[índice]`	Retorna o elemento do índice indicado
`nome[índice] = valor`	Altera o elemento do índice

	indicado
len(nome)	Retorna o comprimento do array (quantidade indicada a declaração)

SLICES

Slices são parecidas com arrays, pois armazenam vários valores do mesmo tipo em uma única variável. No entanto, é possível aumentar ou diminuir a quantidade de elementos. Desse modo, além do comprimento também possuem capacidade.

Para adicionar novos elementos em índices não existentes, é necessário utilizar o método append.

DECLARAÇÃO

Uma slice pode ser inicializada como um array sem quantidade.

```go
package main

// Pacotes e bibliotecas
import "..."

// Variável global vazia
// var nome [quantidade]tipo
var sliceGlobalVazia [10]int

// Variável global com conteúdo
// var nome = [quantidade]tipo {valor1, valor2, ...}
var sliceGlobalComConteudo = [2]bool{true, false}

func main() {
	// Variável local vazia
	// nome [quantidade]tipo
	sliceLocalVazia := [10]string{}

	// Variável local com conteúdo
	// nome = [quantidade]tipo {valor1, valor2, ...}
	sliceLocalComConteudo := [2]float32{20.10, 20.22}
}
```

Além disso, existe a declaração com a função make.

```go
package main

// Pacotes e bibliotecas
import "..."

// Variável global
// var nome = make(tipo, quantidade, capacidade)
var sliceGlobal = make([]int, 1, 2)

func main() {
	// Variável local
	// nome = make(tipo, quantidade, capacidade)
	sliceLocal := make([]string, 3, 4)
}
```

MÉTODOS

Tabela 9 - Métodos da slice

Fonte: Autores

SINTAXE	FUNÇÃO
`nome[índice]`	Retorna o elemento do índice indicado
`nome[índice] = valor`	Altera o elemento do índice indicado
`len(nome)`	Retorna o comprimento da **slice**
`cap(nome)`	Retorna a capacidade da **slice**
`nome = append(nome, elemento1, …)`	Adiciona os elementos no final da **slice**

PERCORRENDO

Para percorrer os elementos, utilizamos a estrutura de repetição `for`.

```
6  func main() {
7        slice := []int {}
8
9        // for indice, valor := range nome {...}
10       for indice, valor := range slice {
11             ...
12       }
13 }
```

FATIANDO

Ao fatiar, a slice anterior não pode mais ser utilizada pois sofre alterações.

Caso algum índice não seja informado, a fatia começará no início ou irá até o final.

```
6  func main() {
7        slice := []bool{true, false}
8
9        // nome := nomeSlice[índiceInício:índiceFinal]
10       fatia := slice[:1]
11 }
```

ANEXANDO SLICES

É necessário utilizar as reticências (...) para que os elementos sejam copiados.

```
6  func main() {
7          slice1 := []int{1, 2, 3}
8          slice2 := []int{4, 5, 6}
9
10         // nome := append(nomeSlice1, nomeSlice2...]
11         slice := append(slice1, slice2...)
12 }
```

Para excluir elementos, é necessário unir as fatias sem os valores que deseja retirar.

```
6  func main() {
7          slice := []int{1, 2, 3}
8
9          // nome := append(fatia1, fatia2...]
10         novo := append(slice[:1], slice[2:]...)
11 }
```

SLICE MULTIDIMENSIONAL

É possível criar slices dentro de slices, com quantos níveis necessitar.

```go
package main

// Pacotes e bibliotecas
import "..."

// Variável global vazia
// var nome = [][]...tipo {[]tipo {}, []tipo {}, ...}
var sliceGlobalVazia = [][]string{[]string{}, []string{}}

// Variável global com conteúdo
// var nome = [][]...tipo {[]tipo {valor1, valor2, ...}, []tipo {valor1, valor2, ...}, ...}
var sliceGlobalComConteudo = [][]int{
	[]int{11, 2010},
	[]int{1, 2022},
}

func main() {
	// Variável local vazia
	// nome := [][]...tipo {[]tipo {}, []tipo {}, ...}
	sliceLocalVazia := [][]bool{[]bool{}, []bool{}}

	// Variável local com conteúdo
	// nome := [][]...tipo {[]tipo {valor1, valor2, ...}, []tipo {valor1, valor2, ...}, ...}
	sliceLocalComConteudo := [][]float32{
		[]float32{1.1, 20.10},
		[]float32{1, 20.22},
	}
}
```

MAPS

Maps armazenam diversos elementos formados por pares chave:valor, sendo cada chave única dentro da estrutura e não permanecendo na ordem de inserção.

