

1 Introdução

Dart é uma linguagem de programação criada pelo Google em 2011. A sua versão 1.0 foi liberada em 2013. Vejamos algumas de suas principais características.

Dart é uma linguagem compilada e interpretada.

Código Dart pode ser compilado para:

- código de máquina (Linux, MacOS, Windows).
- Web (Javascript e WebAssembly (baixo nível como assembly, desempenho quase nativo, integrável com Javascript)).
 - Mobile (Android e iOS).

Compiladores **Just in time** e **Ahead of time**

Seu **compilador just in time** tem as seguintes características:

- Código compilado apenas no momento em que for executado.
- Compilação sob demanda (somente arquivos acessados são compilados).
- Desenvolvedor não precisa esperar que a aplicação seja compilada por completo para testá-la.
- A cada nova alteração de código, não há a necessidade de recompilar a aplicação.
 - Viabiliza o hot reload.
- Se fossemos fazer o deploy da aplicação, seria necessário incluir o código fonte original e o compilador (que, por si só, pode ser maior que a aplicação).
 - Próprio para desenvolvimento e testes.

Seu compilador **ahead of time**:

- Aplicação compilada por completo, antes de ser colocada em execução.
- Código fonte original e compilador não são incluídos no produto final.
 - Aplicação mais “leve” (menos bytes).
- Próprio para deploy em produção.

Supporte aos seguintes paradigmas: Orientação a Objetos, Funcional, Imperativo e Reflexivo (inspeção da estrutura do programa em tempo de execução)

Seu sistema de tipos é gradual. Uma espécie de mistura de tipagem estática e dinâmica que veremos em breve. Também inclui inferência de tipos.

2 Desenvolvimento

2.1 Documentação oficial A documentação oficial de Dart, semelhante ao Javadoc do Java, pode ser encontrada em

<https://api.dart.dev/stable>

As principais classes, como String, int, bool etc se encontram no pacote dart:core. Veja.

Dart > dart:core library

dart:core library

Built-in types, collections, and other core functionality for every Dart program.

This library is automatically imported.

Some classes in this library, such as `String` and `num`, support Dart's built-in data types. Other classes, such as `List` and `Map`, provide data structures for managing collections of objects. And still other classes represent commonly used types of data such as URLs, dates and times, and errors.

Numbers and booleans

`int` and `double` provide support for Dart's built-in numerical data types: integers and double-precision floating point numbers, respectively. An object of type `bool` is either true or false. Variables of these types can be constructed from literals:

```
int meaningOfLife = 42;
double valueOfPi = 3.141592;
bool visible      = true;
```

Strings and regular expressions

A `String` is immutable and represents a sequence of characters.

```
String shakespeareQuote = "All the world's a stage, ...";
```

`StringBuffer` provides a way to construct strings efficiently.

```
var moreShakespeare = StringBuffer();
moreShakespeare.write("And all the men and women ");
moreShakespeare.write("merely players; ...");
```

The `String` and `StringBuffer` classes implement string splitting, concatenation, and other string manipulation features.

CLASSES

- BigInt
- bool
- Comparable
- DateTime
- Deprecated
- double
- Duration
- Enum
- Expando
- Finalizer
- Function
- Future
- int
- Invocation
- Iterable
- Iterator
- List
- Map
- MapEntry
- Match
- Null
- num
- Object
- Pattern

2.2 DartPad - Editor online O ambiente Dart inclui o DartPad, um editor on-line que viabiliza a execução de testes rápidos, particularmente útil para quem está começando. Ele pode ser visitado em

<https://dartpad.dartlang.org/>

Ao visitar o DartPad, você verá **três regiões e um botão para execução do código**.

- **A região principal** em que escrevemos código.
- **O Console**. Ele exibe o resultado do programa após clicarmos no botão para execução.
- **Documentation**. Exibe a documentação do item sobre o qual clicarmos.

O exemplo a seguir mostra um trecho de código que veio pré-definido no momento em que abrimos o DartPad. Nele, já clicamos no botão para execução e, a seguir, clicamos sobre a função print, a fim de exibir a sua documentação.

The screenshot shows the DartPad interface with the following details:

- Top Bar:** DartPad, New Pad, Reset, Format, Install SDK.
- Code Area:** Shows the following Dart code:

```
1 void main() {
2   for (int i = 0; i < 5; i++) {
3     print('Hello ${i + 1}');
4   }
5 }
```

A tooltip says: "Clique sobre o nome da função print para ver a sua documentação".
- Run Button:** A blue button labeled "Run" with a play icon.
- Console Area:** Displays the output of the code execution:

```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

A tooltip says: "Clique aqui para apagar o texto do console." and "O resultado do programa aparece aqui, no console."
- Documentation Area:** Shows the documentation for the `print` function:

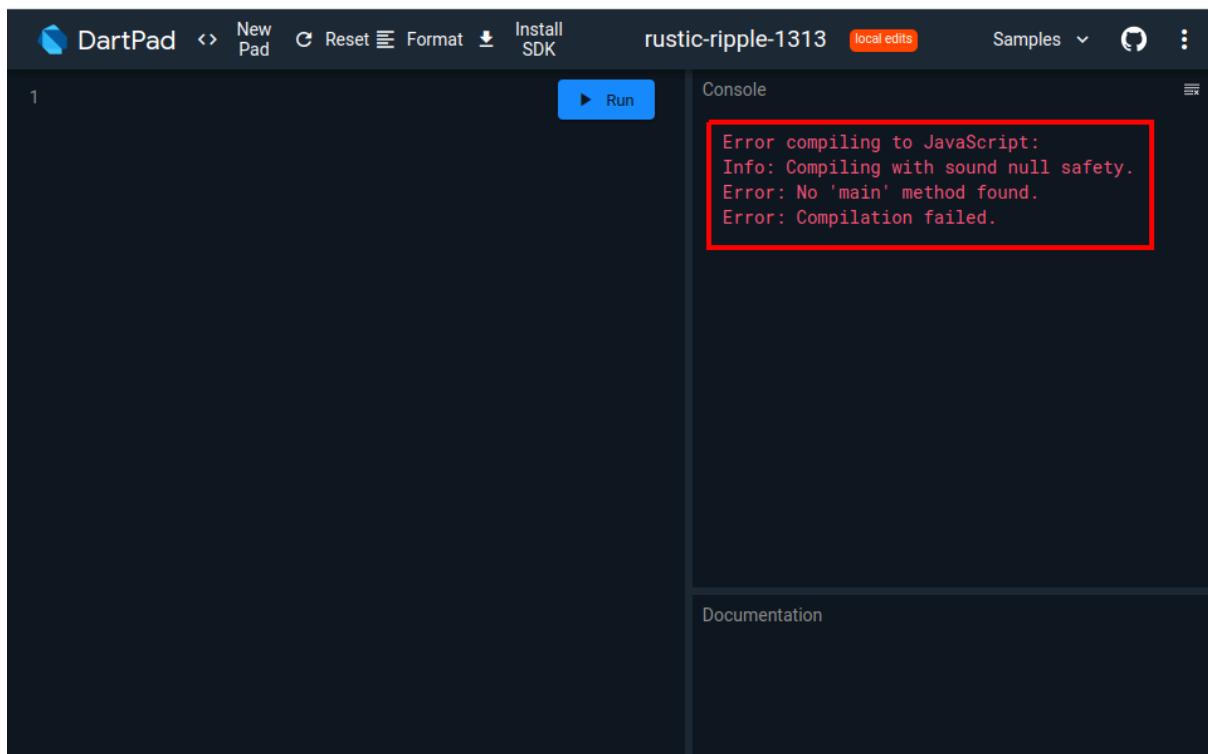
```
void print(Object? object)
```

Prints a string representation of the object to the console.

Open library docs ↗

A tooltip says: "Depois de clicar no nome da função print, você verá a sua doc aqui".

Para começarmos nossos testes, apague todo o código pré definido. Observe que **não é possível executar um programa que não possua uma função principal**.



2.3 Hello, World com Dart Para escrever um programa “Hello, World” com Dart, o primeiro passo é definir uma função principal. Veja.

```
void main() {  
}
```

A função **print** é responsável por exibir conteúdo no console. Strings são delimitadas por aspas simples ou duplas. Não se esqueça de incluir o ponto-e-vírgula no final.

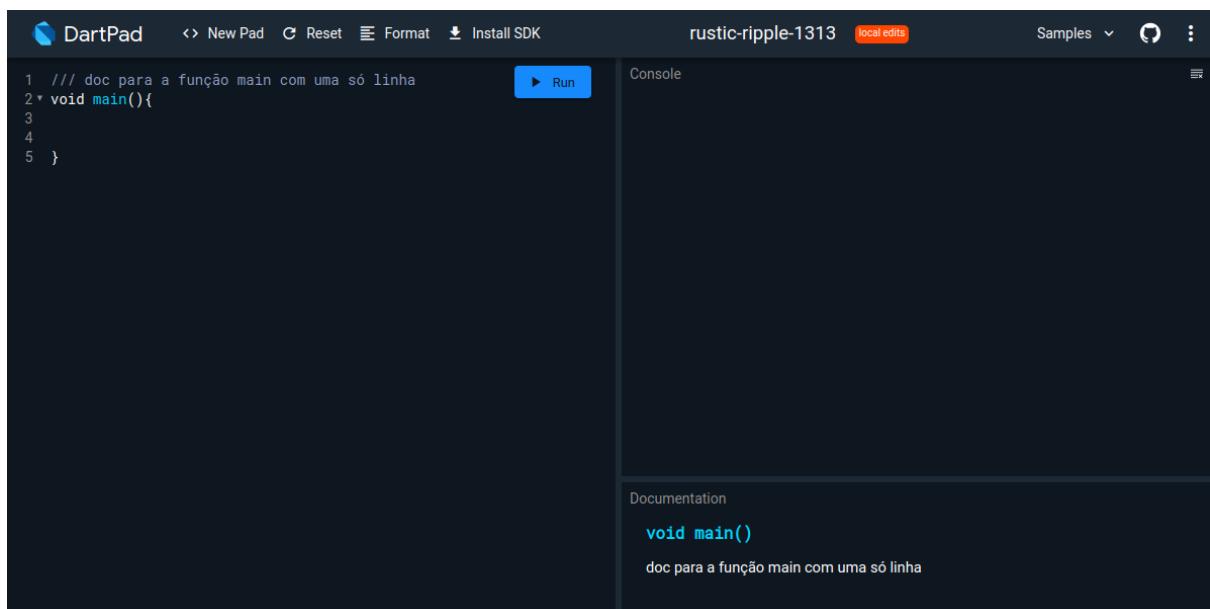
```
void main() {  
    print('Hello, World');  
}
```

2.4 Comentários em Dart Há dois tipos de comentários em Dart. Aqueles de uma única linha e aqueles de múltiplas linhas. Veja.

