

Linguagem Dart



Introdução

- Dart é uma linguagem de programação apresentada pelo Google, em 2011. A ideia inicial era substituir o JavaScript como a principal linguagem embutida em navegadores.
- Dart começou a aparecer efetivamente por conta do framework Flutter para desenvolvimento de aplicações nativas, tanto para iOS quanto para Android, embora também seja possível desenvolver aplicações web e desktop.



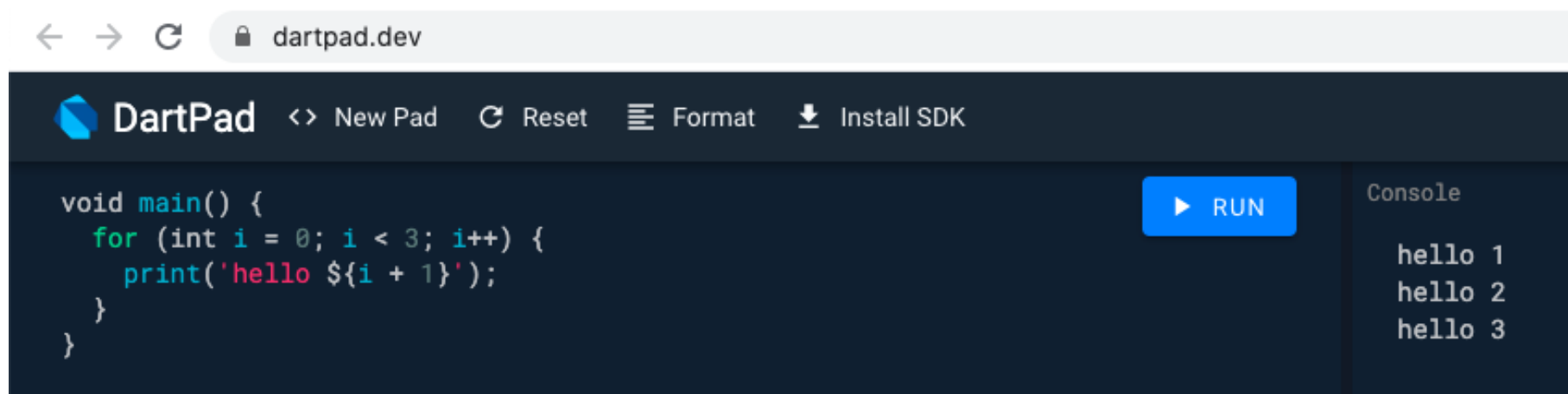
Algumas características

- Parecida com C, C++, Java, C#.
- É orientada a objetos.
- Fortemente tipada, mas não é necessário definir pois o Dart consegue inferir.
- O uso do caracter underline (_) precedendo um atributo ou método o torna privado.
- Dart pode ser compilada em ahead-of-time (AOT) e just-in-time (JIT). O primeiro significa que o código é compilado para ARM nativo (desempenho de aplicação nativa). O segundo permite a compilação direta no device (smartphone, por exemplo) com a aplicação em execução (isso possibilita o hot-reload).