DECLARAÇÃO

```
1  package main
2
3  // Pacotes e bibliotecas
4  import "..."
5
6  // Variável global vazia
7  // var nome = map[tipoChave]tipoValor
8  var mapGlobalVazio = map[string]int{}
9
10 // Variável global com conteúdo
11 // var nome = map[tipoChave]tipoValor {chave:valor, chave:valor, ...}
12 var mapGlobalComConteudo = map[string]int{
13         "pet":      2010,
14         "apostila": 2022,
15 }
16
17 func main() {
18         // Variável local vazia
19         // nome := map[tipoChave]tipoValor
20         mapLocalVazio := map[int]bool{}
21
22         // Variável local com conteúdo
23         // nome := map[tipoChave]tipoValor {chave:valor, chave:valor, ...}
24         mapLocalComConteudo := map[int]bool{
25                 0: false,
26                 1: true,
27         }
28 }
```

MÉTODOS

Tabela 10 - Métodos do map

Fonte: Autores

SINTAXE	FUNÇÃO
nome[chave]	Retorna o valor da chave indicada
nome[chave] = valor	Altera o valor da chave indicada
valor, ok := nome[chave]	Verifica a existência da chave no map Caso não exista, a variável valor será 0 e a ok será *false*
delete(nome, chave)	Remove o elemento chave:valor

PERCORRENDO

Para percorrer os elementos, utilizamos a estrutura de repetição for.

```
6  func main() {
7      map := map[int]bool{
8          0: false,
9          1: true,
10     }
11
12     // for chave, valor := range nome {...}
13     for chave, valor := range map {
14         ...
15     }
16 }
```

Você está pronto para os exercícios de Nível #3!
Pratique antes de continuar

STRUCT

O struct permite armazenar dados de diferentes tipos, como um formulário que pode ser preenchido com nome, idade e endereço.

```go
package main

import "fmt"

type cliente struct {
        nome      string
        sobrenome string
        seguro    bool
}

func main() {
        cliente1 := cliente{
                nome:      "Ana",
                sobrenome: "Leme",
                seguro:    false,
        }
        cliente2 := cliente{"Iara", "PET", true}

        fmt.Println(cliente1)
        fmt.Println(cliente2)
}
```

O struct é um tipo, então é declarado como os outros. É importante sempre declarar os tipos dos dados. No exemplo, nome e sobrenome são strings e seguro um bool. Observe que existem duas maneiras de atribuir dados a uma variável. Lembre-se que as vírgulas são obrigatórias após cada informação.

STRUCT EMBUTIDO

Como o próprio nome indica, struct embutido trata-se de um struct contido em outro, não existindo um limite de structs que podem ser embutidos.

```go
package main

import "fmt"

type pessoa struct {
        nome  string
        idade int
}

type profissional struct {
        pessoa
        titulo  string
        salario int
}
```

Observe que no struct profissional, está embutido o struct pessoa.

```go
func main() {

        pessoa2 := profissional{
                pessoa: pessoa{
                        nome:  "Maria",
                        idade: 26,
                },
                titulo:  "auxiliar",
                salario: 1500,
        }


        fmt.Println(pessoa2)
}
```

Note que o processo para preencher um struct embutido é semelhante a um struct normal, a diferença é que estamos preenchendo o struct pessoa dentro do struct profissional.

Para identificar se existe um struct embutido, podemos analisar a saída produzida.

```
{{Maria 26} auxiliar 1500}
```

Ainda no exemplo anterior, observe que a saída produzida por cada struct foi separada pelas chaves, então o nome e a idade são um struct, e, a profissão e salário, outro, onde está contido o anterior.

COMO ACESSAR CAMPOS STRUCT

Nem sempre será necessário visualizarmos todos os dados de um struct, portanto, precisamos acessar somente o desejado.

Caso deseje acessar apenas a idade de uma pessoa:

```
16 func main() {
17
18         pessoa1 := pessoa{
19                 nome:  "Joaquim",
20                 idade: 45,
21         }
22
23         fmt.Println(pessoa1.idade)
24
25 }
26
```

```
45
```

Note que tivemos como saída apenas a idade, mesmo tendo o nome preenchido.