```
void main() {
  //sou um comentário de uma única linha

  /**
   *
   * Sou um comentário de múltiplas linhas
   */
}
```

Há também os **comentários utilizados para gerar documentação**. Se precisar de uma única linha, use barras triplas (///). Se precisar de múltiplas linhas, use / ** para começar e */ para terminar. Lembre-se de clicar no nome da função main para ver a documentação no DartPad. Veja um exemplo de comentário de documentação de única linha.



Agora um exemplo de comentário de documentação de múltiplas linhas.

The screenshot shows the DartPad interface with a dark theme. In the code editor, there is a multi-line documentation comment above a `void main()` function:

```

1 * /**
2  * Doc para a função main
3  * usando
4  * múltiplas
5  * linhas
6 *
7 */
8 void main(){
9
10}

```

A blue "Run" button is located at the top right of the code editor. To the right of the editor is a "Console" panel which is currently empty. Below the code editor is a "Documentation" panel. It contains the following information:

- void main()**
- Doc para a função main usando múltiplas linhas

Se desejarmos, podemos fazer referência a um elemento de código (como um método, classe etc) colocando seu nome dentro de colchetes. Veja a diferença.

The screenshot shows the DartPad interface with a dark theme. In the code editor, there is a single-line documentation comment enclosed in brackets above a `void main()` function:

```

1 * /**
2  * O que acontece
3  * se o nome da função [main]
4  * for colocando entre colchetes?
5 */
6 void main(){
7
8
9 }
10

```

A blue "Run" button is located at the top right of the code editor. To the right of the editor is a "Console" panel which is currently empty. Below the code editor is a "Documentation" panel. It contains the following information:

- void main()**
- O que acontece se o nome da função `main` for colocando entre colchetes?

No DartPad, o nome do elemento fica destacado com outra cor. Se eventualmente gerarmos essa documentação para publicação, ele se torna um link para o elemento, viabilizando a navegação entre elementos, assim como acontece na documentação oficial de Dart.

2.6 Tipos de dados, variáveis e inicialização

Os tipos de dados em Dart são os seguintes.

Em Dart

Tipo	Observação	Exemplo
int	Números inteiros (64 bits).	int a = 2;
double	Números reais (64 bits)	double d = 2.5;
num	Tipo numérico genérico. Admite a atribuição de int ou double. num é superclasse de int e de double.	num n1 = 2; num n2 = 2.5;
String	Uma sequência de zero ou mais caracteres representados em UTF-16. Pode ser delimitada por aspas simples ou duplas . Podemos criar strings de múltiplas linhas usando aspas simples ou duplas “triplas”. Podemos criar strings “raw” precedendo o seu valor com a letra r. Assim, caracteres de escape são ignorados.	String s1 = 'abc'; String s2 = "abc"; String s3 = "uma string de múltiplas linhas com aspas simples"; String s4 = """ uma string de múltiplas linhas com aspas duplas"""; String s5 = r'O \n serve para pular uma linha';
bool	Valores booleanos	bool deMaior = false; bool alto = true;

Nota. Há outros tipos em Dart, como Records, Lists, Sets, Maps, Runes e Symbols, os quais estudaremos em outro momento. Veja mais em <https://dart.dev/language/built-in-types>.

Os identificadores (nomes de variáveis, métodos, classes etc) devem estar de acordo com as seguintes regras.

- Podem incluir letras maiúsculas.
- Podem incluir letras minúsculas.
- Podem incluir dígitos.
- Podem incluir o símbolo underscore (_).
- Podem incluir o símbolo \$.
- Não podem começar com dígito.

Nota. É comum utilizar a notação **camel case** em programas Dart.

Veja um programa que faz uso de alguns dos tipos mencionados.

```
void main(){
    String nome = "João"; //aspas duplas
    String sobrenome = 'Silva'; //aspas simples
    String endereço = '''
        Rua B,
        número 1234, Vila J''' //aspas simples triplas
    bool deMaior = false;
    int idade = 17;
    num peso = 80.5;
    double altura = 1.82;
    //string raw
    String comoPularLinha = r"Pule uma linha com \n";
    print (nome);
    print(sobrenome);
    print(endereço);
    print(deMaior);
    print(idade);
    print(peso);
    print(altura);
    print(comoPularLinha);
}
```

Se desejar, você pode exibir o tipo de uma variável com **runtimeType**. Veja alguns exemplos.

```

void main() {
    int a = 2;
    double b = 3.5;
    String c = "abc";
    bool d = true;
    num e = 2;
    num f = 2.2;

    print(a.runtimeType);
    print(b.runtimeType);
    print(c.runtimeType);
    print(d.runtimeType);
    print(e.runtimeType);
    print(f.runtimeType);

    //também dá para exibir o tipo de literais
    print(2.runtimeType);
    print(true.runtimeType);
}

```

2.7 Concatenação, interpolação e multiplicação de Strings Em Dart, concatenamos Strings utilizando o operador `+`. Veja.

```

void main(){
    String nome = "João";
    String sobrenome = 'Silva';
    String endereco = '''
        Rua B,
        número 1234, Vila J'''';
    bool deMaior = false;
    int idade = 17;
    num peso = 80.5;
    double altura = 1.82;
    print ('Me chamo ' + nome);
}

```

Observe que **variáveis de tipos diferentes não podem ser concatenadas com +**. Entretanto, podemos resolver isso obtendo a representação textual do objeto que desejamos envolver na concatenação. Cabe ao método **toString** produzi-las. Veja.

```
void main() {
    String nome = "João";
    String sobrenome = 'Silva';
    String endereco = '''
        Rua B,
        número 1234, Vila J'''';
    bool deMaior = false;
    int idade = 17;
    num peso = 80.5;
    double altura = 1.82;
    //erro, concatenando string com int, não pode
    //print ('Minha idade é ' + idade);
    //mas podemos concatenar string com string, convertendo antes
    print ('Minha idade é ' + idade.toString());
}
```

O efeito da concatenação pode ser obtido utilizando-se a **interpolação**. Em geral, o código fica mais fácil de entender, além de permitir o uso de variáveis de outros tipos além de Strings. O símbolo utilizado para a interpolação é o **\$**. A notação é **\$variavel** ou **\${expressao}**.

```

void main(){
    String nome = "João";
    String sobrenome = 'Silva';
    String endereco =
        Rua B,
        número 1234, Vila J'''';
    bool deMaior = false;
    int idade = 17;
    num peso = 80.5;
    double altura = 1.82;
    //interpolação com $variavel
    print ('Me chamo $nome');
    //interpolação com ${expressão}
    print ('Meu sobrenome é ${sobrenome}');
    //com int e double
    print ("Tenho $idade anos e $peso kg");
}

```

Observe que há um espaço em branco entre o peso e a sigla **kg**. Talvez queiramos deixar de exibi-lo. Também é possível que desejemos avaliar uma expressão como uma soma de dois valores. Para tal, podemos usar **\${expressao}**. Veja.

```

void main(){
    String nome = "João";
    String sobrenome = 'Silva';
    String endereco =
        '''Rua B,
        número 1234, Vila J'''';
    bool deMaior = false;
    int idade = 17;
    num peso = 80.5;
    double altura = 1.82;
    print ('Me chamo $nome');
    print ('Meu sobrenome é ${sobrenome}');
    //com int e double
    print ("Tenho $idade anos e ${peso} kg");
    print ('Ano que vem terei ${idade + 1} anos.');
}

```

Podemos multiplicar uma string por um número inteiro N. O resultado será a concatenação da string N vezes.

```
void main() {
    var letra = 'x';
    print(letra * 10);
}
```

2.8 Conversão entre tipos Podemos utilizar os métodos **parse** e **toString** para converter de String para número e de número para String, respectivamente. Veja.

```
void main(){
    //de string para int
    String idadeTextual = "25";
    int idade = int.parse(idadeTextual);
    print(idade);

    //de string para double
    String pesoTextual = '85.2';
    double peso = double.parse(pesoTextual);
    print(peso);

    //de string para num
    String alturaTextual = '1.8';
    num altura = num.parse(alturaTextual);
    print(altura);

    String logradouro = "Rua B";
    int numero = 325;
    //não podemos concatenar string com int, lembra?
    //print(logradouro + ', número: ' + numero);
    //mas podemos converter para String antes
    print(logradouro + ', número: ' + numero.toString());
}
```

Não podemos atribuir uma variável int a uma variável double, embora possamos atribuir um valor inteiro literal a um double. Observe. **Isso acontece pois ambas as classes int e double herdam da classe num. Porém, a classe int não herda da classe double.** Ou seja, a classe

int não passa no teste É-UM double. Veremos mais sobre isso quando estudarmos sobre herança.

```

void main() {
    //aqui tudo bem, 2 é um literal inteiro
    //promoção implícita feita pelo compilador
    double d1 = 2;
    int i1 = 2;
    //erro em tempo de compilação
    //não dá pois i1 é um inteiro e d2 é um double
    //a classe int não passa no teste é um double
    double d2 = i1;
    //podemos resolver assim
    double d2 = i1.toDouble();
    print(d2);
}

```

Observe que o tipo de coerção a seguir não existe em Dart. Se desejarmos obter a parte inteira de um número real, podemos usar métodos apropriados. Observe.

```

void main() {
    double a = 1.2;
    //erro
    //int b = (int) a;
    //mas podemos usar métodos

    //arredonda
    int b = a.round();
    print(b);

    //teto
    b = a.ceil();
    print(b);
    //chão
    b = a.floor();
    print(b);
}

```

Nota. Mais adiante, estudaremos sobre o operador **as**.

2.9 Operadores aritméticos

Vejamos os operadores aritméticos da linguagem Dart.

Operador	Observação	Exemplo
+	soma (ou concatenação de strings)	int a = 5 + 2; // 7 String s = "a" + "bc" //abc;
-	Subtração	int a = 5 - 2; //3
*	multiplicação	int a = 5 * 2; //10
/	divisão real	double a = 5 / 2; // 2.5 //int a = 5 / 2; é um erro, pois 5 / 2 é double
~/	divisão inteira	int a = 5 ~/ 2 //2; //double a = 5 ~/ 2; é um erro, pois int não herda de double
%	resto de divisão inteira	int a = 5 % 2; // 1

Dart possui os seguintes **operadores aritméticos compostos**.

Operador	Observação	Exemplo
+=	soma e atribuição	int a = 2; a += 2; //4
-=	subtração e atribuição	int a = 2; a -= 2; // 0;
*=	multiplicação e atribuição	int a = 2; a *= 2; //4
/=	divisão real e atribuição	double a = 5; a /= 2; //2.5
~/=	divisão inteira e atribuição	int a = 5; a ~/= 2; // 2
%=	resto de divisão inteira e atribuição	int a = 5; a %= 2; // 1

Assim como em outras linguagens, também temos os **operadores de incremento e decremento**.

Operador	Observação	Exemplo
<code>++</code>	Soma e incremento (pós)	<code>int a = 2; int b = a++; //pós incremento, b vale 2</code>
<code>++</code>	Soma e incremento (pré)	<code>int a = 2; int b = ++a; // pré incremento, b vale 3</code>
<code>--</code>	Subtração e decremento (pós)	<code>int a = 2; int b = a--; //pós decremento, b vale 2</code>
<code>--</code>	Subtração e decremento(pré)	<code>int a = 2; int b = --a; //pré decremento, b vale 1</code>

Relembremos a diferença entre os operadores de pré e pós incremento.

- Quando a operação de incremento é a única existente no contexto, tanto faz usar pré ou pós incremento.
- Quando o contexto possui uma operação qualquer e uma operação de pré incremento, o pré incremento acontece antes da outra operação.
- Quando o contexto possui uma operação qualquer e uma operação de pós incremento, o pós incremento acontece depois da outra operação.