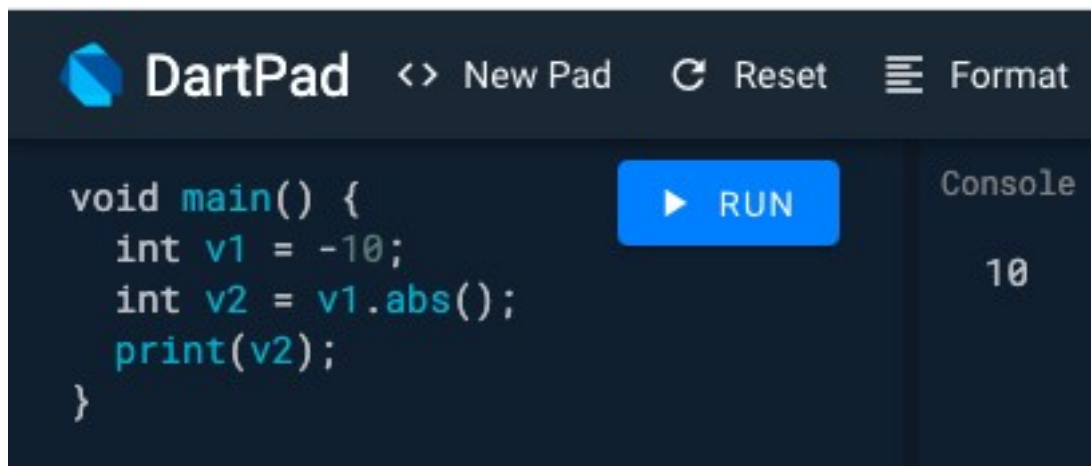
Ferramentas e apoio

- No link <https://dart.dev/guides/language/language-tour> é possível encontrar diversas orientações sobre a programação na linguagem Dart.
- No link <https://dartpad.dev/> podemos colocar códigos Dart e rodá-los para testar conceitos. A imagem abaixo apresenta essa página:



Tipos de dados

- Dart é uma linguagem estaticamente tipada, o quer dizer que uma vez que atribuímos um valor a uma variável, valores de outros tipos não poderão ser armazenados por essa mesma variável. Ou seja, uma variável usada como uma String, por exemplo, não se transformará num número para promover uma soma.
- Tipos primitivos, como existem em linguagens como C ou Java não existem em Dart. Em Dart “Tudo é objeto”. O exemplo abaixo ilustra uma variável do tipo **int** chamando o método **abs()**.



The screenshot shows the DartPad web IDE interface. At the top, there's a header with the DartPad logo and navigation links: '<> New Pad', 'Reset', and 'Format'. Below the header, the main area is split into two sections. The left section contains Dart code:

```
void main() {  
  int v1 = -10;  
  int v2 = v1.abs();  
  print(v2);  
}
```

. To the right of the code is a blue 'RUN' button. The right section, labeled 'Console', shows the output of the code: '10'.



Tipos de dados

num

- O primeiro tipo de dados que iremos tratar é o Number (**num**). Essa classe possui como subtipos **int** e **double**. O exemplo ao lado mostra o uso desses tipos:

```
void main(){
  num n1 = 10.4;

  // int i = n1;
  // Error: A value of type 'num'
  // can't be assigned to a variable of type 'int'.

  int i1 = n1.ceil();
  print(i1);

  // double d1 = n1;
  // Error: A value of type 'num'
  // can't be assigned to a variable
  // of type 'double'.

  double d1 = n1.toDouble();
  print(d1);

  n1 = -d1;
  if(n1.isNegative)
    print(n1);
  else
    print("0 número é positivo");
}
```

▶ Run

Console

11
10.4
-10.4

Documentation



Tipos de dados - String

- Em Dart Strings podem ser representadas por aspas simples ou aspas duplas.

```
1 void main(){  
2   String s = "Pedro";  
3   String s_upper = s.toUpperCase();  
4   String s_lower = s.toLowerCase();  
5   print("$s em caixa alta é $s_upper");  
6   print("$s em caixa baixa é $s_lower");  
7   int tam = s_upper.length;  
8   String s_aux = s_upper.substring(1, 4);  
9   print('O tamanho é $tam');  
10  print('A substring é $s_aux');  
11 }
```

▶ Run

Console

```
Pedro em caixa alta é PEDRO  
Pedro em caixa baixa é pedro  
O tamanho é 5  
A substring é EDR
```



Tipos de dados - bool

- Esse tipo de dados permite armazenar apenas dois valores: true (verdadeiro) e false (falso). O exemplo abaixo ilustra o uso do tipo bool:

```
1 void main(){  
2   bool eh_legal = false;  
3   if(!eh_legal)  
4     eh_legal = true;  
5   print("Agora ficou legal. Certo? $eh_legal");  
6 }  
7
```

