

Desenvolvimento Mobile com

Google Flutter





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](https://www.facebook.com/groups/431311894533110)

 @PROFESSORCHOREN

 @PROFESSORCHOREN

 PROFESSORCHOREN@GMAIL.COM

 ANDERSONCHOREN.COM.BR

Sumário

Capítulo 1 - Introdução à linguagem Dart	17
1.1 Instalação do Flutter/Dart	20
1.2 Uso de comentários no código	22
1.3 O método main	23
1.4 Nosso 'Hello World' de cada dia	23
1.5 Declaração de variável	24
1.6 Declaração de constantes	26
1.7 Tipos de dados	29
1.8 Operadores matemáticos	31
1.9 Operadores relacionais	32
1.10 Operadores Lógicos	34
1.11 Operadores de teste	36
1.12 Estruturas de decisão	37
1.12.1 Estrutura IF	38
1.12.2 Operador ternário	41
1.12.3 Switch	42
1.13 Estruturas de repetição	45
1.13.1 Estrutura de loop While	45
1.13.2 Estrutura de loop Do while	47
1.13.3 Estrutura de loop For	49
1.13.4 Estrutura for-in	50
Capítulo 2 - Introdução ao uso de funções	52
2.1 Declaração de funções	52
2.2 Declaração de parâmetros	53
2.2.1 Parâmetros opcionais	54
2.2.2 Parâmetros nomeados	55
2.3 Arrow Function	58



Capítulo 3 - Manipulação de Strings	59
3.1 Propriedades	60
3.1.1 isEmpty	60
3.1.2 isEmpty	61
3.1.3 length	61
3.2 Métodos	62
3.2.1 Método contains	62
3.2.2 Método startsWith	63
3.2.3 Método endsWith	64
3.2.4 Método substring	64
3.2.5 Método split	65
3.2.6 Método toUpperCase	67
3.2.7 Método toLowerCase	68
3.2.8 Método trim	68
3.2.9 Objeto StringBuffer	69
Capítulo 4 - Manipulação de números	71
4.1 Propriedades	71
4.1.1 Propriedade isFinite	71
4.1.2 Propriedade isInfinite	72
4.1.3 Propriedade isNegative	73
4.2 Métodos	73
4.2.1 Método round	73
4.2.2 Método toDouble	74
4.2.3 Método toInt	75
4.2.4 Método toString	75
4.2.5 Método toStringAsFixed	76
4.2.5 Método parse	76
Capítulo 5 - Introdução a Listas	78
5.1 Declaração de listas	78



5.2	Propriedades	79
5.2.1	Propriedade isEmpty	79
5.2.2	Propriedade isEmpty	80
5.2.3	Propriedade Length	81
5.2.4	Propriedade Reversed	82
5.3	Métodos	85
5.3.1	Método add	85
5.3.2	Método remove	87
5.3.3	Método removeAt	88
5.3.4	Método clear	89
5.3.5	Método contains	90
5.3.6	Método sort	91
5.3.7	Método generate	92
5.3.8	Método elementAt	93
5.3.9	Método every	94
5.3.10	Método forEach	96
5.3.11	Método indexOf	97
5.3.12	Método join	98
5.3.13	Método reduce	99
5.3.14	Método where	100
5.3.15	Método toList	101
Capítulo 6	- Introdução a Mapas	103
6.1	Declaração de mapas	103
6.2	Adicionando elementos ao mapa	104
6.3	Propriedades	105
6.3.1	Propriedade isEmpty	105
6.3.2	Propriedade isEmpty	106
6.3.3	Propriedade Length	107
6.4	Métodos	108



