Desenvolvimento Mobile com

Google Flutter

{ PROFESSOR CHOREN }

Prof. Esp. Anderson Coimbra Choren

Sobre o Autor

Olá, me chamo Anderson Coimbra Choren e gostaria de compartilhar com você, caro leitor, um pouco da minha vida profissional, para que possamos nos conhecer melhor.

Sou um profissional com mais de 14 anos de atuação na área de Tecnologia da Informação. Sendo graduado em Análise e Desenvolvimento de Sistema e pós-graduado em Especialização em Formação de Professores na Docência do Ensino Profissional e Superior, ambos pelas Escolas e Faculdades QI do Brasil, atualmente atuando como professor do curso técnico em informática na mesma instituição, lecionando em disciplinas relacionadas a programação e análise de sistemas. Atuante também nas áreas de UX e UI, áreas essas onde possui formações livres.

FACEBOOK.COM/GROUPS/431311894533110
@PROFESSORCHOREN
@PROFESSORCHOREN
PROFESSORCHOREN@GMAIL.COM
ANDERSONCHOREN.COM.BR

Sumário

Capítulo 1 - Introdução à linguagem Dart

Figura 1.1 Logo da Linguagem Dart

Se você, assim como eu, é um “velho de guarda” nesse universo do desenvolvimento, seja ele mobile, web ou desktop, sabe muito bem que o primeiro passo rumo à excelência na linguagem a qual estamos aprendendo, é conhecer os princípios básicos da linguagem, como a tipagem de variáveis, tipos de dados, paradigmas suportados e etc. E com o Flutter não seria diferente.

Neste capítulo falaremos sobre a linguagem Dart, linguagem utilizada no desenvolvimento de aplicações Flutter.

Mas antes de iniciarmos, permita me explicar uma coisa:

"Flutter é um SDK para desenvolvimento de aplicações nativas para dispositivos com sistema operacional Android ou iOS. Já o

Dart é a linguagem de programação usada para desenvolvermos essas aplicações".

Talvez não tenha ficado clara, então deixe-me resumir. Quando desenvolvemos aplicações para Smartphones Android utilizamos a linguagem Java ou a linguagem Kotlin. Correto? O mesmo se aplica ao iOS onde podemos usar Object-C ou Swift. O Flutter é um SDK que utiliza a linguagem Dart para compilar aplicações nativas para dispositivos Android e iOS, utilizando um mesmo código.

Você deve estar se perguntando: "Onde eu ouvi algo parecido 🤔". Sim, você já ouviu falar desse tipo de abordagem, pois o React Native faz basicamente a mesma coisa. Porém ele usa a linguagem JavaScript, sendo que o Flutter utiliza o Dart.

Para resumir, o Flutter tem a mesma finalidade que o React Native, porém fazendo melhorias onde ele peca 😄.

Mas voltemos ao Dart. Dart é uma linguagem de programação orientada a objetos criada pelo Google, sendo lançada oficialmente no ano de 2011. Não fazendo muito sucesso até ser escolhida como linguagem padrão para o desenvolvimento de aplicações Flutter, o qual também foi criado pela Google e oficialmente lançado em 2017.

Um fator importante sobre Dart é que “ele” (Eu digo ele por parecer um nome masculino, mas seria o nome da linguagem, então seria “a Dart”, mas isso soaria estranho, ao menos na minha visão. Então usarei “o Dart” 😅) possui uma tipagem estática e forte, assim como o Java, C#, TypeScript e outras linguagens parecidas.

OK, mas o que tudo isso quer dizer afinal? 🤔 Beleza, como diria Jack Estripador “Vamos por partes” 😅

Orientação a Objetos: Acho que esse dispensa apresentações. Mas, em todo caso, vamos às explicações. Orientação a Objetos é um paradigma de programação que parte do princípio que todo nosso programa é dividido em partes chamadas de objetos, esses objetos são as entidades que representam o sistema. Bom, como o foco desta publicação não orientação a objetos, não vou me aprofundar muito nesse conceito. Mas se quiser, você pode consultar um artigo da Wikipédia para se inteirar do assunto. Pois O.O é primordial para seus estudos de Dart e Flutter.

Tipagem estática: Basicamente, isso quer dizer que no momento que uma variável é declarada com um tipo de dado, esse tipo não poderá ser modificado. Ou seja, se eu criar uma variável com o tipo de dado inteiro, ou mesmo, não declarar seu tipo e atribuir

um valor inteiro a ela, ela será declarada como inteira, e não poderá receber valores de outros tipos.

Existe uma forma de declarar uma variável com tipagem dinâmica, mas isso é assunto para outra hora.

Tipagem forte: Basicamente, estamos falando da forma como a linguagem tratará operações entre tipos de dados diferentes. Por exemplo: Se eu tentar somar um valor inteiro 5 a uma variável que contém um texto contendo o número 5, a linguagem irá informar que não é possível somar um inteiro com uma string, sendo necessário uma conversão de tipo de dado (Cast) para tal operação.

Bom, agora que já conhecemos um pouco mais sobre “o Dart”, vamos entender um pouco mais sobre a sintaxe e outros aspectos importantes sobre essa linguagem de programação.

1.1 Instalação do Flutter/Dart

Por se tratar de um SDK, o flutter não é instalado em seu computador, mas sim colocado à disposição para uso por outros programas instalados. Dessa forma, não precisaremos nos preocupar em instalá-lo em nossas máquinas, pois basta baixar um arquivo .zip do próprio site do Flutter, o flutter.dev e descompactá-lo no local que desejar.

Para facilitar a sua instalação, o Flutter disponibiliza um tutorial completo de como "instalá-lo" em nossa máquina. Por esse motivo não ficarei aqui “chovendo no molhado”, e vamos aprender através do próprio tutorial do pessoal do Flutter.

Eu recomendo que você instale, se já não estiver utilizando, o Visual Studio Code como editor para seus códigos Flutter.

Mas você deve estar se perguntando “Mas e o Dart, onde eu baixo? 🤔”. Essa é a melhor parte, o Flutter já vem com o Dart empacotado, sendo assim, basta adicionar o binário do Flutter ao seu path de sistema e já poderá rodar comandos Dart direto no seu terminal nativo ou mesmo no terminal do VS 😉.

Se você não sabe adicionar uma variável de ambiente ao Windows, aqui tem um tutorial bem legal sobre isso. Mas se você é um usuário de Mac ou Linux, o próprio tutorial de instalação mostrará como fazer isso.

Há, antes que eu me esqueça. O valor da variável de ambiente é o **caminho da pasta bin** do Flutter, então o caminho varia de acordo com o local onde você colocar o Flutter. Vamos supor que você descompactou no **c:**, então ficará assim:

C:\flutter\bin

Se desejar, também poderá utilizar o Dart Pad para seus estudos. Ele é uma IDE online para Dart e Flutter. Lá você poderá rodar seus códigos sem a necessidade de instalar o Flutter. Mas recomendo que faça uso desta ferramenta **somente até o capítulo 7**. Pois após isso precisaremos do Dart e do Flutter instalado para que possamos dar andamento em nossos estudos.