▶ Run

Console

Agora ficou legal. Certo? true



Tipos de dados - dynamic

- Variáveis do tipo dynamic pode receber valores de todos os outros tipos e é possível mudar esses valores em tempo de execução.

```
1 void main(){
2   dynamic d = 10;
3   print("O valor de d é $d");
4   d = "Pedro";
5   print("O valor de d é $d");
6   d = 20;
7   d = d + 10;
8   print("O valor de d é $d");
9 }
```

Run

Console

O valor de d é 10
O valor de d é Pedro
O valor de d é 30



Tipos de dados - Function

- É possível ter variáveis e parâmetros do tipo Function (função).

```
1 void imprime(String s){
2   print('O valor de s é $s');
3 }
4
5 imprimeSoma(int n1, int n2, Function funcaoImpressao){
6   int soma = n1 + n2;
7   funcaoImpressao(soma.toString());
8 }
9 void main(){
10  Function funcao = imprime;
11  imprimeSoma(4, 5, funcao);
12
13  Function(int, int, Function) f2 = imprimeSoma;
14  f2(10, 20, funcao);
15 }
```

Run

Console

```
O valor de s é 9
O valor de s é 30
```



Tipo de dados - List

- São conjuntos de dados organizados em uma certa ordem (de índices, por exemplo).

```
1 void main(){
2   // List lista1 = List();
3   // Error: Can't use the default List constructor.
4   // List lista1 = List();
5   // Desabilitando Null Safety funciona
6
7   List lista1 = [];
8   lista1.add('A');
9   lista1.add('B');
10  print('$lista1');
11
12  for(String s in lista1)
13    print('\n 0 valor é $s');
14
15  for(int i=0; i<lista1.length; i++)
16    print('\n ${lista1[i]}');
17
18  List lista2 = [1, 2, 3];
19  print("\n $lista2");
20
21  List lista3 = lista2.map((valor) {
22    return valor * 10;
23  }).toList();
24  print("\n $lista3");
25
26  List lista4 = lista2.map((valor) => valor*20).toList();
27  print("\n $lista4");
28
29 }
```

Run

Console

[A, B]

0 valor é A

0 valor é B

A

B

[1, 2, 3]

[10, 20, 30]

[20, 40, 60]

Documentation



Tipos de dados - Map

- É uma estrutura de dados que organiza os elementos por chave/valor. Através de uma chave é possível recuperar um valor desejado.

```
void main() {  
    Map map = { "Fruta": "Manga",  
                "Carro": "Duster",  
                "Telefone": "Motorola"  
            };  
    print("O carro é ${map['Carro']}");  
    print('A fruta é ${map["Fruta"]}');  
    print('O telefone é ${map['Telefone']}');  
  
    map["Computador"] = "Dell";  
    map[3] = "Número 3";  
  
    print(map['Computador']);  
    print(map[3]);  
}
```

Console

```
O carro é Duster  
A fruta é Manga  
O telefone é Motorola  
Dell  
Número 3
```



Funções – Argumentos Opcionais

- Em Dart é possível passar argumentos opcionais como parâmetros:

A '?' indica que l2 pode ser nulo

```
1 double area(double l1, [double? l2]){  
2   if(l2 == null)  
3     return l1*l1;  
4   else  
5     return l1*l2;  
6 }  
7  
8 void main(){  
9   print("A área é ${area(3, 4)}");  
10  print("A área é ${area(3)}");  
11 }
```

▶ Run

Console

A área é 12

A área é 9



Funções – Argumento padrão

- É possível definir um valor padrão para parâmetros opcionais:

Aqui não foi necessária a '?' pois há um valor padrão (nesse caso 7), mas nesse caso a condição ficou inútil (f2 nunca será nulo)

```
1 double area(double l1, [double l2 = 7]){
2   if(l2 == null)
3     return l1*7;
4   else
5     return l1*l2;
6 }
7
8 void main(){
9   print("A área é ${area(3, 4)}");
10  print("A área é ${area(3)}");
11 }
```

The operand can't be null, so the condition is always false.