6.4.1 Método remove	108
6.4.2 Método clear	109
6.4.3 Método containsValue	110
6.4.4 Método containsKey	111
6.4.5 Método update	112
6.4.6 Método map	114
6.4.7 Método forEach	115
6.4.8 Método toString	117
Capítulo 7 - Processamento assíncrono	118
7.1 Introdução a processamento assíncrono?	120
7.1.1 Assincronismo	120
7.2 Palavra-chave await	121
7.3 Palavra-chave async	123
7.4 Objeto Future	124
Capítulo 8 - Trabalhando com bibliotecas	126
8.1 Biblioteca math	127
8.1.1 PI	127
8.1.2 min	128
8.1.3 max	129
8.1.4 sqrt	130
8.1.5 pow	131
8.2 Biblioteca convert	132
8.2.1 json_encode	133
8.2.2 json_decode	134
8.3 Uso de bibliotecas da comunidade	134
8.3.1 Arquivo pubspec.yaml	135
8.3.2 Biblioteca date_format	138
8.3.3 Biblioteca faker	139
8.3.4 Biblioteca strings	140



8.3.5 Biblioteca password_strength	141
8.3.7 Biblioteca password	143
8.3.6 Biblioteca cpfcnpj	144
Capítulo 9 - Orientação a objetos com Dart	146
9.1 Classes	146
9.1.1 Visibilidade	148
9.1.2 Propriedades da classe	149
9.1.3 Métodos da classe	150
9.1.4 Encapsulamento e métodos acessores	151
9.1.5 Método construtor	152
9.1.6 Método toString	154
9.1.7 Instância da classe	155
9.1.8 Referência "this"	156
9.2 Palavra-Chave 'static'	157
9.3 Herança entre Objetos	160
9.3.1 Chamando o construtor da classe mãe	162
9.4 Abstração de Classes	163
9.5 Genéricos	165
9.6 Palavra-chave is	168
9.7 Palavra-chave as	169
Capítulo 10 - Enumeradores	171
10.1 Uso de enumeradores	171
10.2 Enumerador com uso de construtor	172
Capítulo 11 - Exceções	174
11.1 O que é uma exceção	174
11.2 Bloco try/catch	176
11.2.1 Try	177
11.2.2 Catch	177
11.2.3 Finally	178

Capítulo 12 - Null Safe	179
12.1 Permitindo valores nulos	181
12.2 Trabalhando com valores nulos	183
12.2.1 Operador ??	183
12.2.2 Operador ?	185
12.2.3 Combinando poderes	187
12.3 Trabalhando com Null Safe e parâmetros nomeados	188
Capítulo 13 - Introdução ao Flutter	191
13.1 O que é o Flutter	192
13.2 Aplicações Nativas	193
13.3 A Flutter Engine	194
13.4 Hot restart	195
13.5 Estrutura básica de um projeto	196
13.5.1 Estrutura de pastas e arquivos do Flutter	196
13.5.2 Manifesto pubspec.yaml	199
Capítulo 14 - Introdução aos Widgets	202
14.1 O que são Widgets	202
14.2 A árvore de Widgets	203
14.3 A classe "View" do Flutter	204
14.4 Widgets Stateless	204
14.5 Widgets StatefulWidget	205
Capítulo 15 - Widgets Básicos	207
15.1 Container	207
15.1.1 Propriedade child	208
15.1.2 Propriedade alignment	208
15.1.3 Propriedade color	209
15.1.4 Propriedade margin	210
15.1.5 Propriedade padding	211
15.1.6 Propriedade width e height	211



15.2 Text	212
15.2.1 style	212
15.2.1.1 color	213
15.2.1.2 decoration	214
15.2.1.3 fontFamily	214
15.2.1.4 fontSize	215
15.2.1.5 fontStyle	215
15.2.1.6 fontWeight	216
15.3.1 textAlign	217
15.3 Column	217
15.3.1 Propriedade children	218
15.3.2 Propriedade mainAxisAlignment	218
15.3.3 Propriedade crossAxisAlignment	219
15.4 Row	220
Capítulo 16 – Widgets Material Design	222
16.1 O Material Design	222
16.2 Esqueleto da aplicação Material Design	223
16.2 Scaffold	224
16.2.1 AppBar	225
16.2.1.1 Propriedade title	226
16.2.1.2 Propriedade leading	226
16.2.2 body	227
16.2.3 floatingActionButton	229
16.2.4 bottomNavigationBar	230
16.2.4.1 Propriedade items	230
16.2.4.2 Propriedade currentIndex	231
16.2.4.3 Propriedade selectedItemColor	232
16.2.4.4 Propriedade onTap	233
16.2.5 drawer	234