```
void main() {
    int a = 2;
    //somente pré incremento, a vale 3
    ++a;
    print(a);

    //somente pós incremento, a vale 4
    a++;
    print(a);

    //duas operações no contexto
    //print e pré incremento
    //primeiro incrementa e depois exibe
    //vai exibir 5
    print(++a);

    //duas operações no contexto
    //print e pós incremento
    //primeiro exibe e depois incrementa
    //vai exibir 5
    print(a++);

    //agora a vale 6
    print(a);
}
```

A lógica é a mesma para os operadores de pré e pós decremento.

2.10 Operadores relacionais Os operadores relacionais de Dart são os seguintes. **Dê especial atenção aos operadores == e != quando eles operam sobre strings.**

Operador	Observação	Exemplo
==	comparação por igualdade	$2 == 2 // \text{true}$ $2 == 3 // \text{false}$ $\text{"abc"} == \text{"abc"} // \text{true} (\text{sim}, \text{strings são comparadas assim})$ $\text{"abc"} == \text{"ab"} // \text{false}$ $2 == 2.0 // \text{true}$
!=	comparação por diferença	$2 != 2 // \text{false}$ $2 != 3 // \text{true}$ $\text{"abc"} != \text{"abc"} // \text{false}$ $\text{"abc"} != \text{"ab"} // \text{true}$ $2 != 2.0 // \text{false}$
<	menor	$2 < 2 // \text{false}$ $2 < 3 // \text{true}$ $3 < 2 // \text{false}$ $2 < 2.0 // \text{false}$
>	maior	$2 > 2 // \text{false}$ $2 > 3 // \text{false}$ $3 > 2 // \text{true}$ $2 > 2.0 // \text{false}$
<=	menor ou igual	$2 <= 2 // \text{true}$ $2 <= 3 // \text{true}$ $3 <= 2 // \text{false}$ $2 <= 2.0 // \text{true}$
>=	maior ou igual	$2 >= 2 // \text{true}$ $2 >= 3 // \text{false}$ $3 >= 2 // \text{true}$ $2 >= 2.0 // \text{true}$

Atenção. Strings não podem ser comparadas utilizando-se os operadores <, >, <= e >=.

2.11 Operadores lógicos Os operadores lógicos de Dart são os seguintes.

Ou lógico: ||

E lógico: &&

Negação: !

Eles funcionam como esperado, de acordo com a seguinte tabela-verdade. Considere que A e B são duas expressões booleanas.

A	B	!A	!B	A B	A && B
V	V	F	F	V	V
V	F	F	V	V	F
F	V	V	F	V	F
F	F	V	V	F	F

2.12 Operador ternário Dart possui um **operador ternário** que opera como uma espécie de if/else de uma linha só. Veja um exemplo.

```
void main() {
    bool vaiChover = true;
    String levarGuardaChuva = vaiChover ? "SIM!" : "NÃO";
    print (levarGuardaChuva);
    int idade = 17;
    String podeDirigir = idade >= 18 ? "SIM!" : "NÃO";
    print(podeDirigir);
}
```

2.13 O sistema de tipos de Dart é gradual. Uma espécie de mistura de tipagem estática e dinâmica.

Uma **linguagem estaticamente tipada** é aquela cuja verificação de tipo é feita pelo compilador. Exemplos de linguagens estaticamente tipadas são **Java, Kotlin, C++ e Scala**.

Uma **linguagem dinamicamente tipada** é aquela cuja verificação de tipo é feita em tempo de execução. Exemplos de linguagens dinamicamente tipadas são **Javascript e Python**.

Uma **linguagem com sistema de tipos gradual** possui **tipagem estática e dinâmica**. Exemplos de linguagens assim são **TypeScript, C# e Dart**.

Veja os exemplos a seguir.

```
void main() {  
    //variável estaticamente tipada (int)  
    //em tempo de compilação e em tempo de execução é int  
    int a = 2;  
    print(a.runtimeType); //int  
    //erro em tempo de compilação, tentativa de chamar um método que a  
    //classe int não possui  
    //a.indexOf("b");  
    //erro, atribuição de string a variável de tipo int  
    //a = "abc";  
    //variável dinamicamente tipada  
    //em tempo de compilação não há checagem  
    //em tempo de execução o tipo é int  
    dynamic b = 2;  
    //erro somente em tempo de execução  
    //b.indexOf("2");  
    print(b.runtimeType); // int  
    b = "abc";  
    print(b.runtimeType); // String  
}
```

Dart também inclui **inferência de tipos**. Ela entra em cena quando utilizamos a palavra **var**. Veja.

```
void main(){
  //é claro que "João" é uma String, não precisamos dizer isso
  //explicitamente
  //String nome = "João";
  var nome = "João";
  print(nome.runtimeType);
  //observe que uma vez que o tipo tenha sido inferido, não vale
  tentar mudar
  //erro, nome é String
  //nome = 2;
  //o mesmo vale para idade, que é int
  //int idade = 25;
  var idade = 25;
  print(idade.runtimeType);
  //e assim por diante
  var vaiChover = true;
  print (vaiChover.runtimeType);
  var salario = 2532.2;
  print (salario.runtimeType);
}
```

2.14 Constantes: final versus const Também podemos declarar **constantes**. Isso também envolve a inferência de tipos e mais: uma vez que a variável tenha sido inicializada, o ambiente Dart não permitirá que façamos nova atribuição. Aqui vamos utilizar a palavra chave **final**.

```
void main() {
    final nome = "João";
    print(nome);
    final idade = 17;
    print(idade);
    //String
    print(nome.runtimeType);
    //int
    print(idade.runtimeType);
    //erro, nome é constante
    //nome = "Pedro";
    //erro, idade também é constante
    //idade++;
    //observe que podemos falar o tipo explicitamente, embora não seja
    //necessário
    final String endereco = "Rua J";
    print(endereco);
    //String
    print(endereco.runtimeType);
    //observe que não necessariamente a inicializamos no momento em que
    //é declarada
    final peso;
    //mas é um erro tentar usar
    //erro, peso não foi inicializada
    //print(peso);
    //erro, se não foi inicializada também não tem tipo
    //print(peso.runtimeType);
}
```

Também podemos criar constantes com a palavra **const**. São constantes criadas em tempo de compilação e o ambiente Dart é capaz de gerar código otimizado quando as utilizamos. Mas quando utilizá-las? Utilizamos constantes com **const** quando sabemos seu valor no momento em que são declaradas e, além disso, esse valor é literal, conhecido em tempo de compilação.

Também podemos criar constantes com a palavra **const**. São constantes criadas em tempo de compilação e o ambiente Dart é capaz de gerar **código otimizado quando as utilizamos**. Mas quando utilizá-las? Utilizamos constantes com **const** quando

- sabemos seu valor no momento em que são declaradas e, portanto, já as inicializamos neste momento
- o valor atribuído é conhecido em tempo de compilação

```
void main() {
    //tudo bem, inicializado com valor literal, conhecido em tempo de
    //compilação
    const nome = "Ana";

    //também pode falar o tipo explicitamente, embora seja
    //desnecessário
    const String sobrenome = "Silva";

    //também vale com outros tipos
    const int idade = 18;
    const bool vaiChover = true;
    const deMaior = false;

    //também vale com interpolação de string
    const nomeCompletoInterpolacao = '$nome $sobrenome';

    //e com concatenação
    const nomeCompletoConcatenacao = nome + ' ' + sobrenome;

    //String String int bool bool String String
    print('${nome.runtimeType} ${sobrenome.runtimeType}
    ${idade.runtimeType} ${vaiChover.runtimeType}
    ${deMaior.runtimeType} ${nomeCompletoInterpolacao.runtimeType}
    ${nomeCompletoConcatenacao.runtimeType}');

    //erro, não está sendo inicializada
    //const outroNome;

    //erro, o valor é conhecido somente em tempo de execução
    //o compilador não é responsável por chamar o método toUpperCase
    //const maiusculas = nome.toUpperCase();
```

```
//observe que assim, tudo bem
//o compilador é capaz de fazer continhos
const soma = 2 + 2;
print(soma);
print(soma.runtimeType);
//tudo bem também
const n1 = 2, n2 = 3;
const n3 = n1 + n2;
print('$n1 + $n2 = $n3');
var n4 = 2;
//erro, var somente é conhecido em tempo de execução, o compilador
não pode resolver
//const n5 = n4 + 2;

final n6 = 5;
//com final também não dá, pois a inicialização em tempo de
compilação não é obrigatória
//const n7 = n6 + 2;
}
```

Nota. Como decidir entre **var**, **final** e **const**? Observe que var resolve o problema de maneira generalizada. Tudo aquilo que fazemos com final e const podemos fazer com var. Então por que não apenas utilizar var? Pelo **princípio do menor privilégio** e pela **possibilidade de otimização de código**. Resumindo, use:

- **var**: se for necessário alterar o valor ao longo do tempo. Ou seja, se existirem funcionalidades que requeiram este privilégio.
- **final**: se após a primeira atribuição o valor não puder mais ser alterado, mas ele não for conhecido no momento em que a constante for declarada ou não for literal.
- **const**: se após a primeira atribuição o valor não puder ser mais alterado e, além disso, ele já for conhecido no momento em que a constante for declarada e for literal, conhecido em tempo de compilação.

Comece por const. Se não puder usar, use final. Se não puder, use var.

2.14 Estruturas de seleção Dart possui as seguintes estruturas de seleção.

- **If, if/else, if/else encadeado, if/else aninhado**
- **Switch/case (switch statements, switch expressions)**

Veja algumas variações da estrutura if/else.

```
void main() {
    //if sem else (lembre-se do efeito das chaves)
    const idade = 19;
    if (idade > 18)
        print('Pode dirigir'); //dentro do if
        print('Até logo');//fora do if
    if (idade > 18){
        //dentro do if
        print ('Pode dirigir');
    }
    //fora do if
    print("Até logo");
    //if/else
    const nome = "Ana";
    if (nome.startsWith('A'))
        print('O nome começa com A');
    else
        print('O nome não começa com A');
    //também pode usar chaves (só no if, só no else ou em ambos, você
    escolhe)
    if (nome.startsWith('A')){
        print('O nome começa com A');
    }
    else{
        print('O nome não começa com A');
    }
    //if/else encadeado
    const nota = 10;
    if (nota >= 9)
        print("A");
    else if (nota >= 7)
        print("B");
    else if (nota >= 5)
        print("C");
```

```
else
    print("R");
//if/else aninhado
const numero = 18;
if (numero % 2 == 0) {
    print("É par");
    if (numero % 4 == 0)
        print("Divisível por 4");
    else
        print("Não é divisível por 4");
}
else{
    print("É ímpar");
    if (numero % 3 == 0){
        print("Divisível por 3 também");
    }
    else{
        print("Não é divisível por três");
    }
}
```

A linguagem Dart também possui a clássica estrutura **switch/case**. Veja um exemplo típico. Observe que estamos usando o comando break para cada cláusula case, supostamente enviando a lógica em queda do switch (fall-through).

```
void main() {  
    const nota = 9;  
    switch (nota) {  
        case 10:  
            print('A');  
            break;  
        case 9:  
            print('A');  
            break;  
        case 8:  
            print('B');  
            break;  
        case 7:  
            print('C');  
            break;  
        case 6:  
            print('D');  
            break;  
        case 5:  
            print('E');  
            break;  
        default:  
            print('R');  
            break;  
    }  
}
```

Há um aspecto importante a ser considerado quanto ao funcionamento da execução fall-through do switch/case em Dart: Ele somente acontece para cláusulas case que não possuam comando algum. Aquelas que possuem pelo menos um comando, como um print, têm um break implícito. Veja o exemplo a seguir.

```
void main() {  
    //observe que o Dart não tem suporte à execução fall-through para  
    cláusulas case não vazias (aquela em que não especificamos break e  
    múltiplos cases são executados)  
    //assim, as instruções break do primeiro exemplo são redundantes e  
    desnecessárias  
    //isso vale a partir da versão 3.0.0 do Dart  
    const nota = 10;  
    switch (nota) {  
        case 10:  
            print('A');//break implícito  
        case 9:  
            print('A');//break implícito  
        case 8:  
            print('B');//break implícito  
        case 7:  
            print('C');//break implícito  
        case 6:  
            print('D');//break implícito  
        case 5:  
            print('E');//break implícito  
        default:  
            print('R');//break implícito  
    }  
  
    //fall-through para cláusulas case vazias  
    switch (nota) {  
        case 10: //cláusula case vazia, sem break implícito, fall-through  
            acontece  
        case 9:  
            print('A');  
        case 8:  
            print('B');  
        case 7:  
            print('C');  
        case 6:  
            print('D');  
        case 5:  
            print('E');  
        default:  
            print('R');
```

```

    }
}
```

Há uma outra forma de fazer um único tratamento para valores diferentes. Podemos utilizar o operador `||`. Veja.