Run

Console

A área é 12
A área é 21



Funções – Argumentos nomeados

- É possível atribuir valores de parâmetros opcionais de forma nomeada (independente da ordem entre si, mas após os parâmetros obrigatórios):

```
1 double area(double l1, {double l2 = 7}){  
2     return l1*l2;  
3 }  
4  
5 void main(){  
6     print("A área é ${area(3, l2: 5)}");  
7     print("A área é ${area(3)}");  
8 }
```

▶ Run

Console

A área é 15

A área é 21



Funções – Argumentos nomeados - required

- É possível obrigar o programador a informar um parâmetro nomeado através da palavra **required**.

```
1 double area(double l1, {required double l2}){  
2     return l1*l2;  
3 }  
4  
5 void main(){  
6     print("A área é ${area(3, l2: 5)}");  
7  
8     // print("A área é ${area(7)}");  
9     //Error: Required named parameter 'l2' must be provided.  
10    // print("A área é ${area(7)}");  
11 }
```

Run

Console

A área é 15



Classes – Declaração e new

- Em Dart não é necessário, mas é permitido, utilizar o comando `new` para alocar um objeto.

O uso da palavra **late** indica que o atributo não poderá ser nulo mas que será atribuído de forma tardia, ou seja, fora do construtor



The screenshot shows a Dart IDE with a code editor on the left and a console on the right. The code defines a `Pessoa` class with `late` attributes for `nome` and `idade`. In the `main` function, two objects are created: `p1` using the default constructor and `p2` using the `new` keyword. Both objects have their `nome` and `idade` attributes set. The console output shows the printed information for both objects.

```
1 class Pessoa{
2   late String nome;
3   late int idade;
4 }
5
6 void main(){
7   Pessoa p1 = Pessoa();
8   Pessoa p2 = new Pessoa();
9
10  p1.nome = 'Pedro';
11  p1.idade = 30;
12
13  p2.nome = 'Ana';
14  p2.idade = 60;
15
16  print("A idade de ${p1.nome} é ${p1.idade}");
17  print("A idade de ${p2.nome} é ${p2.idade}");
18 }
```

Run

Console

A idade de Pedro é 30
A idade de Ana é 60



Classes – Métodos estáticos

- Dart possui métodos estáticos.

```
1 class Pessoa{
2   late String nome;
3   late int idade;
4
5   static void imprimirNomeClasse(){
6     print("Pessoa");
7   }
8 }
9
10 void main(){
11   Pessoa.imprimirNomeClasse();
12 }
```

Run

Console

Pessoa



Classes – construtor padrão

- Os construtores em Dart possuem sintaxe simplificada e resumida:

Aqui podemos retirar o **late** dos atributos pois há apenas um construtor e ele garante que os atributos terão um valor não nulo

```
1 class Pessoa{
2   String nome;
3   int idade;
4
5   Pessoa(this.nome, this.idade);
6 }
7
8 void main(){
9   Pessoa p1 = new Pessoa("Pedro", 30);
10  Pessoa p2 = Pessoa('Ana', 20);
11
12  print("${p1.nome} tem ${p1.idade}");
13  print("${p2.nome} tem ${p2.idade}");
14 }
```

Run

Console

Pedro tem 30
Ana tem 20



Classes – construtores nomeados

- Em Dart um construtor é definido como padrão e outros construtores devem ser nomeados:

```
1 class Pessoa{
2   late String nome;
3   int idade;
4
5   Pessoa(this.nome, this.idade);
6
7   // Não é permitido 2 construtores padrão
8   // Pessoa(this.nome);
9
10  Pessoa.anonima(this.idade){
11    this.nome = "Anonima";
12  }
13 }
14
15 void main(){
16   Pessoa p1 = new Pessoa("Pedro", 30);
17   Pessoa p2 = Pessoa('Ana', 20);
18   Pessoa p3 = Pessoa.anonima(50);
19
20   print("${p1.nome} tem ${p1.idade}");
21   print("${p2.nome} tem ${p2.idade}");
22   print("${p3.nome} tem ${p3.idade}");
23 }
```