1.2 Uso de comentários no código

Bom, nesse quesito, o dart é muito parecido com praticamente todas as linguagens de programação existentes hoje em dia. Mas, em todo caso, vamos às formas de declarar comentários em nossos códigos.

👨‍💻 Mão no código

```dart
// Comentário de única linha

/*
 * Comentário
 * de
 * múltiplas
 * linhas
 */

/// Comentário de documentação
```

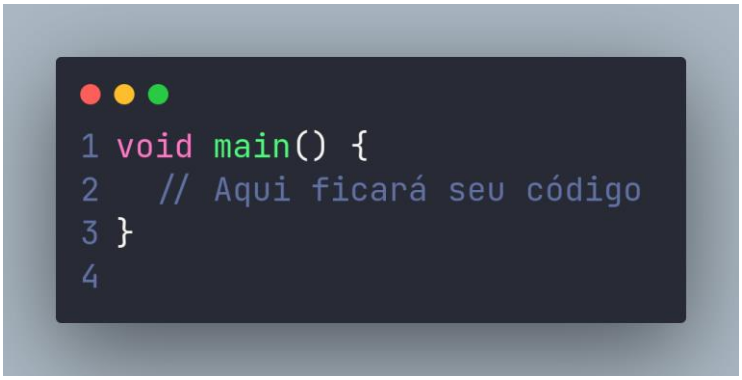
Como podemos perceber no código apresentado acima, podemos fazer uso de dois tipos de comentários, usando **//** para comentários de única linha e **/**/** para comentários de múltiplas linhas.

1.3 O método main

Se você é um desenvolvedor mais experiente, já deve saber do que se trata essa função e poderá pular este tópico. Mas caso esse seja seu primeiro contato com programação, deixe-me explicar o que é essa função e qual sua importância.

O método **main** é o principal método do Dart, sendo o método que a aplicação irá chamar no momento de sua inicialização. É nele que declaramos nossos códigos iniciais, e as chamadas de outros métodos de nosso sistema.

👨‍💻 Mão no código

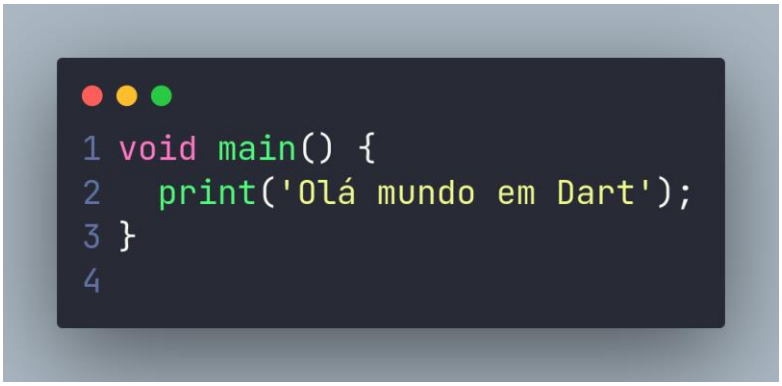


1.4 Nosso 'Hello World' de cada dia

Bem, como dizem os gurus da programação: "Se seu primeiro contato com a linguagem não foi um **Olá mundo**, você está aprendendo errado 😅". Então vamos aprender agora a dar nosso "Olá mundo em Dart". O que não é nada difícil, basta chamarmos a função main, dentro

dela utilizarmos o comando print, passando como parâmetro o valor que desejamos imprimir no console.

👨‍💻 Mão no código



🖨️ Resultado:

```
Olá mundo em Dart
```

1.5 Declaração de variável

A declaração de um variável, talvez seja o segundo passo em qualquer linguagem, logo atrás do nosso comando de impressão. E com o Dart não seria diferente.

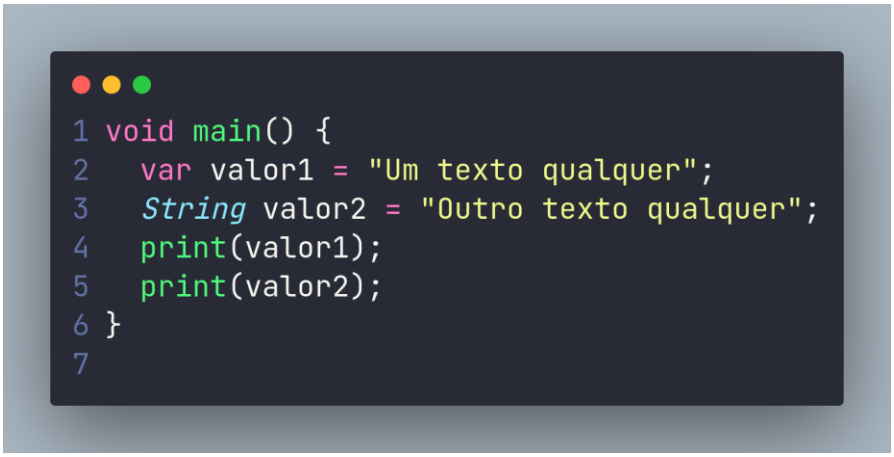
Ao declararmos uma variável no Dart, podemos utilizar duas estratégias:

- A primeira é declarar a variável utilizando a palavra reservada “var”. Dessa forma, a variável receberá o tipo de dado do mesmo tipo do valor informado. Ou seja, se eu criar uma chamada **valor1** e utilizar a palavra reservada

"var", poderei colocar qualquer tipo de valor nela, seja **um número**, **um texto** ou mesmo um **valor booleano**.

- A segunda estratégia seria inferir o tipo de dado da variável no momento da criação da mesma.

👨‍💻 Mão no código



```dart
void main() {
  var valor1 = "Um texto qualquer";
  String valor2 = "Outro texto qualquer";
  print(valor1);
  print(valor2);
}
```

🖨️ Resultado:

```
Um texto qualquer

Outro texto qualquer
```

Um aspecto importante sobre o Dart é o fato de **não possuir tipos de dados primitivos**. Ou seja, todos os tipos de dados são referências de objetos do pacote **dart.core**.

👨‍💻 Mão no código

```dart
void main() {
  double valor1 = 10.15;
  int convertido = valor1.toInt();
  print(convertido);
}
```

🖨️ Resultado:

```
10
```

1.6 Declaração de constantes

A declaração de constante em Dart é tão simples quanto a criação de variáveis. Para declararmos uma constante em nosso código basta adicionarmos a palavra reservada “const” antes da declaração da tipagem da variável. Isso dirá ao Dart que aquela será uma variável imutável, ou seja, uma constante.