```

void main() {
    //veja outro exemplo que permite que dois valores sejam tratados no
    mesmo case
    const nota = 10;
    switch (nota) {
        case 10 || 9:
            print('A');
        case 8:
            print('B');
        case 7:
            print('C');
        case 6:
            print('D');
        case 5:
            print('E');
        default:
            print('R');
    }
}
```

O `switch/case` de Dart também é capaz de **lidar com Strings**. Há uma restrição: as strings que aparecem nas cláusulas `case` têm de ser conhecidas em tempo de compilação.

```

void main() {
    //com strings
    var vaiChover = "Sim";
    switch(vaiChover) {
        case 'Sim':
            print("Leve guarda chuva");
        default:
            print("Não leve guarda chuva");
    }
}
```

Também é possível manipular valores reais.

```
void main() {
    double nota = 9.7;
    switch (nota) {
        case > 9 && <= 10:
            print('A');
        case > 8 && <= 9:
            print('B');
        case > 7 && <= 8:
            print('C');
        case > 6 && <= 7:
            print('D');
        case > 5 && <= 6:
            print('E');
        default:
            print('R');
    }
}
```

E até listas.

```
void main() {
    var frutas = ['banana', 'laranja'];
    //switch com a lista inteira
    switch (frutas) {
        case ['banana', 'laranja']:
            print('banana e laranja');
        case ['banana', 'maçã']:
            print('banana e maçã');
        default:
            print('não sei');
    }
}
```

Na verdade, o switch/case de Dart é capaz de lidar com quaisquer **Patterns**. Veja mais no link a seguir.

<https://dart.dev/language/pattern-types>

Veja como utilizar o switch/case com **continue** em conjunto com um rótulo. Observe que não necessariamente os cases devem ser sequenciais.

```
void main() {
    var nota = 10;
    switch (nota) {
        case 10:
            print("Parabéns, você tirou 10!");
            continue conceito; // vai desviar para o rótulo conceito
        conceito://rótulo
        case 9:
            print("Você tirou um A!");
    }
}
```

Há um tipo de switch chamado de **switch expression**. Ele pode ser utilizado para produzir um valor que pode ser atribuído a uma variável, por exemplo.

- O símbolo => separa o valor envolvido no teste do switch do valor a ser produzido
- Cada cláusula é separada por vírgula
- Não escrevemos a palavra case
- O default é representado pelo símbolo _

```
void main() {
    var mediaFinal = 5;
    final conceito = switch(mediaFinal) {
        10 || 9 => 'A',
        8 => 'B',
        7 => 'C',
        6 => 'D',
        5 => 'E',
        _ => 'R' // faz o papel do default
    };
    print (conceito);
}
```

Exercícios

1. Calcule a área de um círculo com um raio de 5. (Use a fórmula da área do círculo: πr^2).
2. Encontre as raízes de uma equação quadrática com $a = 1$, $b = -3$ e $c = 2$. (Use a fórmula do discriminante: $x = [-b \pm \sqrt{b^2 - 4ac}] / 2a$). Nota: pesquise sobre a biblioteca math na documentação oficial e descubra como importá-la.
3. Calcule o volume de uma esfera com raio 4. (Use a fórmula do volume de uma esfera: $4/3\pi r^3$)
4. Determine o valor final obtido pela seguinte expressão numérica: $2 + 3 * 4 - (2 * 3) + 2^3$.
5. Converta uma temperatura de 100 graus Fahrenheit para Celsius. (Use a fórmula: $C = (F - 32) * 5/9$).
6. Converta uma temperatura de 36 graus Celsius para Fahrenheit. (Use a fórmula: $F = C * 9/5 + 32$)
7. Converta uma distância de 100 quilômetros para milhas. (Use a fórmula: $M = Km * 0.62137$).
8. Converta uma distância de 60 milhas para quilômetros. (Use a fórmula: $Km = M * 1.60934$)
9. Converta 100 libras para quilogramas. (Use a fórmula: $Kg = Lb * 0.453592$).
10. Converta 72 polegadas em metros. (1 polegada = 0.0254 metro).
11. Converta 3 litros em galões americanos. (1 litro = 0.264172 galão americano).
12. Converta 48 onças para gramas. (1 onça = 28.3495 gramas).
13. Converta 5 metros quadrados para pés quadrados. (1 metro quadrado = 10.7639 pés quadrados).
14. Converta 120 milhas por hora para metros por segundo. (1 milha por hora = 0.44704 metros por segundo).
15. Você tem um restaurante que funciona das 8h às 20h. Se um cliente chegar fora desse horário, ele deve ser informado de que o restaurante está fechado. Além disso, se o cliente vier entre as 14h e as 16h, ele deve ser informado de que é a hora do almoço.
16. Você é um bibliotecário e precisa verificar se um livro pode ser emprestado. Se o livro estiver disponível e não for um dos mais procurados, ele pode ser emprestado por 14 dias.

Se for um dos mais procurados, só pode ser emprestado por 7 dias. Se o livro não estiver disponível, ele não pode ser emprestado.

17. Você é um consultor de viagens e precisa informar aos clientes sobre o clima de seu destino de viagem. Se o destino for tropical, o clima será quente. Se for no norte, será frio. Se for no deserto, será quente durante o dia e frio à noite. Se for na montanha, será frio e possivelmente com neve.

18. Você precisa calcular o IMC de uma pessoa e classificá-la como "abaixo do peso", "normal", "sobrepeso" ou "obesidade", com base nos resultados.

19. Você está analisando os dados de uma postagem nas redes sociais. Se a postagem tiver mais de 100 curtidas e mais de 50 compartilhamentos, ela será considerada "popular". Se tiver menos de 10 curtidas e menos de 5 compartilhamentos, será considerada "não popular". Todas as outras postagens serão consideradas "médias".

20. Você está planejando um evento e precisa determinar a melhor data. Se a data proposta for durante a semana de trabalho e não houver outro evento programado para o mesmo dia, ela será considerada "ótima". Se for no fim de semana ou houver outro evento no mesmo dia, será considerada "ruim".

21. Você é um analista de ações e precisa aconselhar seus clientes sobre quando comprar ou vender ações. Se a ação estiver em alta e a empresa tiver bons lucros, é hora de vender. Se a ação estiver em baixa e a empresa tiver prejuízo, é hora de comprar. Em todos os outros cenários, é melhor esperar.

22. Você está jogando um jogo de estratégia e precisa determinar a melhor ação a tomar. Se o inimigo estiver atacando e suas defesas estiverem baixas, é melhor fortalecer suas defesas. Se o inimigo estiver atacando e suas defesas estiverem fortes, é melhor contra-atacar. Se o inimigo não estiver atacando, é melhor focar na coleta de recursos.

23. Você é um meteorologista e precisa prever o tempo para amanhã. Se a pressão do ar estiver caindo e houver umidade no ar, é provável que chova. Se a pressão do ar estiver subindo e o ar estiver seco, é provável que esteja ensolarado. Em todos os outros cenários, o tempo será parcialmente nublado.

24. Você é um detetive e precisa determinar se um suspeito é culpado ou inocente. Se o suspeito tiver um álibi sólido, ele é inocente. Se o suspeito não tiver um álibi e houver evidências físicas que o liguem ao crime, ele é culpado. Em todos os outros casos, mais investigação é necessária.

25. Dados dia e mês de nascimento, determine o signo da pessoa usando switch/case.

26. Dado um alimento predefinido, seu programa deve ser capaz de exibir a quantidade de calorias desse alimento. Pesquise 5 alimentos na Internet.

27. O programa deve simular uma rodada de pedra, papel e tesoura. As escolhas do jogador e do computador podem ser predefinidas.

Resolva os seguintes exercícios do Beecrowd. Os valores “lidos” podem ser fixos no programa. Ou se desejar, pesquise sobre o pacote `dart:io`. Observe que o `dartpad` não tem suporte a este pacote.

<https://www.beecrowd.com.br/judge/pt/problems/view/1008>

<https://www.beecrowd.com.br/judge/pt/problems/view/1019>

<https://www.beecrowd.com.br/judge/pt/problems/view/1018>

Referências

Dart programming language | Dart. Google, 2023. Disponível em <<https://dart.dev/>>. Acesso em agosto de 2023.

1 Introdução

Neste material, prosseguimos com os estudos sobre as principais características da linguagem Dart.

2 Desenvolvimento

2.1 Estruturas de repetição

As estruturas de repetição de Dart são as seguintes.

- **for**
- **for each**
- **while**
- **do/while**

Veja um exemplo de for.

```
void main() {
  for (int i = 0; i < 10; i++) {
    print(i);
  }
}
```

No exemplo a seguir, iteramos sobre uma coleção (veremos mais sobre coleções em breve) utilizando um for each.

```
void main() {
  const nomes = ['Pedro', 'Maria', 'João'];
  //em vez de final, podemos usar var e o tipo
  //explicito da variavel (String nesse caso)
  //mas não podemos usar const
  for (final nome in nomes) {
    print(nome);
  }
}
```

Como em outras linguagens de programação, as três regiões do for são opcionais. No exemplo a seguir, dado que não há condição de continuidade, temos um loop infinito.

```
void main() {
    //loop infinito com for
    for ( ; ; ) {
        print('loop infinito');
    }
}
```

O while é semelhante àquele que conhecemos de outras linguagens.

```
void main() {
    //loop com while
    int contador = 0;
    while (contador <= 10) {
        print("Contador: $contador");
        contador++;
    }
}
```

Assim como o do/while.

```
void main() {
    //loop com do/while
    int i = 0;
    do {
        print("O valor de i é $i");
        i++;
    } while (i < 10);
}
```

2.2 enum Muitas linguagens de programação oferecem um recurso conhecido como enum. Ele permite que especifiquemos uma lista de constantes, promovendo a legibilidade do programa. Por exemplo, ao invés de representarmos os dias da semana como números de 1 a 7, podemos criar um enum, que é uma lista com os nomes dos dias explícitos.