Run

Console

Pedro tem 30
Ana tem 20
Anonima tem 50



Classes – construtores nomeados

- Dart permite a criação de construtores nomeados sem código. Na prática um construtor desse tipo apenas vai alocar o espaço necessário para o objeto (seria como um malloc em C, ou seja, não faria nenhum código de inicialização).



```
1 class Pessoa{
2   late String nome;
3
4   Pessoa.instance();
5   Pessoa.outrConstrutor();
6 }
7
8 void main(){
9   Pessoa p1 = Pessoa.instance();
10  p1.nome = "Pedro";
11  print("O valor do nome é ${p1.nome}");
12  Pessoa p2 = Pessoa.outrConstrutor();
13  p2.nome = "Ana";
14  print("O valor do nome é ${p2.nome}");
15 }
```

Run

Console

O valor do nome é Pedro
O valor do nome é Ana



gets e sets

- É possível definir métodos **get** e **set** para que funcionem como se fossem atributos.
- A sintaxe da seta (`=>`) é muito utilizada para métodos get e set. Sendo uma forma resumida para a definição de uma função ou método.

```
1 class Pessoa{
2   String nome = "";
3   int _idade = 0;
4
5   String get idade{
6     return '$_idade anos';
7   }
8
9   set idade (valor){
10    _idade = valor;
11  }
12 }
13
14 void main(){
15   Pessoa p1 = Pessoa();
16   p1.idade = 20;
17   print(p1.idade);
18   Pessoa p2 = Pessoa();
19   p2.idade = 30;
20   print(p2.idade);
21 }
```

Run

Console

20 anos
30 anos

```
1 class Pessoa{
2   String nome = "";
3   int _idade = 0;
4
5   String get idade => '$_idade anos';
6   set idade (valor) => _idade = valor;
7 }
```

Run



Atributos, métodos e classes privados

- Para que um atributo, método ou classe seja privado basta colocar '_' na frente do atributo ou classe.
- Em Dart algo ser privado significa que é válido apenas no contexto do arquivo onde foi criado.

```
class _A{  
  int? _atributo_privado;  
  void _metodo_privado(){  
    return;  
  }  
}  
  
class A{  
  int? _atributo_privado;  
  void metodo_publico(){ return; }  
  void _metodo_privado(){  
    _A local_a = _A();  
    local_a._metodo_privado();  
    local_a._atributo_privado = 10;  
    return;  
  }  
}
```

```
import 'a.dart';  
  
class B{  
  void usandoAe_A(){  
    A a = A();  
    a.metodo_publico();  
    a._metodo_privado();  
    a._atributo_privado = 10;  
  
    _A a2 = _A();  
  }  
}
```



Herança

```
1 class Pessoa{
2   late String nome;
3   int idade;
4
5   Pessoa(this.nome, this.idade);
6
7   Pessoa.anonima(this.idade){
8     this.nome = "Anônimo";
9   }
10 }
11
12 class PessoaFisica extends Pessoa{
13   late String cpf;
14
15   PessoaFisica(nome, idade, this.cpf):super(nome, idade);
16
17   PessoaFisica.anonima(idade, {cpf = "000"}):super.anonima(idade){
18     this.cpf = cpf;
19   }
20 }
21
22 void main(){
23   PessoaFisica p1 = PessoaFisica("Pedro", 30, "999.999.999-99");
24   PessoaFisica p2 = PessoaFisica.anonima(30, cpf: "xxx");
25   PessoaFisica p3 = PessoaFisica.anonima(40);
26
27   print('${p1.nome} tem ${p1.idade} e cpf ${p1.cpf}');
28   print('${p2.nome} tem ${p2.idade} e cpf ${p2.cpf}');
29   print('${p3.nome} tem ${p3.idade} e cpf ${p3.cpf}');
30 }
```