16.3	TextButton	236
16.3.1	Propriedade child	237
16.3.2	Propriedade onPressed	238
16.3.3	Propriedade style	239
16.3.3.1	Propriedade primary	239
16.3.3.2	Propriedade background	240
16.3.3.3	Propriedade elevation	241
16.3.3.4	Propriedade shadowColor	242
16.4	Image	243
16.4.1	Carregando imagens locais	244
16.4.2	Carregando imagens através da internet	245
16.4.3	Propriedade width e height	246
16.4.2	Propriedade fit	247
16.4.3	Propriedade loadingBuilder	249
16.5	Icon	250
16.5.1	Propriedade size	252
16.5.2	Propriedade color	253
16.6	IconButton	254
16.6.1	icon	254
16.6.2	onPressed	255
16.6.3	iconSize	257
16.6.4	color	258
16.7	ElevatedButton	259
16.7.1	Propriedade Child	259
16.3.2	Propriedade onPressed	260
16.3.3	Propriedade style	262
16.3.3.1	Propriedade primary	262
16.3.3.2	Propriedade elevation	263
16.3.3.3	Propriedade shadowColor	264
16.8	Card	265



16.8.1 Propriedade Child	265
16.8.2 Propriedade color	266
16.8.3 Propriedade elevation	268
16.8.4 Propriedade shadowColor	269
16.8.6 Propriedade margin	270
16.8.6 Propriedade shape	272
16.9 Divider	273
16.9.1 Propriedade color	274
16.9.2 Propriedade thickness	276
16.9.3 Propriedade height	277
16.9.4 Propriedades indent e endindent	278
16.10 Chip	280
16.10.1 Propriedade label	280
16.10.2 Propriedade labelPadding	281
16.10.3 Propriedade labelStyle	282
16.10.4 Propriedade backgroundColor	284
16.10.5 Propriedade avatar	285
16.10.6 Propriedade elevation	287
16.10.7 Propriedade padding	288
16.10.8 Propriedade shape	290
16.11 DropdownButton	291
16.11.1 Propriedade dropdownColor	293
16.11.2 Propriedade hint	294
16.11.3 Propriedade style	295
16.11.3 Propriedade isExpanded	297
16.11.4 Propriedade underline	298
16.12 OutlinedButton	300
16.12.1 Propriedade style	301
16.12.1.1 Propriedade primary	301
16.12.1.2 Propriedade side	302



16.13	TabBar	304
16.13.1	DefaultTabController	304
16.13.2	TabBarView	305
16.13.3	TabBar	306
Capítulo 17	- Widgets de Layout	308
17.1	Single-child layout widgets	308
17.1.1	Align	308
17.1.1.1	Propriedade alignment	309
17.1.2	Center	310
17.1.3	Expanded	311
17.1.3.1	Propriedade flex	313
17.1.4	Padding	314
17.1.5	SizedBox	315
17.2	Multi-child layout widgets	316
17.2.1	Stack	316
17.2.2	Wrap	319
17.2.2.1	children	319
17.2.2.2	alignment	320
17.2.2.3	direction	322
17.2.2.4	runSpacing	324
17.2.2.5	spacing	326
Capítulo 18	- Widgets de formulário	328
18.1	TextField	328
18.1.1	decoration	328
18.1.1.1	Border	329
18.1.1.2	HintText	330
18.1.1.3	HintStyle	331
18.1.1.4	ErrorText	332
18.1.1.5	ErroStyle	333