Acessando dados de um struct embutido:

```
16 func main() {
17
18         pessoa2 := profissional{
19                 pessoa: pessoa{
20                         nome:  "Maria",
21                         idade: 26,
22                 },
23                 titulo:  "auxiliar",
24                 salario: 1500,
25         }
26
27         fmt.Println(pessoa2.pessoa.nome)
28
29 }
30
```

```
Maria
```

Nesse caso, precisamos acessar o struct pessoa e depois o nome, pois o struct pessoa está embutido.

Também é possível acessar normalmente, como no primeiro exemplo, caso não exista conflito de nome das variáveis.

STRUCT ANÔNIMOS

Esses structs não possuem um tipo declarado, apenas uma estrutura montada que não pode ser reutilizada.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6         x := struct {
7                 nome  string
8                 idade int
9         }{
10                nome:  "Giovane",
11                idade: 64}
12        fmt.Println(x)
13 }
```

Observe que, no exemplo, não temos a declaração type, apenas nomeamos o struct, declaramos as chaves com seus tipos e preenchemos. Esse tipo de struct é descartável, então não deve ser utilizado em todos os casos.

Você está pronto para os exercícios de Nível #4!
Pratique antes de continuar

FUNÇÕES

Como em outras linguagens de programação, a função representa uma parte do código que uma vez definida pode ser reutilizada. A linguagem GO possui uma grande biblioteca de funções predefinidas, como as do pacote FMT.

DEFININDO UMA FUNÇÃO

Para definirmos uma função, utilizamos a palavra func seguida pelo nome escolhido e os parênteses contendo os parâmetros da função, podendo ser vazio.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          valor := multiplica(2, 3)
7
8          fmt.Println(valor)
9
10 }
11
12 func multiplica(x, y int) int {
13         return x * y
14
15 }
```

Observe que no exemplo acima nossa função recebe os argumentos de acordo com os parâmetros solicitados, essa

é uma função com argumento e retorno, então após os parâmetros é necessário definir o tipo do retorno.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          basica()
7  }
8
9  func basica() {
10         fmt.Println("Continue aprendendo com o PET")
11
12 }
```

Nesse caso, nossa função não possui argumentos ou um retorno, chamamos ela de função básica.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          argumento("erro")
7  }
8
9  func argumento(x string) {
10         if x == "erro" {
11                 fmt.Println("Tente novamente")
12         } else if x == "acerto" {
13                 fmt.Println("Parabéns")
14         } else {
15                 fmt.Println("Inválido")
16         }
17
18 }
```

Observe o exemplo: aqui temos uma função que recebe um argumento e exibe uma saída de acordo. Caso tente chamar essa função sem um parâmetro, será exibido um erro.

```go
package main

import "fmt"

func main() {
        total := soma(10, 3, 5, 6, 8)
        fmt.Println(total)
}

func soma(x ...int) int {
        soma := 0
        for _, v := range x {
                soma += v
        }
        return soma
}
```

Note no exemplo que usamos o operador `...` no parâmetro da nossa função `soma`. Isso permite que a função receba quantos elementos o usuário desejar.

FUNÇÕES E SLICES

```go
package main

import "fmt"

func main() {
        si := []int{10, 3, 5, 6, 8}
        total := soma(si...)
        fmt.Println(total)
}

func soma(x ...int) int {
        soma := 0
        for _, v := range x {
                soma += v
        }
        return soma
}
```

A função soma está esperando um int e não um slice, então, para que funcione corretamente, precisamos utilizar o operador ... que irá desenrolar os valores.

DEFER

Trata-se de uma instrução para que uma função seja executada por último, independente da ordem.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6         defer fmt.Println("Esse é o primeiro")
7         fmt.Println("Esse é o segundo")
8 }
```

Temos duas instruções, a primeira começa com a palavra-chave defer,seguida de uma instrução print,a próxima linha possui apenas uma instrução print.

```
Esse é o segundo
Esse é o primeiro

Program exited.
```

Note que a segunda linha foi impressa primeiro, isso acontece porque qualquer instrução acompanhada de defer não é invocada até o final da função na qual tenha sido usada.

MÉTODOS

Um método é uma função anexada a um tipo, permitindo que seja possível comunicar como os dados devem ser

usados. Um método não está disponível para qualquer tipo de dado, deve ser utilizado apenas pelo tipo definido.

A sintaxe para definir um método é semelhante a sintaxe de uma função, a diferença é que adicionamos um parâmetro extra após a palavra-chave func, para especificar o receptor do método. Receptor é o tipo que você deseja para definir seu método.

