Introdução ao Go

Aula 1: Introdução ao Módulo

Agora que começamos este curso,

e que já escrevemos nosso primeiro programa básico em Go,

é hora de aprofundar um pouco mais

e explorar os conceitos essenciais do Go,

os conceitos principais que você sempre vai precisar, independentemente do tipo de aplicação Go que for desenvolver.

Nesta seção, vamos dar uma olhada nos componentes principais.

Você pode pensar nesses componentes como as partes fundamentais presentes em qualquer programa Go.

Depois, exploraremos o conceito de **valores e tipos**,

um conceito crucial, como você verá.

Ao longo do caminho, também aprenderemos sobre **variáveis** e **constantes**.

Além disso, veremos as **funções**:

como criar funções,

como executá-las,

e qual é a ideia por trás delas.

Por fim, vamos dar uma primeira olhada nas **estruturas de controle**,

que são importantes, como você verá,

para controlar como o código será executado

e determinar qual parte do código será executada em determinado momento.

Temos muitos conceitos para abordar, então vamos começar!

Aula 2: O que é Go? E por que ele é incrível?

Então, o que é Go e por que você deveria usá-lo?

Go é uma linguagem de programação, mais especificamente, uma linguagem **open-source** (código aberto) desenvolvida e publicada pelo Google.

É uma linguagem relativamente nova. Ela foi criada em 2007 e lançada publicamente em 2009.

Mas o que torna Go tão especial e popular?

- **Simplicidade, clareza e escalabilidade**: Go é projetado para ser simples de entender e usar, mas ao mesmo tempo, é poderoso o suficiente para lidar com projetos grandes e escaláveis.
- **Desempenho excelente**: Go foi desenvolvido com foco em alta performance, sendo uma ótima escolha para aplicações que exigem eficiência.
- **Concorrência**: Go é incrível para trabalhar com **concorrência**, ou seja, para executar várias tarefas simultaneamente de forma eficiente.
- **"Baterias incluídas"**: A linguagem vem com muitos recursos essenciais já integrados. Isso significa que você não precisa de uma tonelada de bibliotecas externas para realizar tarefas básicas.
- **Tipagem estática**: Go é uma linguagem com **tipagem estática**, o que ajuda você a detectar e corrigir muitos erros logo no início do desenvolvimento, antes mesmo de testar ou executar sua aplicação. Isso reduz os problemas que aparecem apenas durante a execução.

Por conta dessas vantagens, Go se tornou uma linguagem muito utilizada em áreas como:

- **Aplicações de rede** e **APIs**.
- **Aplicações baseadas em microserviços**.
- Ferramentas de linha de comando (**CLI - Command-Line Interface**).

Ao longo deste curso, vamos explorar essas características e benefícios na prática, desenvolvendo vários projetos e exemplos que se encaixam nessas áreas.

Go é uma linguagem poderosa e eficiente, e você vai perceber isso enquanto avançamos!

Aula 3: Organizando o Código com Pacotes

Um dos elementos essenciais em qualquer programa Go é o **"package"** que colocamos no início do arquivo.

Se você olhar aqui no topo do código, temos essa declaração de pacote (também chamada de *cláusula de pacote*).

E por que isso é importante? Porque **todo arquivo de código Go precisa ter essa instrução de pacote**.

Se você tentar removê-la e estiver usando a extensão Go no VS Code, verá que o editor exibirá um erro: ele espera encontrar uma declaração de pacote, mas encontra, por exemplo, um comando `import` como o primeiro elemento do arquivo.

Portanto, **a declaração de pacote é obrigatória**. Mas qual é a ideia por trás dela?

Pacotes: Para que servem?

Em Go, usamos pacotes para **organizar nosso código**.

- **Todo programa Go precisa ter pelo menos um pacote.**
- Mas você pode criar vários pacotes no mesmo projeto.
- Além disso, um único pacote pode ser dividido entre vários arquivos.

Por exemplo:
Eu posso adicionar outro arquivo Go ao projeto, e ele ainda pode pertencer ao mesmo pacote `main`. Isso é possível e bastante comum.

Pacotes ajudam a manter o código mais **modular e organizado**. Isso é especialmente útil quando trabalhamos com múltiplos arquivos ou projetos maiores, onde cada pacote pode conter funcionalidades específicas.

Usando recursos de outros pacotes

Uma das grandes vantagens dos pacotes é que podemos **exportar e importar funcionalidades** de um pacote para outro.