Ao declararmos uma constante no Dart, podemos utilizar duas estratégias:

- A primeira é declará-la utilizando a palavra reservada **“const”.**
- A segunda estratégia seria utilizar a palavra reservada **“final”.**

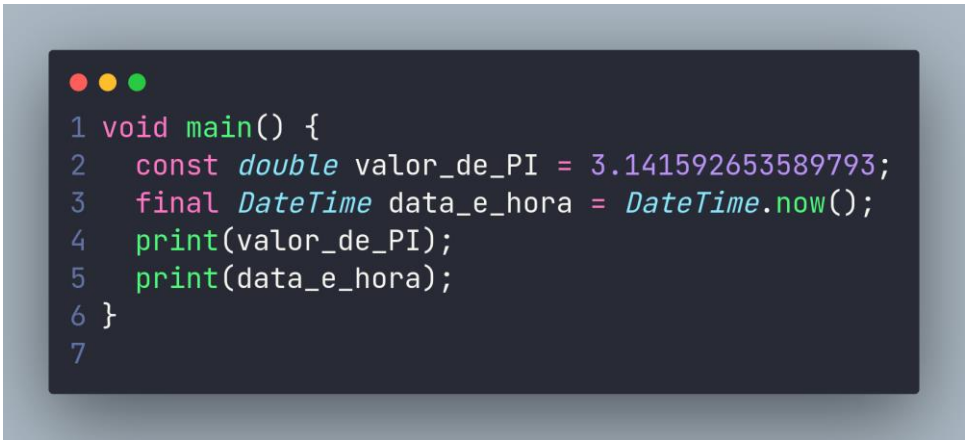
Mas afinal, qual a real diferença entre essas duas implementações? É uma questão de gosto, ou não? Qual devo usar? 🧐

Bom, essas implementações têm uma finalidade em comum, a declaração de uma constante. Porém existe uma diferença fundamental entre elas:

Uma declaração **const** é uma declaração em tempo de compilação, e a declaração **final** é uma declaração em tempo de execução. Mas você deve estar se perguntando “Ok. Mas o que isso quer dizer? 🧐”.

Bom, de forma resumida, uma declaração em tempo de compilação, exige que o valor da variável esteja disponível no momento em que a aplicação é compilada, não podendo depender da execução de algum método. Já a implementação em tempo de execução permite que o valor da constante seja inferido após a execução de um determinado método.

👨‍💻 Mão no código



```
1 void main() {
2   const double valor_de_PI = 3.141592653589793;
3   final DateTime data_e_hora = DateTime.now();
4   print(valor_de_PI);
5   print(data_e_hora);
6 }
7
```

🖨️ Resultado:

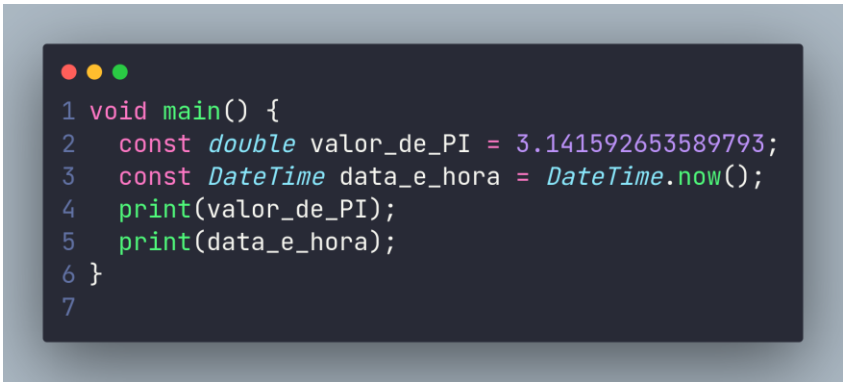
```
3.141592653589793

2021-02-13 17:32:42.507
```

Ok. Mas se eu tentar usar a palavra const na linha 3, o que acontecerá? 🤔

Bom, vejamos o que acontece se tentarmos definir uma constante de tempo de compilação com um valor de tempo de execução.

👨‍💻 Mão no código



```
1 void main() {
2   const double valor_de_PI = 3.141592653589793;
3   const DateTime data_e_hora = DateTime.now();
4   print(valor_de_PI);
5   print(data_e_hora);
6 }
7
```

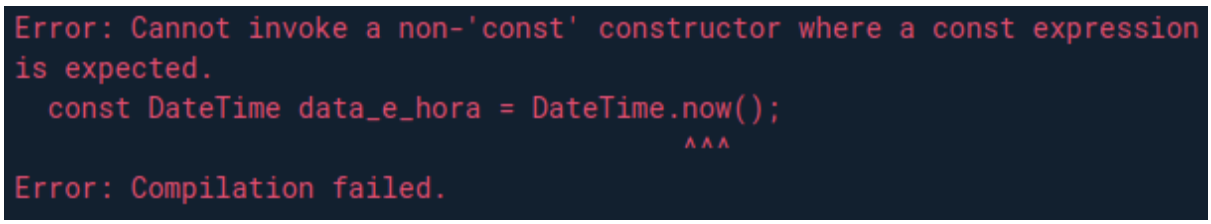


Figura 1.1 – Erro de compilação – Uso de constante de tempo de compilação

Como podemos ver na figura 1.1, o Dart nos apresenta um erro de compilação devido ao fato de estarmos utilizando uma constante de compilação para armazenar um valor de tempo de execução.

1.7 Tipos de dados

Um aspecto muito importante sobre qualquer linguagem são os tipos de dados suportados. E no Dart podemos dizer que estamos vivenciando um sonho, pois o Dart possui uma variedade de tipos de dados e como já foi comentado, todos eles são objetos, já que o Dart não possui tipos primitivos, como acontece com linguagens como o Java e o PHP.

Dentre os principais tipo de dados que o Dart possui, podemos ressaltar os seguintes:

- **String:** Representa uma sequência de **caracteres alfanuméricos**, podendo conter letras, números e caracteres especiais (!,@,#,$,%,&, e etc.);
- **double:** Representa um **valor com fração de até 64 bits**;
- **bool:** Representa um valor booleano **true** ou **false**;

- **int:** Representa um **valor inteiro de até 64 bits**;
- **List:** Representa uma coleção de objetos indexados por um valor inteiro;
- **Map:** Representa uma coleção de objetos indexados por um conjunto de chave/ valor;
- **num:** Representa um número inteiro ou com precisão. Esse objeto é implementado pelas classes **Int** e **Double**;
- **Object: É a classe base para todos os objetos Dart**. Como Object é a raiz da hierarquia de classes Dart, todas as outras classes Dart são subclasses de Object;
- **Future:** Um Future é usado para representar um valor potencial, ou erro, que estará disponível em algum momento no futuro. Os receptores de um futuro podem registrar retornos de chamada que tratam do valor ou erro, uma vez que ele está disponível.
- **Dynamic:** Como o próprio nome já sugere, representa um valor dinâmico. Ou seja, um tipo de dado que pode ser de qualquer tipo e pode receber objetos de qualquer tipo. Enfim, **dinâmico**.

Existem outros objetos no core do Dart, neste tópico tratamos dos principais e mais utilizados em nosso cotidiano de desenvolvimento. Caso deseje conhecer mais sobre os demais tipos de dados do Dart, consulte a documentação oficial da linguagem.

1.8 Operadores matemáticos

O Dart, assim como praticamente qualquer linguagem de programação existente, possui uma série de operadores matemáticos, indo desde as **4 operações básicas** da matemática até operadores complexos como **resto da divisão** e **pré-incremento** e **pós-incremento**.