```
1  package main
2
3  import "fmt"
4
5  type pessoa struct {
6          nome      string
7          sobrenome string
8  }
9
10 func (p pessoa) bomDia() {
11         fmt.Println(p.nome, "diz bom dia")
12 }
13
14 func main() {
15         ana := pessoa{"Ana", "Silva"}
16         ana.bomDia()
17
18 }
```

No exemplo, "bomDia" é o método, então essa função só se aplica ao tipo pessoa, caso eu tente chamar a função sem estar com o tipo correto ocorrerá um erro. O método permite que apenas o tipo adequado seja utilizado.

```
Ana diz bom dia

Program exited.
```

Observe que a saída foi correta.

INTERFACES E POLIMORFISMO

Na linguagem GO os valores podem ter mais de um tipo, uma interface permite que isso aconteça.

No polimorfismo, através de uma mesma ação é possível trabalhar com tipos diferentes.

```go
package main

import "fmt"

type pessoa struct {
        nome      string
        sobrenome string
        idade     int
}
type dentista struct {
        pessoa
        especialidade string
        salario       float64
}
type arquiteto struct {
        pessoa
        tipoDeConstrucao string
        tamanhoDaCasa    string
}

func (x dentista) bomDia() {
        fmt.Println(x.nome, "diz bom dia")
}
func (x arquiteto) bomDia() {
        fmt.Println(x.nome, "diz bom dia")
}
```

```go
type gente interface {
        bomDia()
}

func humano(g gente) {
        g.bomDia()
}

```

```
36 func main() {
37
38         carlos := dentista{
39                 pessoa: pessoa{
40                         nome:      "Carlos",
41                         sobrenome: "Dentadura",
42                         idade:     46},
43                 especialidade: "Arrancar dentes",
44                 salario:       150000,
45         }
46
47         marcelo := arquiteto{
48                 pessoa: pessoa{
49                         nome:      "Marcelo",
50                         sobrenome: "Predio",
51                         idade:     26},
52                 tipoDeConstrucao: "Casas",
53                 tamanhoDaCasa:    "Mansão",
54         }
55         humano(carlos)
56         humano(marcelo)
57
58 }
```

Observe que temos a interface "humano", mesmo com valores de tipos diferentes podemos chamar a mesma coisa dentro de cada valor, então, mesmo que o tipo arquiteto e o tipo dentista possuam diferentes tipos de valores, consigo utilizar a interface para chamá-las e utilizar apenas o desejado. Não é necessário utilizarmos um método específico, como naquele exemplo anterior do bom dia.

FUNÇÃO ANÔNIMA

A função anônima pode ser declarada e chamada ao mesmo tempo, ela não precisa de um nome, porém trata-se de uma função que pode ser usada apenas uma vez.

```go
package main

import "fmt"

func main() {
        x := 5
        func(x int) {
                fmt.Println(x, "vezes 5 é: ")
                fmt.Println(x * 5)
        }(x)
}
```

```
5 vezes 5 é:
25

Program exited.
```

FUNÇÃO COMO EXPRESSÃO

Uma variável recebe como valor uma função.

```go
package main

import "fmt"

func main() {
        x := 5
        y := func(x int) {
                fmt.Println(x, "vezes 5 é: ")
                fmt.Println(x * 5)
        }
        y(x)
}
```

Observe que a variável “y” possui como valor uma função anônima.

RETORNANDO FUNÇÃO

Trata-se de uma função que retorna a outra função.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := retornaFuncao()
7          y := x(3)
8          fmt.Println(y)
9  }
10
11 func retornaFuncao() func(int) int {
12
13         return func(i int) int {
14                 return i * 5
15         }
16
17 }
```

No exemplo, o retorno da função se tornou o valor da variável "x", passando o argumento 3 para a função de retorno (linha 13), retornando o resultado.

CALLBACK

Nesse caso, o argumento de uma função é outra função, utiliza-se para que uma função seja executada logo em seguida.

```go
package main
import "fmt"

func main() {
	t := pares(soma, []int{50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60}...)
	fmt.Println(t)
}

func soma(x ...int) int {
	n := 0
	for _, v := range x {
		n += v
	}
	return n
}
func pares(f func(x ...int) int, y ...int) int {
	var slice []int
	for _, v := range y {
		if v%2 == 0 {
			slice = append(slice, v)
		}
	}
	total := f(slice...)
	return total
}
```

Observe no exemplo que ao chamar a função "pares" estou passando como argumento a função "soma" e um slice,então, a função de soma será executada logo em seguida da função "pares".