- Por exemplo, um **pacote A** pode usar algo do **pacote B**.
- Isso permite que cada arquivo ou pacote foque em funcionalidades específicas, mantendo o código mais limpo e enxuto.

O que estamos vendo no exemplo?

Aqui no código, temos uma **declaração de importação** com o comando `import`. Estamos importando o pacote **FMT**.

- Esse **FMT** não foi escrito por nós.
- Ele faz parte da **biblioteca padrão do Go** (que mencionei na aula anterior).

A biblioteca padrão do Go vem com vários pacotes pré-instalados, prontos para serem usados em qualquer programa.

O pacote **FMT**, por exemplo, oferece funções para exibir texto na tela, como a função `Print`.

Explorando a biblioteca padrão

Se você quiser saber mais sobre esses pacotes que vêm com o Go, há uma documentação oficial da biblioteca padrão.

- Lá, você pode buscar pelo pacote **FMT**, por exemplo, e aprender mais sobre suas funcionalidades.
- Mas não se preocupe, no curso vamos usar isso na prática.

Por que escolhemos o nome `main` para o pacote?

O nome `main` é especial no Go.

- Todo programa Go precisa de um **pacote principal**, chamado de `main`.
- É nele que o programa começa a ser executado.

Quando você cria projetos maiores, pode dividir o código em vários pacotes, mas o ponto de entrada do programa sempre estará no pacote `main`.

E é por isso que começamos o curso usando o pacote `main`. É o ponto de partida do nosso código!

Pacotes são fundamentais no Go, tanto para organização quanto para reutilização de código. Vamos explorar isso ainda mais à medida que avançarmos no curso!

Aula 4: A Importância do Nome "main"

Por que escolhemos o nome do pacote como **main**? Por que não algo como "app", "hello" ou qualquer outro nome?

Na verdade, **você pode usar qualquer nome para um pacote** no Go, especialmente em projetos maiores que utilizam múltiplos pacotes.
Por exemplo, o pacote `fmt`, que já usamos, tem o nome "fmt" e não "main".

Mas o pacote `main` tem algo **especial**: ele indica para o Go que **esse é o ponto de entrada principal** da aplicação que estamos criando.

Por que o pacote `main` é tão importante?

O nome `main` informa ao Go que o código dentro deste pacote será **executado primeiro**.

Durante o desenvolvimento, usamos comandos como:

```bash
go run nome_do_arquivo.go
```

Esse comando é útil porque permite rodar o código diretamente, sem precisar de uma compilação formal. Mas no mundo real, principalmente em produção, as coisas funcionam de maneira diferente.

Executando em produção

Quando você quiser disponibilizar sua aplicação para outras pessoas, **elas não terão o Go instalado no sistema**.
Então, como elas executarão sua aplicação?

É aqui que entra o comando `go build`.

O que o comando `go build` faz?

- Quando você executa o comando `go build` dentro do diretório do seu projeto, ele cria um **arquivo executável**.
- Esse executável pode ser rodado em sistemas que **não têm o Go instalado**.

Por exemplo:

```bash
go build
```

Depois de rodar esse comando, será gerado um arquivo executável (no Windows será algo como `projeto.exe`, e no Linux ou Mac, será um arquivo sem extensão).

Agora, você pode distribuir esse arquivo, e qualquer pessoa poderá executá-lo diretamente no sistema operacional.

Erros ao usar `go build`

Se você tentar rodar `go build` em um projeto sem o pacote `main`, verá o seguinte erro:

```bash
cannot find main module
```

Isso acontece porque o Go **precisa do pacote `main` para criar um executável**. Sem ele, o Go não sabe qual código executar primeiro.