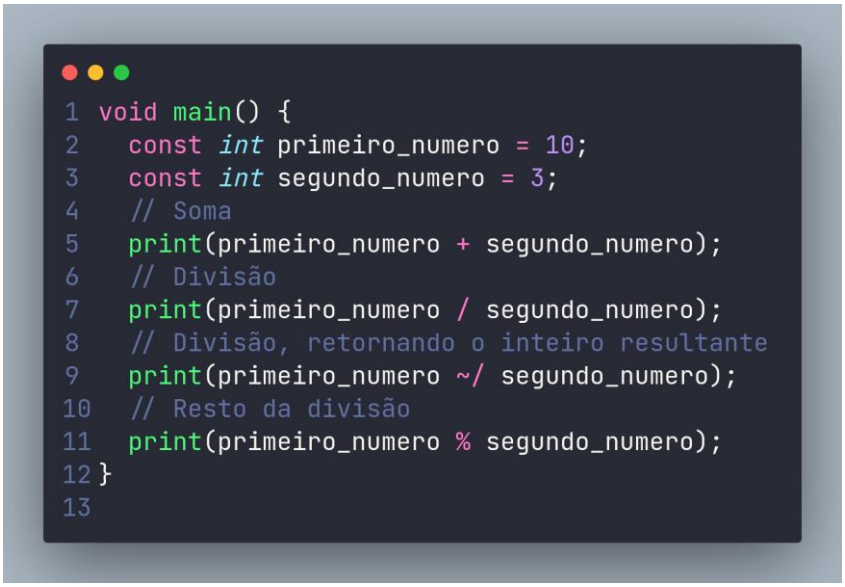
Na tabela 1.1 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.1 - Operadores aritméticos no Dart

Operador	Significado
+	Adição
–	Subtração
-expressão	Menos unário, também conhecido como negação (inverter o sinal da expressão)
*	Multiplicação
/	Divisão
~/	Divisão, retornando o inteiro resultante

%	Resto da divisão, também conhecido como módulo da divisão

👨‍💻 Mão no código



```dart
void main() {
  const int primeiro_numero = 10;
  const int segundo_numero = 3;
  // Soma
  print(primeiro_numero + segundo_numero);
  // Divisão
  print(primeiro_numero / segundo_numero);
  // Divisão, retornando o inteiro resultante
  print(primeiro_numero ~/ segundo_numero);
  // Resto da divisão
  print(primeiro_numero % segundo_numero);
}
```

🖨️ Resultado:

```
13
3.3333333333333335
3
1
```

1.9 Operadores relacionais

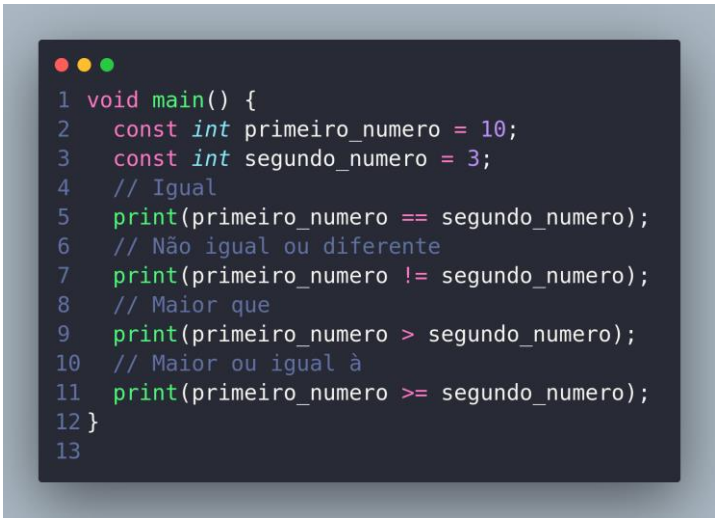
Outro aspecto muito comum e primordial em qualquer linguagem de programação é a capacidade de relacionar e comparar valores. E com o Dart não é diferente, pois com ele é possível comparar valores de forma simples e extremamente prática.

Na tabela 1.2 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.2 - Operadores relacionais no Dart

Operador	Significado	Objetivo prático
==	Igual	Verificar se o valor à esquerda do operador é igual ao que está à direita do mesmo.
!=	Não igual ou diferente	Verificar se o valor à esquerda do operador é diferente do que está à direita do mesmo.
>	Maior que	Verificar se o valor à esquerda do operador é maior que está à direita do mesmo.
<	Menor que	Verificar se o valor à esquerda do operador é menor que está à direita do mesmo.
>=	Maior ou igual à	Verificar se o valor à esquerda do operador é maior ou igual que está à direita do mesmo.
<=	Menor ou igual à	Verificar se o valor à esquerda do operador é menor ou igual que está à direita do mesmo.

👷 Mão no código



```dart
void main() {
  const int primeiro_numero = 10;
  const int segundo_numero = 3;
  // Igual
  print(primeiro_numero == segundo_numero);
  // Não igual ou diferente
  print(primeiro_numero != segundo_numero);
  // Maior que
  print(primeiro_numero > segundo_numero);
  // Maior ou igual à
  print(primeiro_numero >= segundo_numero);
}

```

🖨️ Resultado:

```
false
true
true
true
```

1.10 Operadores Lógicos

Outro grupo de operadores muito importante para qualquer linguagem de programação são os operadores lógicos. Esses operadores são capazes de comparar conjuntos de operadores relacionais, resultando em um valor booleano.

Na tabela 1.3 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.3 - Operadores lógicos no Dart

Operador	Significado
!*expressão*	**Inverte** o resultado lógico (Verdadeiro se torna falso e falso se torna verdadeiro)
||	Retorna verdadeiro caso **ao menos uma das comparações tiverem um valor verdadeiro**. Caso contrário retorna **falso**
&&	Retorna verdadeiro caso **todas as comparações tiverem um valor verdadeiro**. Caso contrário retorna **falso**

👨‍💻 Mão no código

```dart
void main() {
  const int primeiro_numero = 10;
  const int segundo_numero = 3;
  const int terceiro_numero = 14;
  // O primeiro número é maior que os demais
  print(primeiro_numero > segundo_numero && primeiro_numero>terceiro_numero);
  // O segundo número é maior que o primeiro OU menor que o segundo
  print(segundo_numero > primeiro_numero || segundo_numero<terceiro_numero);
  // O primeiro número NÃO é maior que os demais
  print(!(primeiro_numero > segundo_numero && primeiro_numero>terceiro_numero));
}
```

🖨️ Resultado:

```
false
true
true
```

1.11 Operadores de teste

Os operadores de teste são muito úteis quando necessitamos realizar verificações e associações em tempo de execução.

Esses operadores também podem ser utilizados para realizar uma conversão explícita de dados, o que pode ser muito útil em determinados casos e operações do cotidiano de nossas aplicações.