▶ Run

Console

Pedro tem 30 e cpf 999.999.999-99
Anônimo tem 30 e cpf xxx
Anônimo tem 40 e cpf 000



Sobrescrita - toString

```
class Pessoa{
  String nome;
  int idade;

  Pessoa(this.nome, this.idade);

  String toString(){
    return "Nome: $nome, Idade: $idade";
  }
}
class PessoaFisica extends Pessoa{
  String cpf;

  PessoaFisica(nome, idade, this.cpf):super(nome, idade);

  String toString(){
    return "CPF: $cpf " + super.toString();
  }
}

void main() {
  PessoaFisica p1 = PessoaFisica( "Pedro", 30, "999.999.999-99");
  print(p1);
}
```

▶ RUN

Console

CPF: 999.999.999-99 Nome: Pedro, Idade: 30



Mixin

- Dart não possui suporte a herança múltipla. Para resolver essa necessidade Dart utiliza Mixins.
- Mixins permitem adicionar métodos de outras classes sem usar herança.

```
1 class BordaSuperior{
2   String texto = "Borda Superior";
3   void imprimirBordaSuperior(){
4     print("=====$texto====");
5   }
6 }
7
8 class BordaInferior{
9   String texto = "Borda Inferior";
10  void imprimirBordaInferior(){
11    print("*****$texto*****");
12  }
13 }
14
15 class Botao1 with BordaSuperior, BordaInferior{
16   void desenharBotao(){
17     this.imprimirBordaSuperior();
18     print(this.texto);
19     this.imprimirBordaInferior();
20   }
21 }
22
23 class Botao2 with BordaInferior, BordaSuperior {
24   void desenharBotao(){
25     this.imprimirBordaSuperior();
26     print(this.texto);
27     this.imprimirBordaInferior();
28   }
29 }
30
31 void main(){
32   Botao1 b1 = Botao1();
33   b1.desenharBotao();
34   print("\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
35   Botao2 b2 = Botao2();
36   b2.desenharBotao();
37
38 }
39
```

Console

```
===== Borda Inferior =====
Borda Inferior
***** Borda Inferior *****

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

===== Borda Superior =====
Borda Superior
***** Borda Superior *****
```



Mixin

- Note que, se uma classe usa outras duas e essas possuem o mesmo atributo ou método, fica valendo o atributo ou método da última selecionada.

```
1 class BordaSuperior{
2   String texto = "Borda Superior";
3   void imprimirBordaSuperior(){
4     print("=====$texto====");
5   }
6 }
7
8 class BordaInferior{
9   String texto = "Borda Inferior";
10  void imprimirBordaInferior(){
11    print("*****$texto*****");
12  }
13 }
14
15 class Botao1 with BordaSuperior, BordaInferior{
16   void desenharBotao(){
17     this.imprimirBordaSuperior();
18     print(this.texto);
19     this.imprimirBordaInferior();
20   }
21 }
22
23 class Botao2 with BordaInferior, BordaSuperior{
24   void desenharBotao(){
25     this.imprimirBordaSuperior();
26     print(this.texto);
27     this.imprimirBordaInferior();
28   }
29 }
```

```
==== Borda Inferior ====
Borda Inferior
***** Borda Inferior *****

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

==== Borda Superior ====
Borda Superior
***** Borda Superior *****
```

```
void main(){
  Botao1 b1 = Botao1();
  b1.desenharBotao();
  print("\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
  Botao2 b2 = Botao2();
  b2.desenharBotao();
}
```