```

enum DiaSemana { SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO,
DOMINGO}

void main() {
    //exibindo a lista inteira
    print(DiaSemana.values);
    DiaSemana dia = DiaSemana.SEGUNDA;
    //exibindo o nome desse valor
    print(dia.name);
    switch (dia) {
        case DiaSemana.SEGUNDA:
            print("Nããããããããoooooooo!!!!!!!");
        case DiaSemana.TERCA:
            print(":(");
        case DiaSemana.QUARTA:
            print(":|");
        case DiaSemana.QUINTA:
            print(":)");
        case DiaSemana.SEXTA:
            print("Eeeeeeeeeeeeeeee!!!!!!!");
        case DiaSemana.SABADO:
            print("=DDDDDD");
        case DiaSemana.DOMINGO:
            print(" :):");
    }
}

```

2.3 Instalação do SDK, VS Code, pasta e teste inicial Passaremos a utilizar uma instalação local de Dart e editaremos código utilizando o VS Code. O SDK de Dart pode ser obtido a partir do seguinte link.

<https://dart.dev/get-dart>

Se tiver planos de utilizar o framework Flutter, pode ser boa ideia fazer a sua instalação. Inclusive, ela já inclui o SDK de Dart, já que Flutter é escrito em Dart. Por isso, **se optar por instalar o Flutter, não precisará instalar o Dart à parte. Flutter pode ser obtido em**

<https://docs.flutter.dev/get-started/install>

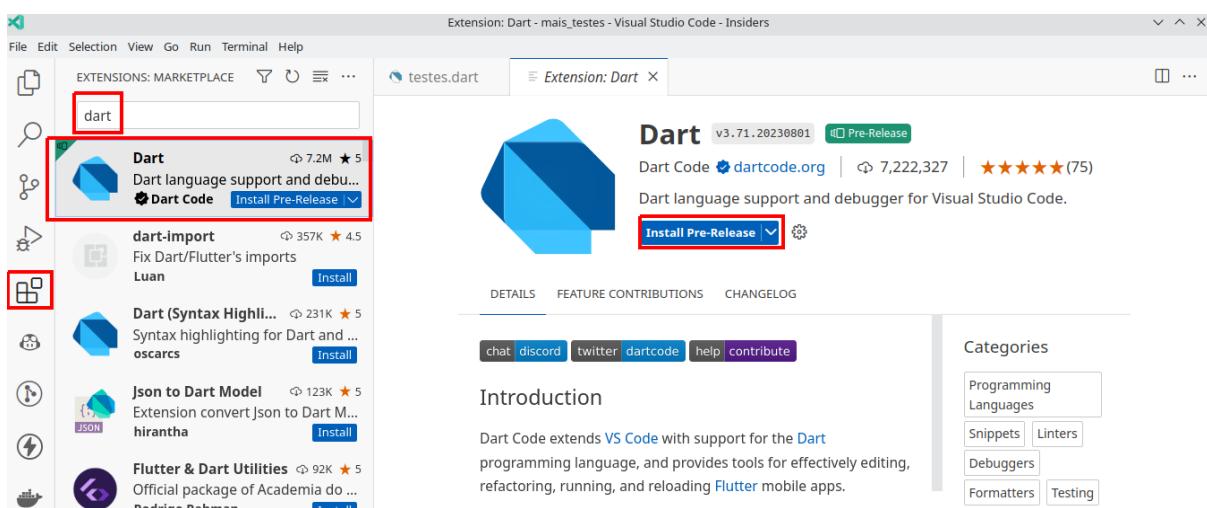
A seguir, crie uma pasta para abrir os seus projetos. Se estiver usando o Windows, pode ser interessante usar algo como

C:\Users\seuUsuario\Documents\projetos_dart\mais_testes

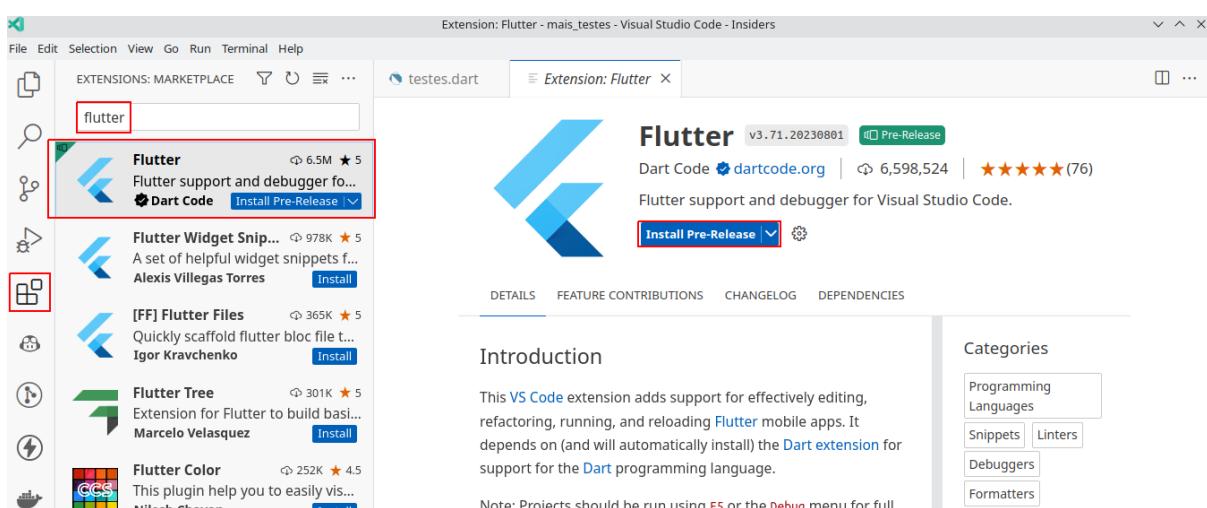
A seguir, abra o VS Code e clique File >> Open Folder. Navegue até a pasta e vincule o VS Code a ela. No VS Code, crie um arquivo chamado **testes.dart**. Claro, o nome pode ser qualquer um.

É recomendável fazer a instalação da extensão Dart para o VS Code. Para tal, abra a seção Extensions dele e busque por Dart.

Nota. É natural que seu VS Code mostre o texto **Install** em vez de **Install Pre-Release**, a menos que também esteja utilizando a versão insiders do VS Code.



Caso vá desenvolver com Flutter, também é interessante fazer a instalação da extensão própria para esse arcabouço. Observe que se optar por fazê-lo, estará instalando também a extensão Dart, pois ela é uma dependência da extensão Flutter.



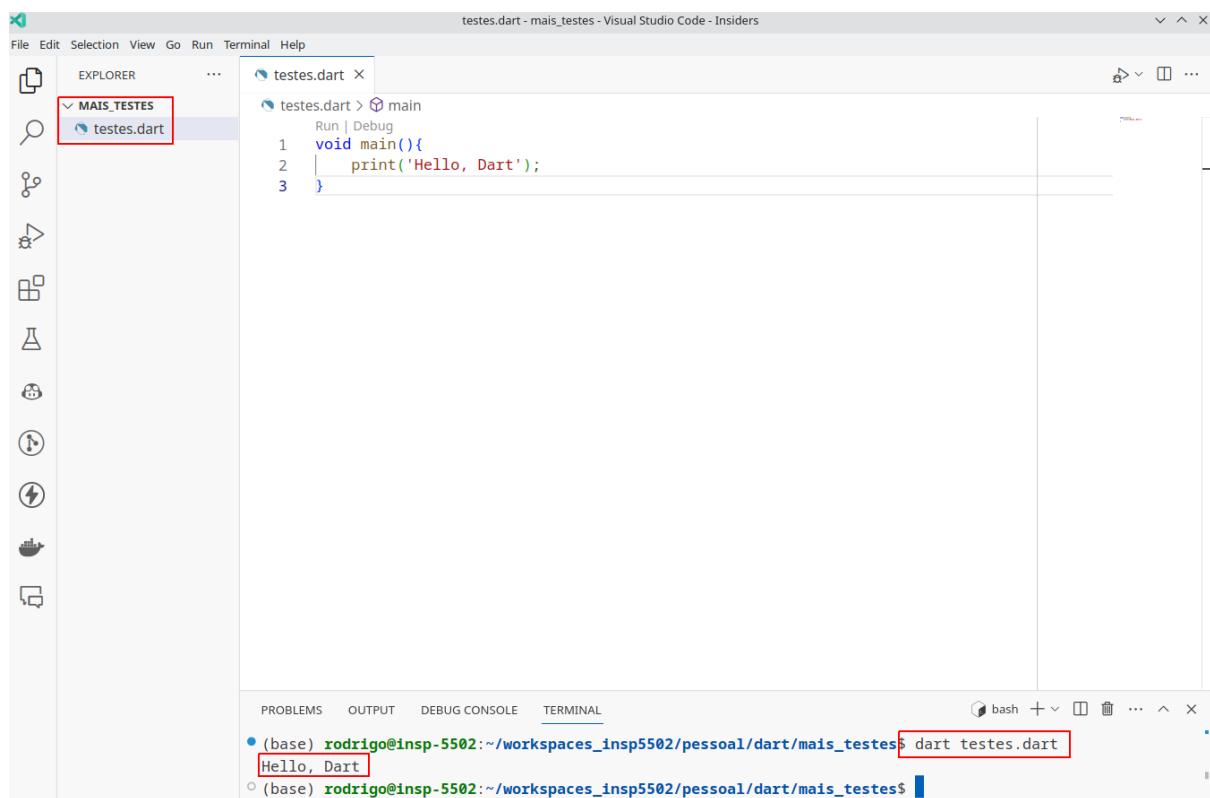
Abra o arquivo **testes.dart** no VS Code e use o seguinte código para testar o seu ambiente.

```
void main() {  
  print('Hello, Dart');  
}
```

Clique Terminal >> New Terminal para abrir um terminal interno do VS Code. Use

dart testes.dart

para fazer o teste. Veja o resultado esperado.



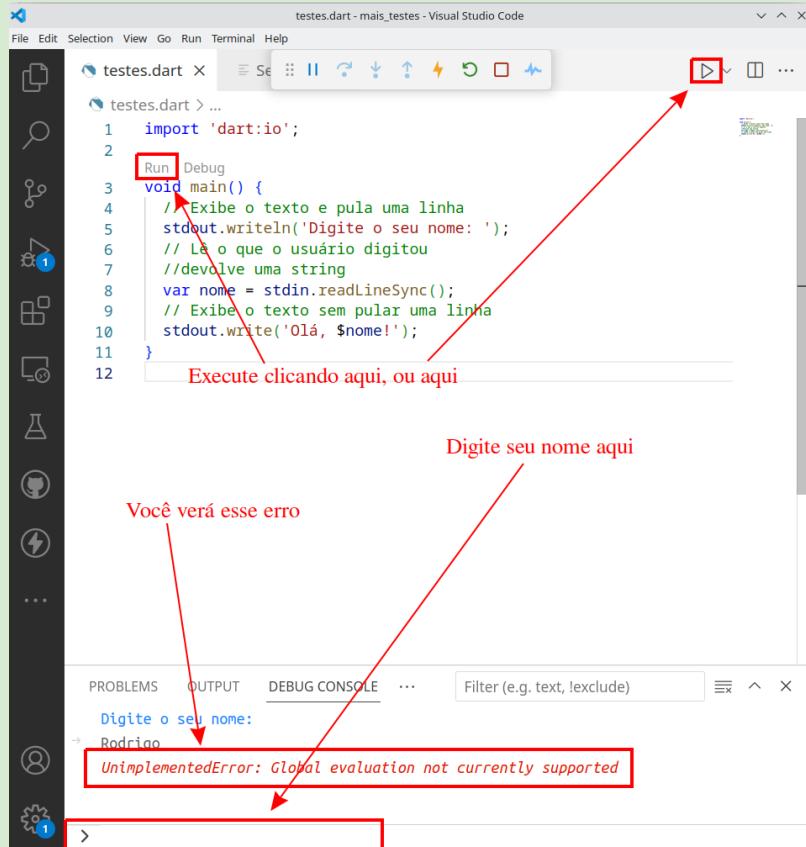
2.4 Entrada e saída de dados Os objetos stdin e stdout representam, respectivamente, a entrada e a saída padrão. Eles fazem parte do pacote dart:io. Podemos utilizá-los para interagir com o usuário.