Na tabela 1.4 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.4 - Operadores de teste no Dart

Operador	Significado
`as`	Conversão de tipos (Também utilizado para criação de prefixos em importações)
`is`	Retorna ***True*** caso um objeto seja de um determinado tipo
`is!`	Retorna ***False*** caso um objeto seja de um determinado tipo

👨‍💻 Mão no código

```dart
void main() {
  const num primeiro_numero = 10.15;
  const num segundo_numero = 3;
  final double terceiro_numero = (primeiro_numero as double) * 2;
  print(primeiro_numero is double);
  print(segundo_numero is double);
  print(terceiro_numero is double);
}
```

🖨️ Resultado:

```
true
false
true
```

1.12 Estruturas de decisão

Muitas são as vezes em que necessitamos comparar valores e objetos em nossos sistemas. E como vimos, o Dart está mais que preparado para tal. Mas até o momento só foi possível comparar tais valores, não não podemos tomar decisões com base nesses resultados. É aí que entram as estruturas de decisão. Também conhecidas como estruturas condicionais ou estruturas de fluxo.

Uma estrutura de decisão, como o próprio nome já sugere, nos permite tomar uma decisão com base em um resultado booleano, seja ele vindo de uma comparação ou mesmo do valor de uma variável ou constante do sistema.

1.12.1 Estrutura IF

A estrutura **IF** é a mais comum e usual estrutura de decisão, e está presente em basicamente 100% das linguagens de programação. Conhecer seu funcionamento é fundamental para qualquer programador, seja ele um iniciante ou experiente no universo do desenvolvimento.

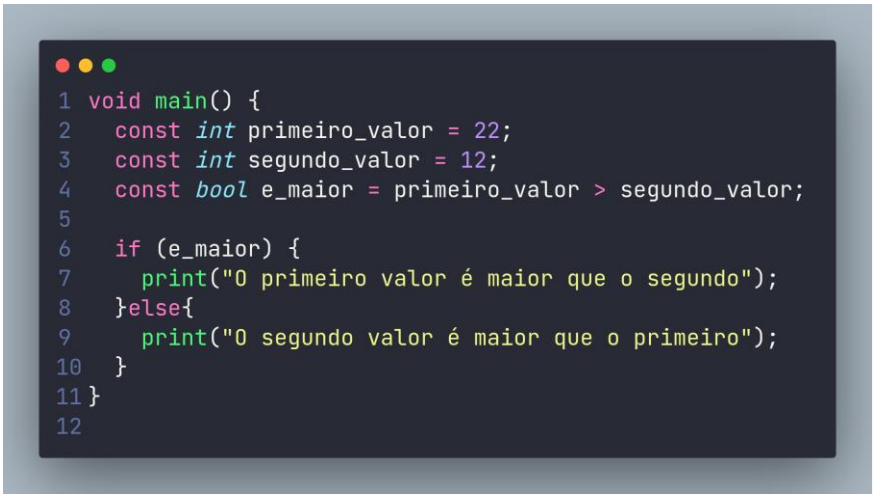
Uma estrutura IF é composta por duas palavras chave, são eles: **if** e **else**. O uso destas duas palavras chave, assim como suas "sub combinações", nos permitem decidir como nosso sistema irá reagir a uma determinada situação. Por exemplo, com uso desta estrutura podemos decidir se um determinado bloco de comandos ou cálculo será executado ou não.

A criação de uma estrutura IF é bastante simples, podemos fazer uso da palavra chave if, onde passamos a condição para sua execução, e caso essa condição seja verdadeira, o bloco de comandos referentes a essa estrutura será executado.

> É importante ressaltar que apesar da estrutura condicional IF normalmente estar associada a estrutura else, seu uso não é obrigatório.

No exemplo abaixo podemos ver um exemplo básico do uso de uma estrutura condicional IF.

👨‍💻 Mão no código



```dart
void main() {
  const int primeiro_valor = 22;
  const int segundo_valor = 12;
  const bool e_maior = primeiro_valor > segundo_valor;

  if (e_maior) {
    print("O primeiro valor é maior que o segundo");
  }else{
    print("O segundo valor é maior que o primeiro");
  }
}

```

🖨️ Resultado:

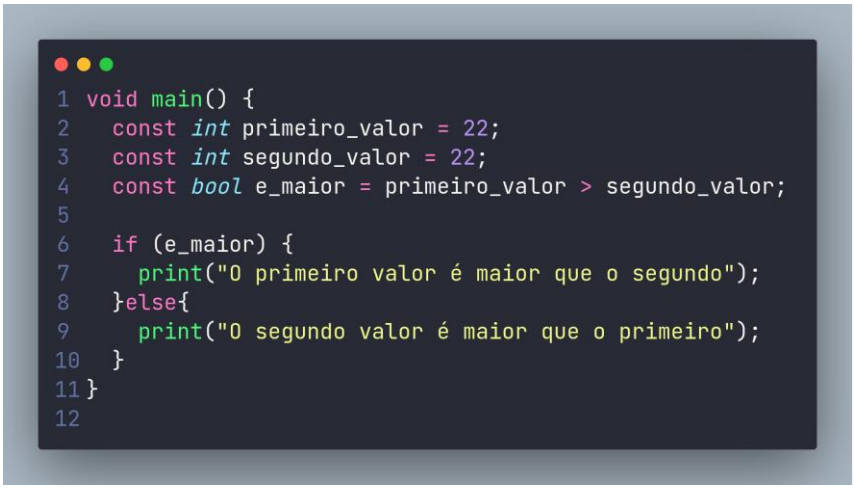
```
O primeiro valor é maior que o segundo
```

Outro aspecto importante a ressaltar é o fato da estrutura “IF” só ser executada se o valor informado como condição seja **True**. Caso contrário, a estrutura “Else” será executada.

Outro aspecto que vale ressaltar é que podemos combinar as duas palavras chaves que compoẽm a estrutura IF para formar mais cenários possíveis.

Pense comigo, e se os valores forem iguais, o que aconteceria? 🧐. Vejamos o que aconteceria neste determinado cenário.

👨‍💻 Mão no código



```dart
void main() {
  const int primeiro_valor = 22;
  const int segundo_valor = 22;
  const bool e_maior = primeiro_valor > segundo_valor;

  if (e_maior) {
    print("O primeiro valor é maior que o segundo");
  }else{
    print("O segundo valor é maior que o primeiro");
  }
}