18.1.1.6 Icon	334
18.1.2 enabled	335
18.1.3 style	336
18.1.4 expands	337
18.1.5 inputFormatters	338
18.1.6 keyboardType	339
18.1.7 maxLength	341
18.1.8 obscureText	342
18.1.9 obscuringCharacter	343
18.1.10 onChanged	344
18.1.11 onSubmitted	346
18.2 Form	347
18.3 TextFormField	348
18.4 Checkbox	349
18.6 Radio Button	353
18.7 showDatePicker	355
18.8 showTimePicker	357
18.9 Slider	358
18.10 Switch	360
Capítulo 19 - Widgets de lista	362
19.1 SingleChildScrollView	362
19.2 ListView	365
19.2.1 Propriedade scrollDirection	366
19.2.2 Propriedade reverse	368
19.2.3 Propriedade itemExtent	369
19.2.5 ListView.builder	370
19.3 ListTile	371
19.3.1 title	372
19.3.2 subtitle	373



19.3.3 leading	374
19.3.4 tileColor	375
19.3.5 trailing	377
19.3.6 dense	378
19.4 GridView	379
19.4.1 GridView.builder	379
19.4.2 GridView.count	380
Capítulo 20 - Navegação entre telas	382
20.1 Sistema de pilha	382
20.2 MaterialPageRoute	383
20.3 Navigator	384
20.3.1 push	384
20.3.2 pushNamed	386
20.3.3 pop	387
20.4 Enviando dados entre telas	388
20.4.1 Enviando dados	388
20.4.2 Recebendo dados	389
Capítulo 21 - Widgets de notificação	391
21.1 showDialog	391
21.2 SimpleDialog	392
21.3 AlertDialog	394
21.3.1 title	394
21.3.2 titlePadding	396
21.3.3 titleTextStyle	397
21.3.4 content	397
21.3.5 contentPadding	398
21.3.6 contentTextStyle	400
21.3.7 actions	401
21.3.8 actionsPadding	402



21.3.9 shape	403
21.3.11 backgroundColor	405
21.4 SnackBar	406
21.4.1 content	406
21.4.2 duration	408
21.4.3 backgroundColor	409
21.4.4 shape	410
21.4.5 behavior	411
21.4.6 action	413
21.5 MaterialBanner	414
21.5.1 content	414
21.5.2 leading	415
21.5.3 background	416
21.5.4 actions	416
21.6 ProgressIndicator	419
21.6.1 CircularProgressIndicator	419
21.6.1.1 strokeWidth	420
21.6.1.2 value	422
21.6.1.3 valueColor	423
21.6.1.4 backgroundColor	424
21.6.2 LinearProgressIndicator	425
21.6.2.2 value	427
21.6.2.3 valueColor	428
21.6.2.4 backgroundColor	429
Capítulo 22 - Widgets de estilização	431
22.1 ThemeData	432
22.1.1 colorScheme	433
22.1.2 primarySwatch	434
22.1.4 scaffoldBackgroundColor	436



22.1.5 appBarTheme	437
22.1.6 textTheme	439
22.1.7 chipTheme	441
22.1.8 cardTheme	444
22.1.9 elevatedButtonTheme	445
22.1.10 inputDecorationTheme	446
22.1.11 floatingActionButtonTheme	448
22.1.12 sliderTheme	449
22.1.14 snackBarTheme	451
22.2 Theme	452
22.3 DarkTheme	453
22.4 ThemeMode	453
22.5 Material Design 3	454
Capítulo 23 - MediaQuery	457
23.1 Propriedade size	457
23.2 Propriedade orientation	459
Capítulo 24 - SQLite	461
24.1 O que é SQLite	462
24.2 Uso consciente do SQLite	462
24.3 Biblioteca sqflite	462
24.3 Criando um banco de dados	463
24.4 Inserindo registros	465
24.5 Buscando registros	466
24.6 Modificando registros	467
24.7 Removendo registros	469
Capítulo 25 - Consumo de Web Services	470
25.1 O que são Serviços Web	470
25.2 O que são APIs	471
25.2.1 RestFul API	472



25.2.2 Verbos HTTP	472
25.3 Biblioteca HTTP	474
25.4 Tratando os dados	477
25.4.1 Método decode	478
25.5 FutureBuilder	479

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 0.0 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

```
1 // Comentário de única linha
2
3 /*
4  * Comentário
5  * de
6  * múltiplas
7  * linhas
8  */
9
10 /// 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 void main() {  
2   // Aqui ficará seu código  
3 }  
4
```

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

de lá utilizarmos o comando `print`, passando como parâmetro o valor que desejamos imprimir no console.