Mixin – exemplo com métodos iguais

```
1 class BordaSuperior{
2   String texto = "Borda Superior";
3   void imprimirBorda(){
4     print("==== $texto =====");
5   }
6 }
7
8 class BordaInferior{
9   String texto = "Borda Inferior";
10  void imprimirBorda(){
11    print("***** $texto *****");
12  }
13 }
14
15 class Botao1 with BordaSuperior, BordaInferior{
16   void desenharBotao(){
17     this.imprimirBorda();
18     print(this.texto);
19     this.imprimirBorda();
20   }
21 }
22
23 class Botao2 with BordaInferior, BordaSuperior {
24   void desenharBotao(){
25     this.imprimirBorda();
26     print(this.texto);
27     this.imprimirBorda();
28   }
29 }
30
31 void main(){
32   Botao1 b1 = Botao1();
33   b1.desenharBotao();
34   print("\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
35   Botao2 b2 = Botao2();
36   b2.desenharBotao();
37 }
38 }
39
```

Console

```
***** Borda Inferior *****
Borda Inferior
***** Borda Inferior *****

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

===== Borda Superior =====
Borda Superior
===== Borda Superior =====
```



const x final

- A palavra **const** não permite a alteração da variável e o valor é colocado em tempo de compilação, ou seja, deve ser um constante (não pode ser advindo de uma função que precisa ser calculada).
- A palavra **final** não permite a alteração de uma variável e o valor é colocado em tempo de execução, ou seja, pode ser calculado por uma função mas uma vez atribuído não poderá mais ser modificado.
- O exemplo ao lado ilustra essa situação:

```
1 void main() {  
2   // Aqui é a mesma coisa  
3   const int n1 = 10;  
4   final int n2 = 20;  
5  
6   //n1 = 15; Não funciona  
7   //n2 = 30; Não funciona  
8  
9   // const int n3 = n1.abs(); Não funciona  
10  final int n4 = n1.abs();  
11  
12  print(n1);  
13  print(n2);  
14  // print(n3);  
15  print(n4);  
16 }
```



Construtor const

- Construtores **const** são construtores cujas classes são compostas apenas por atributos final e não possuem corpo/código interno (nenhuma instrução pode ser executada). Os objetos na prática são “constantes em tempo de compilação”. Isso é especialmente útil em Flutter para evitar recriar objetos que não mudam o estado (utiliza-se o mesmo objeto já que ele é um constante).

```
1 class Pessoa{
2   final String nome;
3   final int idade;
4
5   const Pessoa(this.nome, this.idade);
6 }
7
8 void main() {
9
10  Pessoa p1 = const Pessoa("Pedro", 10);
11  Pessoa p2 = const Pessoa("Pedro", 10);
12  Pessoa p3 = const Pessoa("Pedro", 20);
13
14  Pessoa p4 = Pessoa("Pedro", 10);
15  Pessoa p5 = Pessoa("Pedro", 10);
16  Pessoa p6 = Pessoa("Pedro", 20);
17
18  if(p1 == p2)
19    print("p1 e p2 são o mesmo objeto");
20  if(p1 == p4)
21    print("p1 e p4 são o mesmo objeto");
22  if(p4 == p5)
23    print("p4 e p5 são o mesmo objeto");
24  if(p3 == p6)
25    print("p3 e p6 são o mesmo objeto");
26 }
```

Console

p1 e p2 são o mesmo objeto

Documentation

(new) Pessoa Pessoa(Str:



Operador ! x ?

- Quando trabalhamos com objetos, um objeto pode acessar um campo ou método de um outro objeto interno. Entretanto o que deve acontecer se esse objeto interno for nulo?
- É isso que os operadores ! e ? tratam.
- O operador “!” verificará se há nulo, se houver, irá disparar uma exceção.
- O operador “?” irá retornar nulo (null).
- No próximos próximos slides veremos essa diferença através de um exemplo:

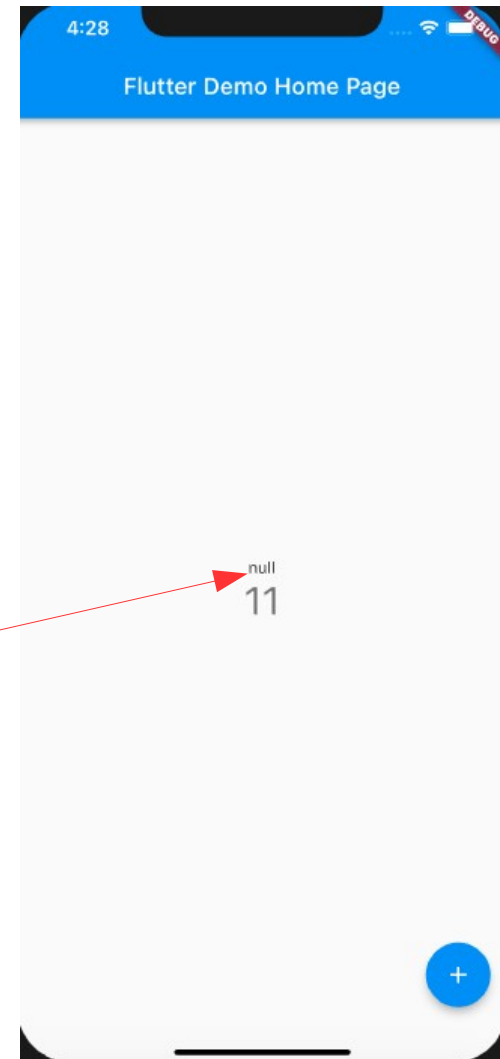


Operador ?

Suponha carro com
valor null em p1

```
Text(  
  '${p1.obterNomeCarro()}',  
), // Text
```

```
class Carro{  
  String? nome;  
  String? modelo;  
}  
  
class Pessoa{  
  String nome;  
  int idade;  
  Carro? carro;  
  
  Pessoa(this.nome, this.idade);  
  
  String? obterNomeCarro(){  
    return this.carro?.nome;  
  }  
}
```

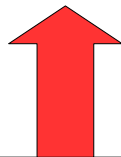


Operador !

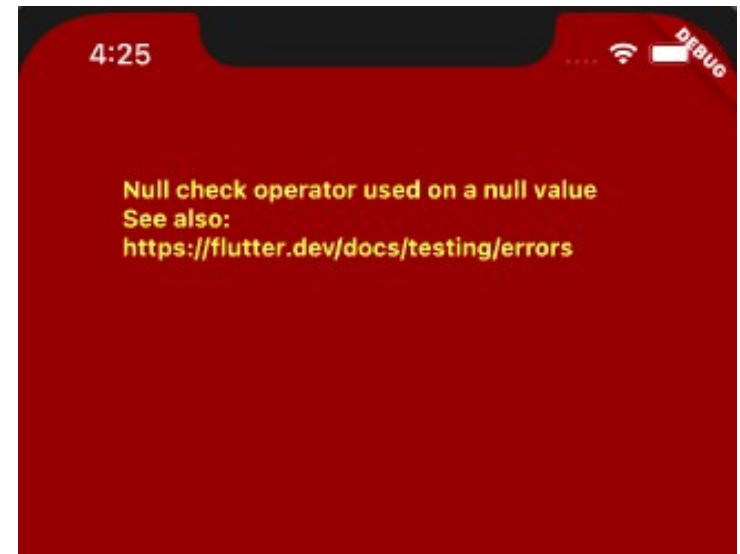
Suponha carro com
valor null em p1

```
class Carro{  
  String? nome;  
  String? modelo;  
}
```

```
class Pessoa{  
  String nome;  
  int idade;  
  Carro? carro;  
  
  Pessoa(this.nome, this.idade);  
  
  String? obterNomeCarro(){  
    return this.carro!.nome;  
  }  
}
```



```
Text(  
  '${p1.obterNomeCarro()}',  
), // Text
```



Dúvidas?