```

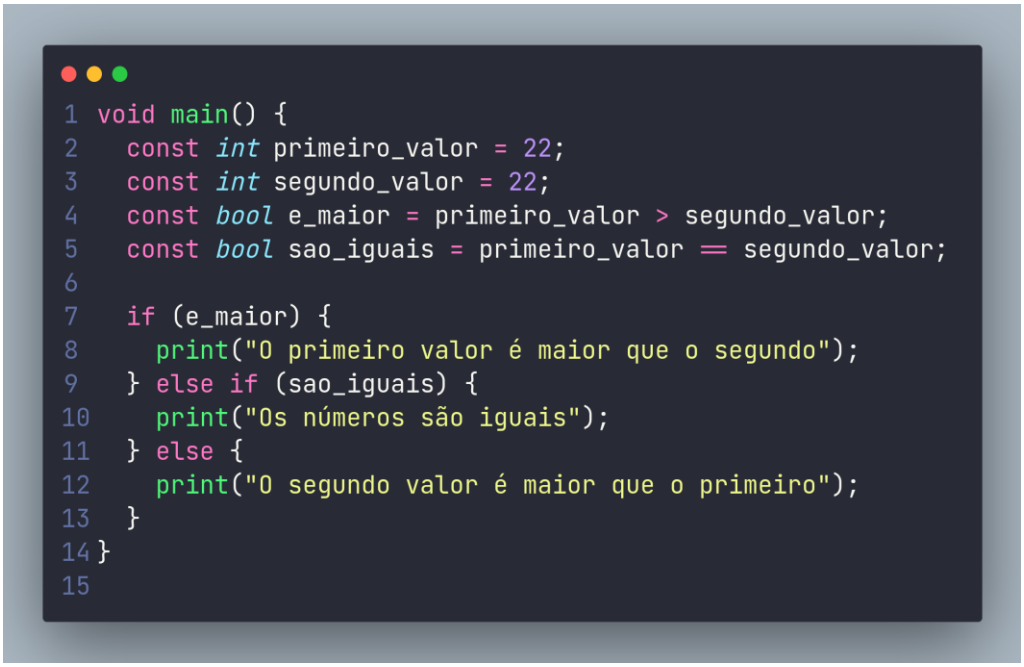
🖨️ Resultado:

```
O segundo valor é maior que o primeiro
```

Logo temos um pequeno problema, pois o segundo valor não é maior que o primeiro, pois ambos são iguais. Mas o que podemos fazer para resolver isso? 🤔

E a resposta é bem simples, vamos verificar a possibilidade de ambos serem iguais. Para isso vamos adicionar mais uma condição à estrutura, essa condição será a palavra chave ”else if”.

👨‍💻 Mão no código



```dart
void main() {
  const int primeiro_valor = 22;
  const int segundo_valor = 22;
  const bool e_maior = primeiro_valor > segundo_valor;
  const bool sao_iguais = primeiro_valor == segundo_valor;

  if (e_maior) {
    print("O primeiro valor é maior que o segundo");
  } else if (sao_iguais) {
    print("Os números são iguais");
  } else {
    print("O segundo valor é maior que o primeiro");
  }
}
```

🖨️ Resultado:

```
Os números são iguais
```

Outro aspecto importante a ressaltar sobre a estrutura IF, é o fato de somente um dos blocos ser executado. Pois quando a estrutura encontra um resultado verdadeiro, o bloco de comandos referente a parte da estrutura é executado e a execução da estrutura como um todo é finalizada.

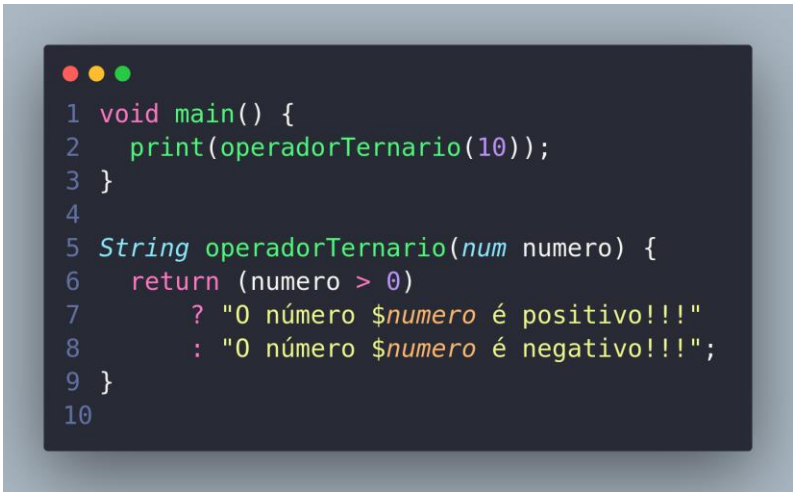
1.12.2 Operador ternário

Um operador ternário é uma forma de declaração de uma estrutura IF/Else, de forma a simplificar sua sintaxe.

Essa abordagem usa uma sintaxe alternativa para cada bloco da estrutura do IF, onde o *"IF"* é substituído por um carácter **"?"**, o qual pode ser lido como **"Caso verdadeiro"**, e o *"Else"* é substituído por um carácter **":"**, o qual pode ser lido como **"Caso falso"**.

Na **figura 1.2** podemos ver uma ilustração da construção de um operador ternário.

👨‍💻 Mão no código



```dart
void main() {
  print(operadorTernario(10));
}

String operadorTernario(num numero) {
  return (numero > 0)
      ? "O número $numero é positivo!!!"
      : "O número $numero é negativo!!!";
}
```

1.12.3 Switch

Essa estrutura de decisão é muito parecida com a estrutura IF, se destacando pelo fato de possuir uma estrutura voltada para validação de diversos cenários distintos. Podemos dizer que a estrutura Switch **permite uma bateria de testes de forma**

organizada e simplificada, uma ótima alternativa para momentos em que temos diversos cenários para validar.

O Switch faz uso de uma estrutura de casos, onde cada caso representa um cenário possível. Sendo que existe também um caso padrão para a possibilidade do valor testado não se enquadre em nenhum dos dados previstos na estrutura do Switch.

Outro aspecto importante sobre essa estrutura é o fato de não possuir um conjunto de delimitadores para seus cenários, ou seja, não utilizamos chaves “{}” para delimitar o encerramento de um cenário, assim como faríamos no IF. Mas sim utilizamos uma instrução break, para que seja possível parar a execução da estrutura sempre que um cenário for atendido.

Também é válido ressaltar que o bloco de cenário padrão (default) não é obrigatório, mas recomendado para informar que nenhum cenário previsto é válido para o valor informado.

No primeiro exemplo encontramos uma validação de strings representando uma estrutura de menu, onde o usuário poderá escolher entre uma gama limitada de opções previamente definidas, onde cada opção representa uma operação dentro do sistema.

Vejamos um exemplo do uso dessa estrutura. Nesse exemplo é simulada a escolha do usuário dentro uma gama limitada de opções. Logo que a escolha é feita, a estrutura Switch entra em ação para verificar a operação escolhida pelo usuário.

👨‍💻 Mão no código

```
void main() {
  // Opção escolhida pelo usuário
  const int opcao = 2;

  switch (opcao) {
    case 1:
      print("Escolheu cadastrar");
      break;
    case 2:
      print("Escolheu editar");
      break;
    case 3:
      print("Escolheu deletar");
      break;
    case 4:
      print("Escolheu listar");
      break;
    default:
      print("Opção inválida");
      break;
  }
}

```

🖨️ Resultado:

```
Escolheu editar
```

1.13 Estruturas de repetição

Neste tópico aprenderemos a utilizar as estruturas de loop. Estruturas estas que nos permitem realizar uma mesma operação uma determinada quantidade de vezes, a qual é controlada por uma variável de controle booleana.

Esse tipo de estrutura é muito utilizada para a manipulação de listas e mapas, devido à estrutura em que eles operam. Porém esse não é seu único uso, apesar de ser o mais comum.