👨‍💻 Mão no código

```
1 void main() {  
2   print('Olá mundo em Dart');  
3 }  
4
```

🖨️ Resultado:

Olá mundo em Dart

1.5 Declaração de variável

A declaração de uma 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

```
1 void main() {  
2   var valor1 = "Um texto qualquer";  
3   String valor2 = "Outro texto qualquer";  
4   print(valor1);  
5   print(valor2);  
6 }  
7
```

 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

```
1 void main() {  
2   double valor1 = 10.15;  
3   int convertido = valor1.toInt();  
4   print(convertido);  
5 }
```

 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
```

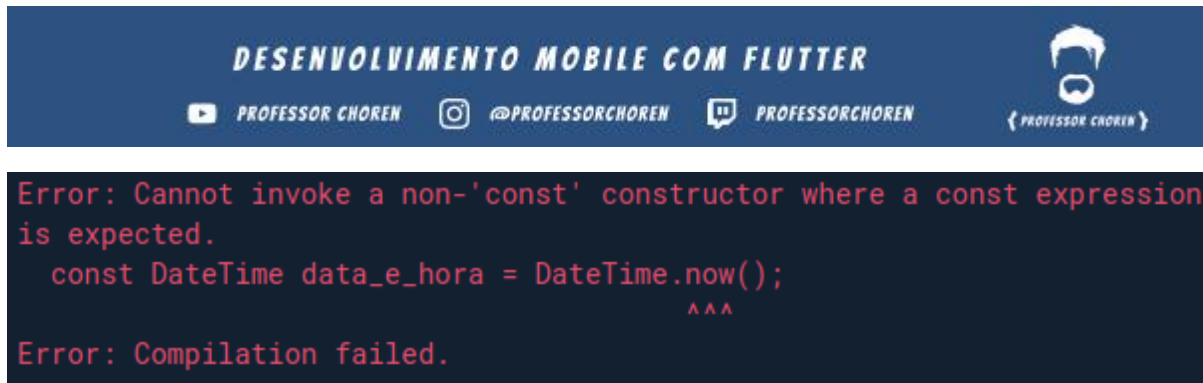


Figura 1.1 - Erro de compilação - Uso de constante de tempo de compilação

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

```

1 void main() {
2   const int primeiro_numero = 10;
3   const int segundo_numero = 3;
4   // Soma
5   print(primeiro_numero + segundo_numero);
6   // Divisão
7   print(primeiro_numero / segundo_numero);
8   // Divisão, retornando o inteiro resultante
9   print(primeiro_numero ~/ segundo_numero);
10  // Resto da divisão
11  print(primeiro_numero % segundo_numero);
12 }
13

```

🖨️ 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

```
1 void main() {  
2   const int primeiro_numero = 10;  
3   const int segundo_numero = 3;  
4   // Igual  
5   print(primeiro_numero == segundo_numero);  
6   // Não igual ou diferente  
7   print(primeiro_numero != segundo_numero);  
8   // Maior que  
9   print(primeiro_numero > segundo_numero);  
10  // Maior ou igual à  
11  print(primeiro_numero >= segundo_numero);  
12}  
13
```

 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
<code>! expressão</code>	Inverte o resultado lógico (Verdadeiro se torna falso e falso se torna verdadeiro)
<code> </code>	Retorna verdadeiro caso ao menos uma das comparações tiverem um valor verdadeiro . Caso contrário retorna falso
<code>&&</code>	Retorna verdadeiro caso todas as comparações tiverem um valor verdadeiro . Caso contrário retorna falso



Mão no código