RECURSIVIDADE

A recursividade consiste em uma função que pode chamar a si mesma, geralmente até atender alguma condição determinada.Trata-se de um recurso muito utilizado na resolução de exercícios como a sequência de Fibonacci e Fatoriais.

```go
package main

import (
        "fmt"
)

func main() {
        fmt.Println(fatorial(3))
}

func fatorial(x int) int {
        if x == 1 {
                return x
        }
        return x * fatorial(x-1)
}
```

No exemplo acima, temos uma função que executa o cálculo fatorial, observe que a função "fatorial" possui uma verificação, caso ela não seja atendida,chama-se novamente a função fatorial, dessa vez com o valor de x-1, desse modo temos a recursividade, a função será executada até o valor de x ser igual a 1.

Você está pronto para os exercícios de Nível #5!
Pratique antes de continuar

PONTEIROS

Somente são utilizados quando o valor origem precisa ser alterado, pois contém o endereço de memória desse valor, evitando a criação de diversas cópias que tornam o programa menos eficiente.

MÉTODOS

OPERADOR	FUNÇÃO

&variável	Referência para o endereço
*endereço	Referência para o conteúdo

Você está pronto para os exercícios de Nível #6!
Pratique antes de continuar

EXERCÍCIOS

NÍVEL #1

1. Utilizando o operador curto de declaração, atribua estes valores às variáveis x, y e z, respectivamente:
   - 2022
   - PET
   - true

   Agora, demonstre os valores nas variáveis com:

   - uma única declaração print
   - múltiplas declarações print

   Link: https://go.dev/play/p/Xhe3XIuRhc4

2. Use var para declarar três variáveis com *package-level scope* e não atribua valores. Utilize os seguintes nomes e tipos:
   - x com tipo int
   - y com tipo string
   - z com tipo bool

   Na função main:

   - Demonstre os valores de cada identificador

   Qual o nome dos valores que o compilador atribuiu para essas variáveis?

   Link: https://go.dev/play/p/29fcmGgQORy

3. Utilizando a solução do exercício anterior:

   Em *package-level scope*, atribua os seguintes valores às variáveis:

   - 2022 para x
   - ADS para y
   - true para z

   Na função main:

   - Use fmt.Sprint para atribuir todos esses valores a uma única variável de nome s
   - Demonstre a variável s

   Link: **https://go.dev/play/p/inGbLV_02rg**

4. Crie um tipo próprio com int e - utilizando a declaração por extenso - uma variável x com este tipo.

   Na função main:

   - Demonstre o valor da variável x
   - Demonstre o tipo da variável x
   - Atribua 2022 à variável x utilizando o operador de atribuição
   - Demonstre o valor da variável x

   Link: **https://go.dev/play/p/fNKnl-0jXEv**

5. Utilizando a solução do exercício anterior:

   Em *package-level scope*, crie uma variável y com o tipo subjacente do tipo criado.

   Na função main:

   - Utilize conversão para atribuir o valor de x para y com o operador de atribuição
   - Demonstre o valor e o tipo de y

Link: https://go.dev/play/p/yh7tiAoi9cH

NÍVEL #2

1. Utilizando o laço de repetição for, exiba os números de 1 a 100.

   Link: https://go.dev/play/p/YU38EwYam1c

2. Utilizando a repetição condicional, crie um *loop* que conte os anos de 2016 até 2022.

   Link: https://go.dev/play/p/ASrxypYeHPw

3. Agora, utilizando a repetição infinita e estruturas condicionais, crie um *loop* que conte os anos de 2002 até 2022.

   Link: https://go.dev/play/p/AdoTQePa-j_a

4. Demonstre o resto da divisão por 3, de todos números entre 9 e 100.

   Link: https://go.dev/play/p/LQNQoYmVhIb

NÍVEL #3

1. Crie um array com 5 valores do tipo int e utilize range para demonstrar estes valores. Depois, demonstre o tipo do array.

   Link: https://go.dev/play/p/A6oa6Z25vvu

2. Crie uma slice do tipo int com 10 valores e utilize range para demonstrar estes valores. Depois, demonstre o tipo da slice.