Outro aspecto importante a se ressaltar sobre as estruturas de loop é justamente o fator que as torna o que que são, os loops (também conhecidos com repetições ou voltas). Uma estrutura de loop possui uma condição de parada, e até que essa condição seja atendida, ela não cessa sua execução, o que pode levar a uma execução infinita, também conhecido como loop eterno.

1.13.1 Estrutura de loop While

O **While** é uma estrutura de loop similar ao IF. Da mesma forma, possui uma condição para executar um bloco de comandos. Porém se distingue dele pelo fato de executar o bloco de comandos até que a condição imposta torna-se falsa. Até esse momento chegar, o bloco de comandos previsto é executado.

👨‍💻 Mão no código

```dart
void main() {
  int controlador = 1;
  while (controlador < 10) {
    print("Volta n° $controlador");
    controlador++;
  }
}
```

🖨️ Resultado:

```
Volta n° 1
Volta n° 2
Volta n° 3
Volta n° 4
Volta n° 5
Volta n° 6
Volta n° 7
Volta n° 8
Volta n° 9
```

Talvez você tenha notado algo incomum em nosso comando “print” usual. Foi introduzido um símbolo de cifrão ($). Isso informa ao Dart que desejamos combinar uma String literal ao valor de uma variável ou constante. Muito útil quando desejamos imprimir texto juntamente com referências de variáveis e constantes.

1.13.2 Estrutura de loop Do while

A estrutura de loop **Do While** é praticamente igual a estrutura do **While**, a grande diferença se encontra no modo como a função valida a condição de parada do loop. Enquanto o **While** valida a condição antes de iniciar o loop, o **Do While** o faz ao final da volta. Isso possibilita a execução do loop ao menos uma vez, dando a oportunidade de inicializar o controlador do loop após o início do mesmo. É importante ressaltar que esse tipo de implementação pode ocasionar problemas na execução do loop, como por exemplo torná-lo eterno, ou seja, onde a condição de parada nunca é alcançada.

👨‍💻 Mão no código

```
1 void main() {
2   int controlador = 1;
3   do {
4     print("Volta n° $controlador");
5     controlador++;
6   } while (controlador < 10);
7 }
8
```

🖨️ Resultado:

```
Volta nº 1
Volta nº 2
Volta nº 3
Volta nº 4
Volta nº 5
Volta nº 6
Volta nº 7
Volta nº 8
Volta nº 9
```

Você deve estar se perguntando “OK, mas afinal quando devo usar o While e quando utilizar o Do While? 🤔”. E a resposta é bem simples: Depende 😌.

Eu vou explicar melhor, calma 😅. Caso você tenha uma condição fixa para a execução do loop, ou seja, um ponto de parada calculável, poderá utilizar ambos. Agora se a condição de parada venha a mudar durante a execução do loop, então utilize o Do While, assim poderá verificar se deverá continuar sempre que a volta anterior tiver terminado. Outro cenário em que o Do While é necessário é quando a condição de parada deve ser definida durante o loop e não antes dele, devido a diversos fatores, como por exemplo, ele for obtido por uma entrada do usuário ou retorno de uma função.

1.13.3 Estrutura de loop For

O **For** é uma estrutura de controle que estabelece um loop de repetição baseado em um contador numérico, diferenciando-se dos demais pelo fato de não utilizar uma condição booleana externa, já que a estrutura da sua sintaxe prove tudo que essa função necessita ser executada, impedindo de ações externas ao loop afetem a execução do mesmo.

Outro fator importante a se levar em consideração é que essa estrutura se baseia em uma contagem e não simplesmente em um valor booleano. Isso quer dizer que a repetição será executada um número limitado de vezes, devido a sua estrutura interna de contagem (incremento ou decremento).

👨‍💻 Mão no código

```dart
void main() {
  for (int indice = 1; indice < 10; indice++) {
    print("Volta n° $indice");
  }
}
```

🖨️ Resultado:

```
Volta nº 1

Volta nº 2

Volta nº 3

Volta nº 4
```

1.13.4 Estrutura for-in

A estrutura de loop **for-in** tem um nome muito parecido com o For, porém são estruturas distintas. Enquanto o loop For utiliza um contador para gerenciar as voltas do loop, o **for-in** faz uso de um contador intrínseco para gerenciar as voltas do loop baseada nas posições de uma coleção (Listas, sets e mapas). Isso faz dele uma estrutura de loop destinada para interações em coleções.

👨‍💻 Mão no código

```dart
void main() {
  const List numeros_primos = [2, 3, 5, 7, 11]; //...
  for (int num in numeros_primos) {
    print(num);
  }
}

```

🖨️ Resultado:

```
2
3
5
7
11
```

Capítulo 2 - Introdução ao uso de funções

Outro aspecto muito importante em uma linguagem de programação é a declaração de funções, e no Dart não seria diferente, por esse motivo destinamos um capítulo dedicado ao seu uso, para que possamos nos aprofundar nos mais diversos conceitos que elas possuem em Dart.

Como já vimos, o Dart é “Verdadeiramente” orientado a objetos, isso quer dizer que tudo na linguagem é um objeto, inclusive as funções. Isso mesmo, você não leu errado, mesmo uma função é um objeto.

No Dart todas as funções herdam implicitamente e classe Function nativa do Dart. Isso significa que uma função pode ser atribuída a uma variável ou passada como argumento para outras funções. Você também pode chamar uma instância de uma classe Dart como se fosse uma função.

2.1 Declaração de funções

Como vimos anteriormente, uma função no Dart também é um objeto, porém um objeto “especial”, pois possui características distintas de um “objeto comum”.

Uma função é formada por três partes fundamentais, sendo elas:

- Nome
- Parâmetros
- Tipo de retorno

Na figura 2.1 podemos ver um diagrama apresentado as partes que compõem uma função no Dart.