```

1 void main() {
2   const int primeiro_numero = 10;
3   const int segundo_numero = 3;
4   const int terceiro_numero = 14;
5   // O primeiro número é maior que os demais
6   print(primeiro_numero > segundo_numero && primeiro_numero > terceiro_numero);
7   // O segundo número é maior que o primeiro OU menor que o segundo
8   print(segundo_numero > primeiro_numero || segundo_numero < terceiro_numero);
9   // O primeiro número NÃO é maior que os demais
10  print(!(primeiro_numero > segundo_numero && primeiro_numero > terceiro_numero));
11 }
12

```



Resultado:

false

true

true

1.11 Operadores de teste

Os operadores de teste são muito úteis quando precisamos 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

```
1 void main() {  
2   const num primeiro_numero = 10.15;  
3   const num segundo_numero = 3;  
4   final double terceiro_numero = (primeiro_numero as double) * 2;  
5   print(primeiro_numero is double);  
6   print(segundo_numero is double);  
7   print(terceiro_numero is double);  
8 }  
9
```

 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

```
1 void main() {  
2   const int primeiro_valor = 22;  
3   const int segundo_valor = 12;  
4   const bool e_maior = primeiro_valor > segundo_valor;  
5  
6   if (e_maior) {  
7     print("O primeiro valor é maior que o segundo");  
8   }else{  
9     print("O segundo valor é maior que o primeiro");  
10  }  
11}  
12
```

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 compõem 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

```
1 void main() {  
2   const int primeiro_valor = 22;  
3   const int segundo_valor = 22;  
4   const bool e_maior = primeiro_valor > segundo_valor;  
5  
6   if (e_maior) {  
7     print("O primeiro valor é maior que o segundo");  
8   }else{  
9     print("O segundo valor é maior que o primeiro");  
10  }  
11}  
12
```

🖨️ 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

```
1 void main() {  
2   const int primeiro_valor = 22;  
3   const int segundo_valor = 22;  
4   const bool e_maior = primeiro_valor > segundo_valor;  
5   const bool sao_iguais = primeiro_valor == segundo_valor;  
6  
7   if (e_maior) {  
8     print("O primeiro valor é maior que o segundo");  
9   } else if (sao_iguais) {  
10    print("Os números são iguais");  
11  } else {  
12    print("O segundo valor é maior que o primeiro");  
13  }  
14}  
15
```

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

```
1 void main() {  
2   print(operadorTernario(10));  
3 }  
4  
5 String operadorTernario(num numero) {  
6   return (numero > 0)  
7     ? "0 número $numero é positivo!!!"  
8     : "0 número $numero é negativo!!!";  
9 }  
10
```

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

```
1 void main() {  
2     // Opção escolhida pelo usuário  
3     const int opcao = 2;  
4  
5     switch (opcao) {  
6         case 1:  
7             print("Escolheu cadastrar");  
8             break;  
9         case 2:  
10            print("Escolheu editar");  
11            break;  
12         case 3:  
13            print("Escolheu deletar");  
14            break;  
15         case 4:  
16            print("Escolheu listar");  
17            break;  
18         default:  
19            print("Opção inválida");  
20            break;  
21     }  
22 }  
23
```



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 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

```
1 void main() {  
2   int controlador = 1;  
3   while (controlador < 10) {  
4     print("Volta nº $controlador");  
5     controlador++;  
6   }  
7 }
```

 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
```

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 nº $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

```
1 void main() {  
2   for (int indice = 1; indice < 10; indice++) {  
3     print("Volta nº $indice");  
4   }  
5 }
```

🖨️ 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

```
1 void main() {  
2   const List numeros_primos = [2, 3, 5, 7, 11]; // ...  
3   for (int num in numeros_primos) {  
4     print(num);  
5   }  
6 }  
7
```



 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 a 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.