   Link: https://go.dev/play/p/igZvmJ3i79H

3. Utilize o exercício anterior para demonstrar:
   - do primeiro ao terceiro elementos

- do quinto ao último elementos
- do terceiro ao penúltimo elementos

Link: https://go.dev/play/p/KqKVy4nX8zH

4. Utilizando a slice pet := []int{2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019}:
   - Adicione o valor 2020
   - Adicione os valores 2021, 2022 e 2023 com uma única declaração
   - Demonstre a slice
   - Adicione a seguinte slice: ads := []int{2024, 2025, 2026, 2027, 2028}
   - Demonstre após o item anterior

   Link: https://go.dev/play/p/4jGYqGrC0Gj

5. Utilizando a slice pet := []int{2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019}:
   - Crie uma slice de nome ads com os valores 2010, 2011, 2012, 2016, 2017, 2018 e 2019
   - Demonstre a slice ads

   Link: https://go.dev/play/p/yKY4hh07MH1

6. Crie uma slice contendo slices de strings ([][]string). Atribua valores a esta slice multidimensional da seguinte maneira:
   - Nome, sobrenome, hobby favorito

   Inclua dados para 3 pessoas e utilize range para demonstrar.

   Link: https://go.dev/play/p/Ww11REuFmZR

7. Crie um map com chave do tipo string e valor do tipo []string. A chave deve conter nomes no formato sobrenome_nome e o valor os hobbies favoritos da pessoa. Demonstre o map.

   Link: https://go.dev/play/p/AWsTFoYLxbk

8. Utilizando o exercício anterior: adicione uma entrada ao map e demonstre-o.

   Link: https://go.dev/play/p/_YRKurFnEOJ

9. Utilizando o exercício anterior: remova uma entrada do map e demonstre-o.

   Link: https://go.dev/play/p/uDoEivkLVKP

NÍVEL #4

1. Crie um struct tipo pessoa que possa conter os seguintes dados: nome, idade e comidas favoritas.Então, crie dois valores do tipo "pessoa", em seguida utilize range na slice que contém as comidas favoritas.

   Link: https://go.dev/play/p/bsKyEafpiJM

2. Crie um struct tipo veículo que possa receber a quantidade de portas e a cor,esse struct deve estar embutido nos próximos. Em seguida, crie o tipo caminhonete que deve conter um tipo bool chamado "tracaoQuatro", o próximo tipo é o sedan, deve conter um tipo bool chamado "modeloLuxo". Agora, crie valores para o tipo sedan e o tipo caminhonete, demonstre esses valores e, um único campo de cada um dos dois.

   Link: https://go.dev/play/p/HoMv-faCoFx

3. Crie um struct anônimo, dentro do struct tenha um valor do tipo map e um do tipo slice.

   Link: https://go.dev/play/p/m8DNYHL_BcD

NÍVEL #5

1. Criei uma função que retorne um inteiro e outra que retorne uma string e um inteiro.

Link: https://go.dev/play/p/m923NkGRXXX

2. Utilizando o recurso de callback, crie um programa que some apenas os números ímpares do conjunto.

   (25,27,28,39,64,22,20,79)

   Link: https://go.dev/play/p/HvJ4OJvXX5O

3. **Utilizando a declaração defer, crie um contexto que demonstre que sua execução só acontece ao final.**

   Link: https://go.dev/play/p/w7p-NetER8x

4. **Criei um struct pessoa que contenha pelo menos 3 campos, em seguida crie uma função que exibe todos esses campos. Lembre-se de atribuir valores ao struct para demonstrar.**

   Link: https://go.dev/play/p/OuNxh7P6EdU

5.

NÍVEL #6

1. Crie uma variável e demonstre o endereço na memória

   Link: https://go.dev/play/p/jOX-ot4dy4j

REFERÊNCIAS BIBLIOGRÁFICAS

KÖRBES, Ellen. **Aprenda Go.** Disponível em: <https://www.youtube.com/playlist?list=PLCKpcjBB_VlBsxJ9IseNxFllf-UFEXOdg>. Acesso em: fevereiro de 2022.

W3SCHOOLS. **Go.** Disponível em: <https://www.w3schools.com/go>. Acesso em: janeiro de 2022.

GO. **A Tour of Go**. Disponível em: <https://go.dev/tour/list>. Acesso em: fevereiro de 2022.