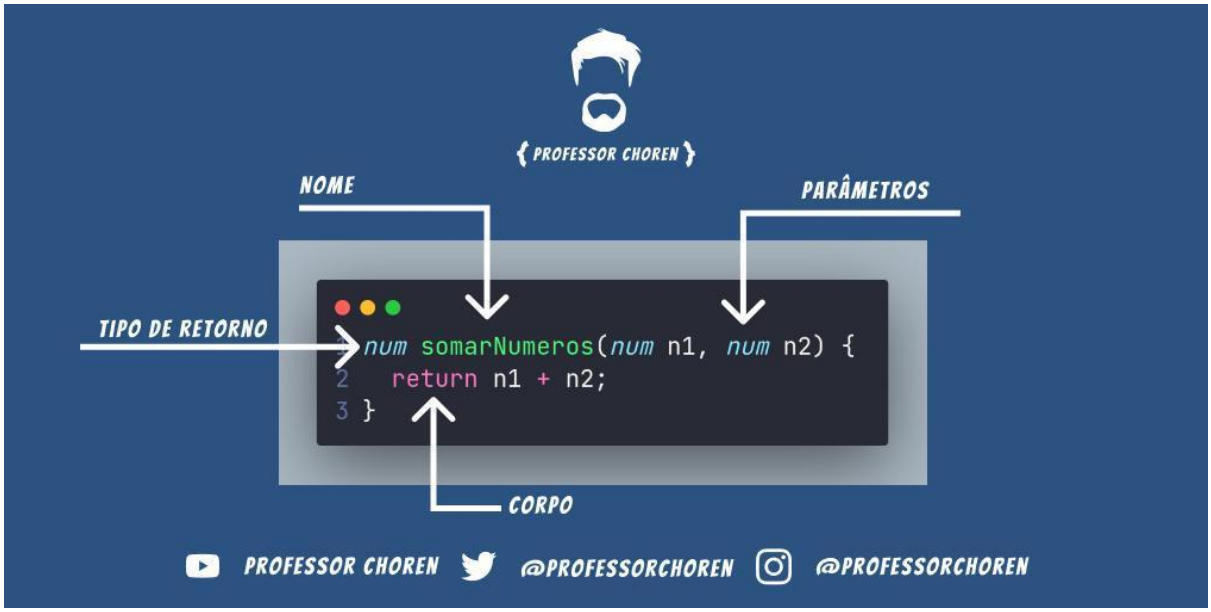


Figura 2.1 - Partes de uma função

É importante ressaltar que uma função pode ou não possuir parâmetros, tudo dependerá da finalidade da função e de sua necessidade de informações externas a seu escopo.

2.2 Declaração de parâmetros

Como vimos anteriormente, uma função pode ou não possuir parâmetros. Isso dependerá da razão de sua existência e da

necessidade de informações externas ao seu escopo. Porém existem alguns aspectos importantes a serem falados sobre o uso de parâmetros.

2.2.1 Parâmetros opcionais

No exemplo apresentado na figura 2.1 vimos a declaração de uma função para a soma de dois números, a qual recebe como parâmetro dois valores numéricos, porém ambos são obrigatórios, ou seja, ambos devem obrigatoriamente ser informados, caso contrário sua execução acarretará um erro em tempo de compilação. Mas e se necessitarmos declarar parâmetros opcionais, ou seja, que podem ou não ser utilizados.

Para esse exemplo, vamos recriar nossa função de soma, agora para que ela permita um terceiro número, mas este sendo opcional.

👨‍💻 Mão no código

```dart
num somarNumeros(num n1, num n2, [num n3]) {
  if (n3 != null) {
    return n1 + n2 + n3;
  }
  return n1 + n2;
}
```

Como podemos ver no exemplo, utilizamos um par de colchetes ([]) para declarar que o terceiro parâmetro é opcional. No corpo da

função incluímos uma condicional IF para verificar se devemos utilizar o terceiro parâmetro ou não. Dessa forma evitando um erro em tempo de execução devido ao fato do parâmetro estar nulo.

2.2.2 Parâmetros nomeados

Outro aspecto interessante sobre o Dart, é a possibilidade de nomear os parâmetros de uma função. Dessa forma podemos informar nossos parâmetros na ordem em que desejarmos, ou até mesmo não informá-los, de forma a criar o mesmo efeito do parâmetro opcional. Mas antes de entendermos como podemos fazer isso, é importante entender como o Dart interpreta os parâmetros de uma função.

O Dart, por padrão, faz uso da declaração posicional de parâmetros. Ou seja, ao chamarmos um método, devemos passar seus parâmetros na mesma ordem em que eles foram declarados na criação da mesma, pois caso contrário, o Dart irá colocar os valores nos locais errados e isso acarretará em um erro na execução da função. Vejamos um exemplo.

👨‍💻 Mão no código

```dart
void main() {
  // Na ordem correta
  print(imprimirSaudacao("Anderson","M"));
  // Com parâmetros invertidos
  print(imprimirSaudacao("F","Michele"));
}

String imprimirSaudacao(String nome, String genero) {
  if (genero == "M") {
    return "Bem-vindo Sr. $nome";
  }
  return "Bem-vindo Sra. $nome";
}

```

🖨️ Resultado:

```
Bem-vindo Sr.Anderson
Bem-vindo Sra.F
```

Como pode perceber, ao inverter a ordem dos parâmetros houve um pequeno erro na impressão dos dados. Apesar de não haver um erro de compilação devido a ambos os parâmetros serem Strings, a ordem pode afetar a execução correta da função.

É aí que entram os parâmetros nomeados. Com eles podemos dar nomes a cada um dos parâmetros da função, dessa forma podemos informá-los na ordem em que desejarmos, pois o Dart irá associar o nome do parâmetro e não sua posição declarativa. Vejamos como podemos mudar nossa função para aceitar parâmetros nomeados.

👨‍💻 Mão no código

```dart
void main() {
  print(imprimirSaudacao(nome: "Anderson", genero: "M"));
  print(imprimirSaudacao(genero: "F", nome: "Michele"));
}

String imprimirSaudacao({String nome, String genero}) {
  if (genero == "M") {
    return "Bem-vindo Sr. $nome";
  }
  return "Bem-vinda Sra. $nome";
}

```

🖨️ Resultado:

```
Bem-vindo Sr. Anderson
Bem-vinda Sra. Michele
```

Perceba que desta vez obtivemos o resultado esperado, pois estamos nomeando nossos parâmetros ao invés de utilizarmos de forma posicional.

Você já deve ter notado que no momento da declaração dos parâmetros de nossa função, colocamos os mesmos dentro um par de chaves ({}). Isso informa para o Dart que nossos parâmetros **nome** e **gênero** são nomeados. Assim podemos passá-los na ordem em que desejarmos.

É importante ressaltar que todos os parâmetros nomeados são considerados pelo Dart como opcionais. Dessa forma, eles podem ou não ser informados na chamados do método. Dessa forma, é necessário ter cuidado com seu uso, pois podemos ter

problemas tanto de execução, quanto de lógica, em nossas aplicações.

2.3 Arrow Function

Outro aspecto importante a ser ressaltado no Dart é o fato do suporte à *"Arrow function"*, também conhecida como *"lambda function"* ou função anônima. Para maiores detalhes, temos um ótimo artigo na Wikipédia sobre esse assunto.

Mas a critério de resumo, esse tipo de declaração nos permite criar funções com uma sintaxe mais simples e limpa, o que deixa nosso código menor e consequentemente mais fácil de entender e de dar manutenção.

Para nosso exemplo, vamos recriar nosso exemplo anterior, desta vez utilizando arrow function e operador ternário. Dessa forma poderemos reduzir as linhas de nossa função, tornando-a mais simples e fácil de compreender.

Mão no código

```dart
void main() {
  // Na ordem correta
  print(imprimirSaudacao("Anderson", "M"));
  // Com parâmetros invertidos
  print(imprimirSaudacao("Michele", "F"));
}

String imprimirSaudacao(String nome, String genero) =>
    (genero == "M") ? "Bem-vindo Sr. $nome" : "Bem-vinda Sra. $nome";
```

Resultado:

```
Bem-vindo Sr. Anderson
Bem-vinda Sra. Michele
```