```

import 'dart:io';
void main() {
    // Exibe o texto e pula uma linha
    stdout.writeln('Digite o seu nome: ');
    // Lê o que o usuário digitou
    //devolve uma string
    var nome = stdin.readLineSync();
    // Exibe o texto sem pular uma linha
    stdout.write('Olá, $nome!');
}

```

Nota. Quando executamos um programa Dart utilizando os atalhos gráficos da extensão, como na figura a seguir, por padrão, a execução é feita utilizando-se o **Debug Console** provido pela extensão.



É importante sabermos que:

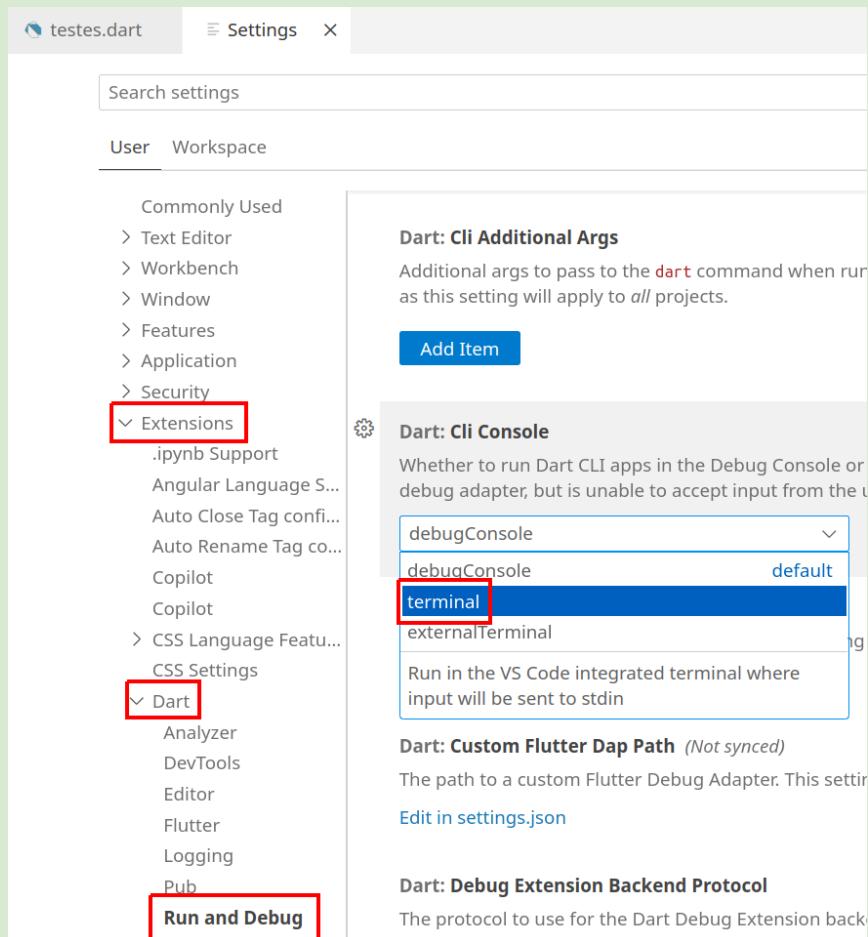
- Como o nome sugere, ele tem suporte a um mecanismo para depuração (execução pausada linha a linha) de código

- Não é possível fazer a entrada de dados usando o pacote **dart:io**.

Se desejar, é possível alterar o ambiente de execução utilizado pela extensão. Podemos deixar de utilizar o debug console e passar a utilizar o terminal padrão. Com isso

- perdemos o suporte a debug oferecido pelo debug console
- podemos capturar dados no terminal usando o pacote **dart:io** ainda que executemos a aplicação usando os atalhos visuais da extensão.

Essa configuração pode ser feita clicando-se em **File >> Preferences >> Settings**. Depois disso, clique em **Extensions >> Dart >> Run and Debug**. Encontre a opção **Cli Console** à direita e troque para **terminal**. Observe.



Se desejar, também é possível não fazer alteração alguma e, quando for necessário capturar dados usando o pacote **dart:io**, executar o programa com **dart arquivo.dart** direto no terminal, em vez de utilizar os atalhos gráficos da extensão.

Observe que o método de captura de dados devolve uma String. Se for necessário lidar com números, é preciso fazer a conversão usando os métodos parse de int, num e/ou double.

```
import 'dart:io';
void main() {
//erro, precisa converter antes
int idade = stdin.readLineSync();
}
```

Pode ser tentador fazer a seguinte conversão. Mas ela também causa erro.

```
import 'dart:io';
void main() {
//erro
//estamos tentando entregar String? para parse
//ele somente recebe String
int idade = int.parse(stdin.readLineSync());
}
```

Isso acontece pois

- O método readLineSync devolve **String?**. É um tipo que representa uma string ou null.
- O método parse recebe **String**. É um tipo que representa uma string necessariamente diferente de null.

Ou seja, é um erro tentar passar null ao método parse. Algo assim: int.parse(null);

E agora?

Você pode fazer validações com estruturas de seleção e/ou repetição, dependendo do que deseja.

```

import 'dart:io';

void main() {
print('Digite sua idade: ');
//String? admite uma string ou null
String? entrada = stdin.readLineSync();
//validação com if else
if (entrada != null) {
int idade = int.parse(entrada);
print ("Você tem $idade anos");
}

//validação com while
while (entrada == null) {
print('Digite sua idade: ');
entrada = stdin.readLineSync();
}
int idade = int.parse(entrada);
print ("Você tem $idade anos");
}

```

Outra opção é usar o operador `!`. Ele é uma promessa de que a expressão sobre a qual opera é diferente de null. Mas observe que estamos nos sujeitando a um erro envolvendo null em tempo de execução.

```

import 'dart:io';
void main() {
print('Digite sua idade: ');
//o operador ! opera sobre a expressão que o antecede
//neste caso, o retorno do método readLineSync()
//estamos prometendo que o valor não será nulo para o
compilador
//ele fala: blz, você se vira com a Dart VM
int idade = int.parse(stdin.readLineSync()!);
print(idade);
}

```

2.5 Controlando o número de casas decimais Podemos controlar o número de casas decimais usando o método `toStringAsFixed`.

```
void main() {
  //com um número fixo
  print('${154.3434234.toStringAsFixed(2)}');
  //com uma variável
  var variavel = 2.3134353543;
  print('${variavel.toStringAsFixed(2)}');
}
```

2.6 Valores aleatórios A geração de valores aleatórios pode ser feita por métodos da classe `Random`.

```
import 'dart:math';

void main() {
  var gerador = Random();
  //inteiro aleatório entre 0 e 5
  var n1 = gerador.nextInt(6);
  print(n1);

  //inteiro aleatório entre 1 e 10
  var n2 = gerador.nextInt(10) + 1;
  print(n2);

  //inteiro aleatório entre 0 e 100
  var n3 = gerador.nextInt(101);
  print(n3);

  //real aleatório entre 0 e 1 ([0, 1])
  var n4 = gerador.nextDouble();
  print(n4);

  //real aleatório entre 0 e 100 ([0, 100])
  var n5 = gerador.nextDouble() * 100;
  print(n5);

  //real aleatório entre 100 e 150
  var n6 = gerador.nextDouble() * 50 + 100;
  print(n6);

  //booleano aleatório
  var n7 = gerador.nextBool();
  print(n7);
}
```

2.7 Papel, pedra e tesoura Utilizemos os recursos vistos até então, ou grande parte deles, para fazer a implementação de um clássico Papel, pedra e tesoura. Para isso, vamos criar uma nova aplicação. Desta vez, ela será um projeto Dart “completo”. Para tal, abra um terminal vinculado à sua pasta de projetos Dart (fora de qualquer projeto específico) e use

```
dart create pedra_papel_tesoura -t console
```

para criar a aplicação.

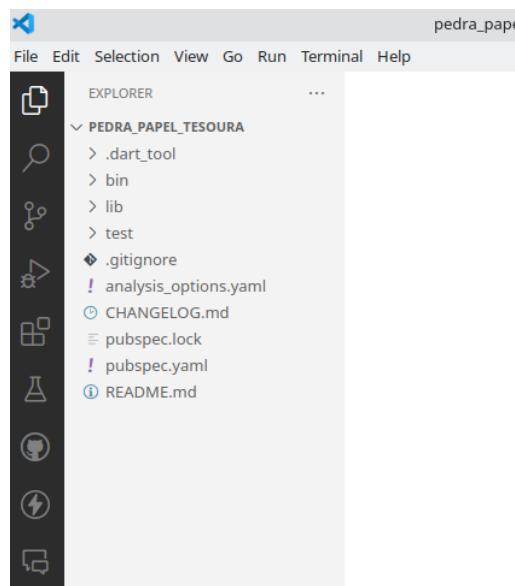
Nota. Use as regras descritas a seguir para escolher o nome de sua aplicação. Basicamente, use **snake_case**.

<https://dart.dev/tools/pub/pubspec#name>

Nota. O parâmetro **t** significa template ou modelo. Estamos usando o modelo console, ou seja, estamos criando uma aplicação para executar na linha de comando. Há outros modelos. Veja:

<https://dart.dev/tools/dart-create>

Depois de criar o projeto, abra o VS Code e clique em **File > Open Folder**. Abra a pasta que acaba de ser criada. Veja o resultado esperado.



Veja uma descrição sucinta de cada item.

.dart_tool: pasta que contém arquivos utilizados por várias ferramentas Dart.

bin: pasta que contém o arquivo .dart principal. A ideia é que definamos a função main ali.

lib: pasta que contém arquivos .dart em geral. É comum fazermos a implementação da aplicação nesta pasta e em subpastas dela.

test: pasta que contém arquivos de teste. Eles podem ser executados com **dart test**.

.gitignore: o clássico arquivo que o Git utiliza para decidir quais arquivos não incluir no repositório

analysis_options.yaml: arquivo de configurações utilizado para, por exemplo, configurar o **linter** do Dart.

Nota. No dicionário, é possível encontrar a seguinte definição para a palavra linter.

“a machine for removing the short fibers from cotton seeds after ginning.”

Traduzindo:

“uma máquina para remover as fibras curtas das sementes de algodão após o descaroçamento”

Há também a seguinte definição:

“línter é um subproduto obtido da planta do algodão que tem uma grande importância. Trata-se das fibras bem curtinhas que ficam grudadas no caroço do algodão após a retirada das mais longas. É o que “sobra” no beneficiamento.”

<https://soudealgodao.com.br/blog/o-que-e-linter-conheca-o-residuo-de-algodao-que-e-pau-pra-toda-obra/>

Também é interessante falar da palavra **lint**, que significa algo como “fiapo”. lint também é o nome de uma ferramenta criada por Stephen C. Johnson, conhecido por ter trabalhado, por exemplo, no desenvolvimento do yacc (yet another C Compiler).

Resumindo, quando falamos de linter/lint, estamos nos referindo a características indesejáveis (fiapos) no código que desejamos remover.

Veja os fiapos “core” e recomendados de Dart aqui:

<https://pub.dev/packages/lints>

CHANGELOG.md um arquivo escrito em Markdown para registrar as alterações feitas na aplicação ao longo do tempo.

Nota. A linguagem Markdown é muito utilizada e é muito útil conhecê-la. Veja mais em:

<https://www.markdownguide.org/>

pubspec.lock: utilizamos para garantir que todos os desenvolvedores de um projeto estão utilizando as mesmas versões de cada dependência. Veja mais aqui:

<https://medium.com/codingmountain-blog/what-is-pubspec-lock-and-why-should-flutter-devs-care-86ab4d0b47fc>

pubspec.yaml Especificamos as dependências do projeto aqui.

README.md Arquivo em markdown que descreve aspectos do projeto, como ele pode ser executado etc.

Sua vez! Tente fazer o jogo, com a ajuda do seu professor. Seja criativo! Faça validações, use enum, variações de switch/case etc.

Veja uma possível implementação. Observe que usamos alguns recursos a serem estudados mais a fundo em breve.

Abra o arquivo **pedra_papel_tesoura.dart** da pasta **lib**. Apague todo o seu conteúdo e substitua pela seguinte função. Ela representa o nosso jogo.

```
void jogo() {  
}
```

A seguir, abra o arquivo **pedra_papel_tesoura.dart** da pasta **bin** e ajuste seu conteúdo como a seguir.