Estrutura básica de um programa Go com o pacote `main`

Aqui está um exemplo de como deve ser a estrutura de um programa Go básico com o pacote `main`:

```go
package main
```

```
import "fmt"

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

1. `package main`: Define que este é o pacote principal.
2. `func main()`: Essa função também é especial no Go. Ela é o **ponto de entrada do programa**. Quando você executa o programa, a função `main` é sempre chamada primeiro.

Resumo

- O nome do pacote `main` é obrigatório para criar um executável em Go.
- A função `main()` dentro desse pacote é onde a execução do programa começa.
- O comando `go build` cria um arquivo executável que pode ser rodado sem precisar do Go instalado no sistema.

Ao longo do curso, vamos ver como construir aplicações mais complexas que aproveitam essas características.

Aula 5: Compreendendo Módulos Go e Construindo Programas Go

Para entender por que o pacote principal deve ser chamado de `main`, precisamos resolver um erro comum: a falta de um **módulo principal**.

Você pode pensar: *"Já temos um pacote chamado `main`, isso não conta como módulo?"*.
A resposta é **não**. Pacotes e módulos são **coisas diferentes no Go**, e essa diferença é importante.

O que é um módulo no Go?

- Um **módulo** é um projeto Go que pode conter vários **pacotes**.

- Um projeto simples, com apenas um arquivo Go, pode ser considerado um módulo.
- Com o tempo, conforme adicionamos mais pacotes, um módulo se torna mais complexo.

Para que o Go reconheça seu projeto como um módulo, precisamos **inicializá-lo como tal**.

Inicializando um módulo com `go mod init`

Para criar um módulo, usamos o comando:

```bash
go mod init nome_do_modulo
```

O `nome_do_modulo` pode ser algo como:

- Uma URL do GitHub (ex.: `github.com/seuusuario/seuprojeto`), caso você planeje hospedar o projeto online.
- Ou um nome fictício (ex.: `example.com/meu-primeiro-app`), apenas para inicialização local.

Exemplo prático: criando um módulo

1. **Erro ao tentar criar um módulo**
   Se você não informar um nome ao rodar `go mod init`, verá o erro:

   ```text
   cannot determine module path
   ```

   A solução é adicionar o nome ou caminho do módulo, por exemplo:

   ```bash
   go mod init example.com/primeiro-app
   ```

Após rodar esse comando, o Go cria o arquivo `go.mod`.

2. **O que é o `go.mod`?**
   O arquivo `go.mod` indica que:
   - A pasta onde ele está e todas as suas subpastas pertencem a esse módulo.
   - O módulo tem o nome ou caminho definido na inicialização (no caso, `example.com/primeiro-app`).

   Aqui está um exemplo de como o arquivo `go.mod` pode parecer:

```text
module example.com/primeiro-app

go 1.20
```

Construindo seu programa com `go build`

Após inicializar o módulo, podemos usar o comando `go build` para compilar o programa.

O que acontece ao rodar `go build`:

- O Go cria um **arquivo executável** no diretório do projeto.
- O nome do executável será baseado no nome do módulo ou no arquivo Go principal.

Exemplo prático: construindo e executando

1. **Rodando o comando de build**

```bash
go build
```

   Se você estiver no Windows, verá um arquivo `primeiro-app.exe`.
   No Linux ou macOS, o arquivo não terá extensão, mas pode ser executado diretamente pelo terminal.

2. **Executando o programa**
   - **No Windows**: Dê um duplo clique no arquivo `.exe`.
   - **No Linux/macOS**: Use o terminal e digite:

     ```bash
     ./primeiro-app
     ```

   Isso executará o programa sem precisar do Go instalado.

O papel do pacote `main`

Se você mudar o nome do pacote `main` para algo como `app`, verá que o comando `go build` não produzirá um executável.
Isso acontece porque o Go **não sabe onde iniciar a execução do programa**, já que o pacote `main` é essencial para definir o ponto de entrada.

Voltando o nome para `main`:

- Se você reverter o pacote para `main` e rodar `go build`, o executável será criado novamente.

Código de exemplo

Aqui está um exemplo completo de um projeto Go com um módulo configurado e o pacote `main`:

Arquivo: `main.go`

```go
package main

import "fmt"

func main() {
```

```
    fmt.Println("Olá, bem-vindo ao meu primeiro programa Go!")
}
```

Passos para criar e compilar:

1. Inicialize o módulo:

   bash

   ```
   go mod init example.com/primeiro-app
   ```

2. Compile o programa:

   bash

   ```
   go build
   ```

3. Execute o programa (no Linux/macOS):

   bash

   ```
   ./primeiro-app
   ```

Resumo

- **Módulo**: Um projeto Go que pode conter múltiplos pacotes.
- `go mod init`: Comando usado para criar o arquivo `go.mod` e inicializar o módulo.
- **Pacote** `main`: Define o ponto de entrada do programa e é obrigatório para gerar um executável.
- `go build`: Cria um executável que pode ser executado em sistemas sem o Go instalado.

Esses conceitos são fundamentais para criar e distribuir aplicações Go. Ao longo do curso, vamos nos aprofundar ainda mais neles!