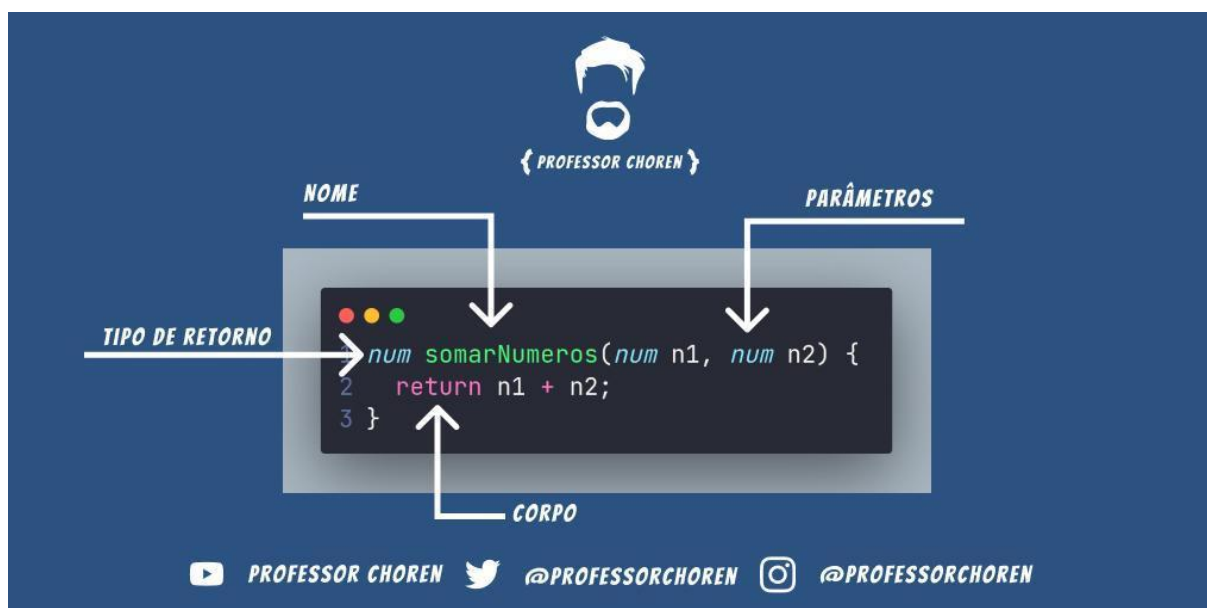


Figura 2.1 – Partes de uma função

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

```
1 num somarNumeros(num n1, num n2, [num n3]) {  
2   if (n3 != null) {  
3     return n1 + n2 + n3;  
4   }  
5   return n1 + n2;  
6 }
```

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

```
1 void main() {  
2   // Na ordem correta  
3   print(imprimirSaudacao("Anderson","M"));  
4   // Com parâmetros invertidos  
5   print(imprimirSaudacao("F","Michele"));  
6 }  
7  
8 String imprimirSaudacao(String nome, String genero) {  
9   if (genero == "M") {  
10    return "Bem-vindo Sr. $nome";  
11  }  
12  return "Bem-vindo Sra. $nome";  
13 }  
14
```

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

```
1 void main() {  
2   print(imprimirSaudacao(nome: "Anderson", genero: "M"));  
3   print(imprimirSaudacao(genero: "F", nome: "Michele"));  
4 }  
5  
6 String imprimirSaudacao({String nome, String genero}) {  
7   if (genero == "M") {  
8     return "Bem-vindo Sr. $nome";  
9   }  
10  return "Bem-vinda Sra. $nome";  
11}  
12
```

🖨️ 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

```
1 void main() {  
2   // Na ordem correta  
3   print(imprimirSaudacao("Anderson", "M"));  
4   // Com parâmetros invertidos  
5   print(imprimirSaudacao("Michele", "F"));  
6 }  
7  
8 String imprimirSaudacao(String nome, String genero) =>  
9   (genero == "M") ? "Bem-vindo Sr. $nome" : "Bem-vinda Sra. $nome";
```



Resultado:

Bem-vindo Sr. Anderson

Bem-vinda Sra. Michele