```
//show indica que queremos apenas aquele membro  
import 'package:pedra_papel_tesoura/pedra_papel_tesoura.dart' show  
jogo;  
  
void main(List<String> arguments) {  
  jogo();  
}
```

A seguir, no arquivo **pedra_papel_tesoura.dart** da pasta lib, importamos os pacotes

- dart:io para capturar as opções do usuário
- dart:math para gerar valores aleatórios, representando a opção do usuário

```
import 'dart:io';
import 'dart:math';
```

Escrevemos um **enum** para representar as opções do jogo, incluindo a opção de sair.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }
```

Na função `jogo`, vamos

- escrever um loop que executa até que o usuário digite 4, indicando que deseja sair.
- a cada iteração, usar a função `sleep` alimentada com uma instância de `Duration` para que, a cada iteração, aguardemos 3 segundos.

Os comentários na função `jogo` indicam as funções que precisamos implementar para obter a implementação completa do jogo, aplicando o paradigma da **divisão e conquista**: cada pedacinho vai ser implementado por uma função separada. Observe que haverá um erro de compilação temporário já que a variável que representa a opção do usuário ainda não foi inicializada. Nada para se preocupar. Ela será inicializada na hora certa.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }

void jogo() {
    int opUsuario;
    do{

        //exibir menu

        //capturar opção do usuário, validando

        //se o usuário digitar 4, sair

        //senão

        //sortear escolha do computador

        //mapear opcao usuario, de int para enum

        //mapear opcao computador, de int para enum

        //exibir as opções de cada um

        //decidir quem venceu, ou se houve empate

        //exibir o resultado
        sleep(Duration(seconds: 3));
    }while(opUsuario != 4);
}
```

Embora não seja estritamente necessário, vamos escrever uma função que exibe algo recebido como parâmetro, essencialmente operando da mesma forma que a função print. De fato, ela usa a função print para cumprir a sua missão.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }

void exibe(String texto) {
    print(texto);
}

void jogo() {
    ...
}
```

A seguir, escrevemos uma função responsável por capturar a opção do usuário.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }

void exibe(String texto) {
    print(texto);
}

int pegaOpcaoUsuario() {
    return int.parse(stdin.readLineSync()!);
}

void jogo() {
    ...
}
```

A função a seguir recebe uma opção e verifica se ela está no intervalo válido.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }

void exibe(String texto) {
    print(texto);
}

int pegaOpciaoUsuario() {
    return int.parse(stdin.readLineSync()!) ;
}

bool opcaoEhValida(int opcao) {
    return opcao >= 1 && opcao <= 4;
}

void jogo() {
    ...
}
```

Uma vez que tenhamos uma opção em mãos, representada como um número inteiro, precisaremos fazer um mapeamento de int para enum. Lembre-se que o índice do enum começa a partir do zero (pedra = 0, papel = 1 etc). Para tornar intuitivo para o usuário, deixamos ele fazer a sua escolha no intervalo de 1 a 4. Por isso, subtraímos 1 na hora de indexar o enum.

```
import 'dart:io';
import 'dart:math';

enum OPCAO { pedra, papel, tesoura, sair }

void exibe(String texto) {
    print(texto);
}

int pegaOpcãoUsuario() {
    return int.parse(stdin.readLineSync()!) ;
}

bool opçãoEhValida(int opção) {
    return opção >= 1 && opção <= 4;
}

OPCAO mapeiaOpção(int opção) {
    return OPCAO.values[opção - 1];
}

void jogo() {
    ...
}
```

A próxima função verifica se houve empate. Se não houve, ela diz quem venceu.

```
...
OPCAO mapeiaOpcao(int opcao) {
    return OPCAO.values[opcao - 1];
}

String decideResultado(OPCAO opcaoUsuario, OPCAO opcaoComputador) {
    if (opcaoUsuario == opcaoComputador) return "Empate";
    if (opcaoUsuario == OPCAO.papel && opcaoComputador == OPCAO.pedra || 
        opcaoUsuario == OPCAO.pedra && opcaoComputador == OPCAO.tesoura || 
        opcaoUsuario == OPCAO.tesoura && opcaoComputador == OPCAO.papel) {
        return "Você venceu";
    }
    return "Computador venceu";
}

void jogo() {
    ...
}
```

Agora podemos começar a implementar o jogo. Começamos exibindo o menu ao usuário, capturando a sua opção e verificando se ele já deseja sair, por ter digitado o valor 4.

```

...
void jogo() {
    int opUsuario;
    do{
        //exibir menu
        //capturar opção do usuário, validando
        do{
            exibe('1-Pedra\n2-Papel\n3-Tesoura\n4-Sair');
            opUsuario = pegaOpcãoUsuario();
            }while(!opcaoEhValida(opUsuario));
        //se o usuário digitar 4, sair
        //senão
        if(opUsuario != 4){
            //sortear escolha do computador

            //mapear opção usuário, de int para enum

            //mapear opção computador, de int para enum

            //exibir as opções de cada um

            //decidir quem venceu, ou se houve empate

            //exibir o resultado

            sleep(Duration(seconds: 3));
        }
    }while(opUsuario != 4);
    exibe('Até logo');
}
}

```

Verifique se está tudo ok executando o programa com

dart run

A escolha do computador pode ser simulada pelo método `nextInt` da classe `Random`.

```
void jogo() {
    int opUsuario;
    do{
        //exibir menu
        //capturar opção do usuário, validando
        do{
            exibe('1-Pedra\n2-Papel\n3-Tesoura\n4-Sair');
            opUsuario = pegaOpcaoUsuario();
        }while(!opcaoEhValida(opUsuario));
        //se o usuário digitar 4, sair
        //senão
        if(opUsuario != 4){
            //sortear escolha do computador
            int opComputador = Random().nextInt(3) + 1;
            //mapear opcao usuario, de int para enum

            //mapear opcao computador, de int para enum

            //exibir as opções de cada um

            //decidir quem venceu, ou se houve empate

            //exibir o resultado

            sleep(Duration(seconds: 3));
        }
    }while(opUsuario != 4);
    exibe('Até logo');
}
```

O próximo passo é traduzir as opções do usuário e do computador, fazendo uso do enum. Podemos também exibi-las.

```

void jogo() {
    int opUsuario;
    do{
        //exibir menu
        //capturar opção do usuário, validando
        do{
            exibe('1-Pedra\n2-Papel\n3-Tesoura\n4-Sair');
            opUsuario = pegaOpcãoUsuario();
        }while(!opcaoEhValida(opUsuario));
        //se o usuário digitar 4, sair
        //senão
        if(opUsuario != 4){
            //sortear escolha do computador
            int opComputador = Random().nextInt(3) + 1;
            //mapear opção usuário, de int para enum
            OPCAO opcaoUsuario = mapeiaOpcão(opUsuario);
            //mapear opção computador, de int para enum
            OPCAO opcaoComputador = mapeiaOpcão(opComputador);
            //exibir as opções de cada um
            exibe(
                'Você (${opcaoUsuario.name}) vs (${opcaoComputador.name})'
                'Computador'
            );
            //decidir quem venceu, ou se houve empate
            //exibir o resultado
            sleep(Duration(seconds: 3));
        }
    }while(opUsuario != 4);
    exibe('Até logo');
}

```

Execute novamente com

dart run

A seguir, resta

- decidir o vencedor (ou se houve empate)
- exibir o resultado
- exibir ***** para separar uma rodada da outra

```

void jogo() {
    int opUsuario;
    do{
        //exibir menu
        //capturar opção do usuário, validando
        do{
            exibe('1-Pedra\n2-Papel\n3-Tesoura\n4-Sair');
            opUsuario = pegaOpcãoUsuario();
        }while(!opcaoEhValida(opUsuario));
        //se o usuário digitar 4, sair
        //senão
        if(opUsuario != 4){
            //sortear escolha do computador
            int opComputador = Random().nextInt(3) + 1;
            //mapear opção usuario, de int para enum
            OPCAO opcaoUsuario = mapeiaOpcão(opUsuario);
            //mapear opção computador, de int para enum
            OPCAO opcaoComputador = mapeiaOpcão(opComputador);
            //exibir as opções de cada um
            exibe(
                'Você (${opcaoUsuario.name}) vs (${opcaoComputador.name})'
            Computador'
            );
            //decidir quem venceu, ou se houve empate
            String vencedor =
                decideResultado(opcaoUsuario, opcaoComputador);
            //exibir o resultado
            exibe(vencedor);
            exibe('*****');
            sleep(Duration(seconds: 3));
        }
    }while(opUsuario != 4);
    exibe('Até logo');
}

```

Faça novos testes com

dart run

Referências

Dart programming language | Dart. Google, 2023. Disponível em <<https://dart.dev/>>. Acesso em agosto de 2023.

1 Introdução

Neste material, prosseguimos com os estudos sobre as principais características da linguagem Dart.

2 Desenvolvimento

2.1 Coleções Dart possui três tipos comuns de coleções

- Listas
- Tuplas ou Records
- Conjuntos (Sets)
- Mapas

Vamos estudar sobre suas principais características. Comece criando um novo projeto com

dart create colecoes -t console

Lembre-se de executar este comando fora de qualquer projeto. Embora seja comum utilizar a pasta **lib** para fazer a codificação, nestes exemplos iniciais vamos simplificar usando o arquivo **colecoes.dart** da pasta **bin**. Observe que ele já possui o método main.

Nota. Observe que a função **main** do projeto possui um parâmetro do tipo **List<String>** É uma lista que permite que entreguemos valores ao programa quando o inicializarmos.

```
void main(List<String> arguments) {
    print(arguments);
}
```

E execute o programa com

dart run colecoes 2 3

Observe que a lista contém os dois valores passados como parâmetro.

2.2 Listas Listas podem ser manipuladas com o operador []. Embora a notação seja semelhante a um vetor “baixo nível” de linguagens como Java e C, o objeto construído é do tipo **List <String>**. Observe, também, que o método print se encarrega de chamar o método `toString` caso não o façamos explicitamente.

```
void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    print(nomes);
    print(nomes.toString());
    print(nomes.runtimeType);
}
```

Podemos acessar os elementos de uma lista usando o operador [] também. Cada um tem a sua posição, começando a contagem do zero.

```
void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    print(nomes);
    print(nomes.toString());
    print(nomes.runtimeType);
    print(nomes[0]);
    print(nomes[1]);
}
```

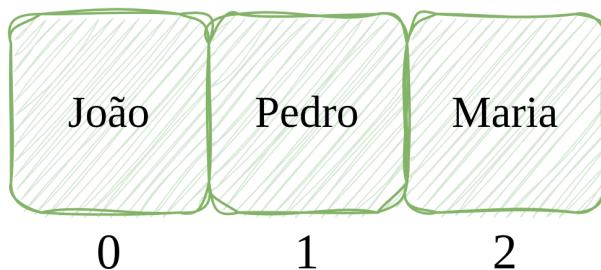
Observe que causamos uma exceção do tipo **RangeError** caso tentemos acessar a lista utilizando uma posição inválida. Claro, trata-se de uma exceção que acontece em tempo de execução.

```

void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    print(nomes);
    print(nomes.toString());
    print(nomes.runtimeType);
    print(nomes[0]);
    print(nomes[1]);
    //RangeError
    print(nomes[-1]);
    //RangeError
    print(nomes[3]);
}

```

Veja a figura.



Também podemos alterar valores da coleção.

```

void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    nomes[0] = 'José';
    print(nomes);
}

```

Podemos iterar sobre uma lista usando uma estrutura de repetição “comum” ou um “*for each*”. Observe que o número de elementos que a lista contém pode ser obtido por meio de sua propriedade **length**.

```

void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    //for comum
    for (int i = 0; i < nomes.length; i++){
        print(nomes[i]);
    }
    //for each
    for (final nome in nomes){
        print(nome);
    }
}

```

Dado que o compilador já inferiu o tipo **List <String>**, não podemos armazenar um objeto de qualquer outro tipo.

```

void main(List<String> arguments) {
    var nomes = ['João', 'Pedro', 'Maria'];
    //erro, a lista é de strings
    nomes[0] = 2;
}

```

Entretanto, se a lista for criada utilizando-se objetos de tipos diversos, o tipo inferido será **List <Object>**. Neste caso, podemos armazenar qualquer coisa que passe no teste **É-UM Object**.

```

void main(List<String> arguments) {
    var itensDiversos = ['Ana', true, 2, 2.5];
    print(itensDiversos);
    //List<Object>
    print(itensDiversos.runtimeType);
    //agora pode
    itensDiversos[0] = false;
    print(itensDiversos);
}

```

Exercício. Escreva um programa que faz a soma dos elementos recebidos como parâmetro pelo método main. Lembre-se de fazer conversões apropriadas. Execute o programa com

dart run colecoes 1 2 3

Vejamos algumas operações básicas que podemos realizar com listas. Veja os comentários.

```
void main(List<String> arguments) {
    var nomes = ['Ana', 'João', 'Maria'];
    //responde se a lista está vazia
    print(nomes.isEmpty);
    //responde se a lista não está vazia
    print(nomes.isNotEmpty);
    //devolve um Iterable<String> contendo os elementos em ordem reversa
    //não altera a lista atual
    print(nomes.reversed);
    //devolve o primeiro elemento da lista
    //se ela estiver vazia, causa um erro
    print(nomes.first);
    //devolve o primeiro ou null, sem causar erro
    print(nomes.firstOrNull);
    //lista vazia
    //Bad state: no element
    //print([] .first);
    //aqui tudo bem, devolve null
    print([] .firstOrNull);
    //o mesmo vale para o último elemento
    print(nomes.last);
    print(nomes.lastOrNull);
}
```

Podemos adicionar elementos a uma lista usando o método **add**. O elemento será adicionado ao final. Se desejarmos podemos adicionar um elemento usando o método **insert**. Neste caso, especificamos a posição em que desejamos que ele seja adicionado. O método **insert** **desloca** os elementos para a direita, se for o caso. Se a coleção possui **n** elementos, **n** é a última posição válida que podemos especificar.

```
void main(List<String> arguments) {
    var nomes = ['Ana', 'João', 'Maria'];
    //adiciona na última posição
    nomes.add('Cristina');
    print(nomes);
    //insere na posição 0
    nomes.insert(0, 'Ana Maria');
    //aqui a lista tem 5 elementos
    print(nomes);
    //podemos adicionar na posição 5
    //obtendo o mesmo funcionamento do add
    nomes.insert(5, 'Vagner');
    print(nomes);
    //aqui a lista tem 6 elementos
    //não podemos adicionar em qualquer posição a partir da 7
    //RangeError
    nomes.insert(7, 'Antônio');
}
```

O método **contains** responde se a lista contém um determinado elemento.

```
void main(List<String> arguments) {
    var nomes = ['Ana', 'João', 'Maria'];
    //true
    print(nomes.contains('Ana'));
    //false
    print(nomes.contains('ANA'));
    //false
    print(nomes.contains('Pedro'));
}
```

Sem usar inferência de tipo, podemos declarar uma lista da seguinte forma.

```
void main(List<String> arguments) {
    List<String> nomes = ['Ana', 'Pedro'];
    print(nomes.runtimeType);
    List <int> idades = [17, 22];
    print(idades.runtimeType);
    List <bool> deMaior = [false, true];
    print(deMaior.runtimeType);

    //podemos também ter uma lista de listas
    var listas = [nomes, idades, deMaior];
    //essa é uma List<>List<Object>>
    print(listas.runtimeType);
    //sem o tipo genérico também pode
    //aqui temos uma lista de dynamic
    //ou seja, ela armazena qualquer coisa
    List lista = [];
    lista.add(true);
    lista.add("Ana");
    print(lista.runtimeType);
    print(lista);
}
```

Também podemos restringir o tipo do objeto armazenado numa lista usando **type annotation**. Observe.

```
void main(List<String> arguments) {
    //List<Object>
    var qualqueroCoisa = [1, true, 'Ana'];
    //<List<String> com type annotation
    var somenteStrings = <String> ['Ana', 'Pedro'];
    print(qualqueroCoisa.runtimeType);
    print(somenteStrings.runtimeType);}
```

Exercício. Escreva um programa que:

- pede ao usuário que faça um jogo da mega sena com 6 números. Use uma lista para armazená-los. Não admita repetições.
- gera um jogo de 6 números da mega sena usando Random e guarda numa lista.
- exibe o jogo do usuário lado a lado com o jogo gerado, ambas ordenadas
- mostra ao usuário quais números ele acertou.

Podemos usar a palavra **final** para declarar listas. Configura seu funcionamento.

```
void main(List<String> arguments) {
    //ok
    final nomes = ['Ana', 'Pedro'];
    //pode alterar o conteúdo da lista!
    nomes[0] = 'Ana Maria';
    //mas não pode alterar o objeto referenciado pela constante nomes
    //erro em tempo de compilação
    nomes = ['João', 'Maria'];
}
```

Também podemos usar **const**. Lembre-se que só é possível se a lista inteira for conhecida em tempo de compilação. Observe que o conteúdo da lista também não pode ser alterado neste caso.

```
void main(List<String> arguments) {
    //ok
    final nomes = ['Ana', 'Pedro'];
    //não podemos alterar o objeto referenciado
    //erro em tempo de compilação
    //nomes = ['João'];
    //também não podemos alterar o conteúdo
    //mas esse é um erro em tempo de execução!
    //o compilador Dart cria uma lista imutável neste caso
    nomes[0] = 'Ana Maria';
}
```

No que diz respeito à manipulação de **null**, queremos saber o seguinte:

- a variável que referencia a lista pode referenciar null
- a lista referenciada pode conter null

Dart possui construções para que lidamos com isso em tempo de compilação, evitando a famigerada “NullPointerException” que acontece em tempo de execução e costuma ser muito custosa. Veja os exemplos.

```

void main(List<String> arguments) {
var nomes1 = ['Ana', 'Pedro'];
//não pode, o tipo já é List<String>
//Strings obrigatórias, não pode null
//erro em tempo de compilação
//nomes1.add(null);
//aqui o tipo é List<dynamic>
var nomes2 = [];
nomes2.add('Ana');
//vale colocar null
nomes2.add(null);

//sem inferência não pode
List <String> nomes3 = [];
//erro em tempo de compilação
//nomes3.add(null);

//a menos que digamos que pode explicitamente
//String? é algo como "opcional", pode ser String ou null
List<String?> nomes4 = [];
nomes4.add(null);

//com type annotation
//dá na mesma, não pode
//var nomes5 = <String> [null];

//a menos que digamos explicitamente que pode, com ?
var nomes6 = <String?> [null];

//observe que aqui é diferente
//a variável lista pode ser null
//mas a lista não pode conter null
List<String>? podeSerNullMasNaoEh = [];
//List<String>
print(podeSerNullMasNaoEh.runtimeType);
//não pode
//podeSerNullMasNaoEh.add(null);

List<String>? podeSerNullEEh = null;
//Null
print(podeSerNullEEh.runtimeType);

//aqui, a variável pode ser null e a lista pode conter null
List <String?>? podeSerEConterNull1;
//null implicitamente
print(podeSerEConterNull1);
//contém null e uma String. int não pode
List <String?>? podeSerEConterNull2 = [null, 'Ana'];
print(podeSerEConterNull2);
}

```

2.3 Tuplas Tuplas são coleções ordenadas imutáveis. São criadas utilizando-se a notação `()`.

```
void main(List<String> arguments) {
    var tupla = ('Ana', 18, true);
    print(tupla);
    // (String, int, bool) é o tipo
    print(tupla.runtimeType);
    // podemos acessar os elementos assim
    // contagem começa do 1
    print(tupla.$1);
    print(tupla.$2);
    print(tupla.$3);
    // erro em tempo de compilação
    // print(tupla.$4);
}
```

2.4 Conjuntos (Sets) Conjuntos são coleções que não admitem elementos repetidos. A construção de um conjunto deve ser feita utilizando-se o operador `{ }`.

```
void main(List<String> arguments) {
    // ok
    var nomes = {'Ana', 'João'};
    print(nomes);
    // _Set<String>
    print(nomes.runtimeType);

    // ok também, mas vai conter somente um "Brasil"
    var países = {'Brasil', 'Brasil'};
    print(países);
}
```

Se construirmos um objeto usando `{ }` e ele possuir pelo menos um elemento “comum”, ele é um conjunto. Se construirmos um objeto usando o mesmo operador e ele não contiver elemento algum, ele é um **mapa**. Estudaremos sobre mapas adiante.

```

void main(List<String> arguments) {
    //tem um elemento, é um conjunto _Set<int>
    var numeros = {1};
    print(numeros.runtimeType);
    //vazio, é um _Map <dynamic, dynamic>
    //estudaremos mais sobre mapas adiante
    var nomes = {};
    print(nomes.runtimeType);

    //aqui estamos dizendo que ele contém Strings
    //com type annotation
    //é um conjunto _Set <String>
    var paises = <String> {};
    print(paises.runtimeType);

    //esse é um mapa com type annotation _Map<int, bool>
    var maiores = <int, bool> {};
    print(maiores.runtimeType);
}

```

Não podemos “indexar” um conjunto.

```

void main(List<String> arguments) {
    var nomes = {'Ana', 'João'};
    //erro em tempo de compilação
    print(nomes[0]);
    //também não dá, não existe esse operador
    print(nomes{0});
}

```

Podemos acessar um elemento com o método **elementAt**. A primeira posição válida também é zero, assim como ocorre com as listas. Além disso, também podemos iterar sobre um conjunto.

```

void main(List<String> arguments) {
    var nomes = {'Ana', 'João'};
    print(nomes.elementAt(0));

    //while comum (for e do/while também vale)
    for (int i = 0; i < nomes.length; i++){
        print(nomes.elementAt(i));
    }

    //for each
    for (final nome in nomes){
        print(nome);
    }
}

```

```
//RangeError
print(nomes.elementAt(2));
}
```

Matematicamente, as operações mais comuns envolvendo conjuntos são

- **união.** Se A e B são conjuntos, a sua união é um conjunto contendo todos os elementos contidos em A e todos os elementos contidos em B.
- **interseção.** Se A e B são conjuntos, a sua interseção é um conjunto contendo todos os elementos contidos em A e todos os elementos contidos em B.
- **diferença.** Se A e B são conjuntos, $A \setminus B$ denota a diferença de A em relação a B e ela é o conjunto que contém todos os elementos de A que não são elementos de B. $B \setminus A$ é o conjunto que contém todos os elementos de B que não estão contidos em A.

```
void main(List<String> arguments) {
    var A = {1, 2, 3, 4, 5, 6};
    var B = {1, 3, 7};
    //1, 2, 3, 4, 5, 6, 7
    print(A.union(B));
    //1
    print(A.intersection(B));
    //2, 4, 5, 6
    print(A.difference(B));
    //7
    print(B.difference(A));
    //conjunto vazio
    print(A.difference(A));
}
```

Exercício. Complete o seguinte programa. Ele deve mostrar

- Todos os países em que se fala português e todos os países da Europa.
- Todos os países em que se fala português e que são europeus.
- Todos os países em que se fala português e que não são europeus.
- Todos os países exceto aqueles em que se fala português e que são europeus (simultaneamente).

```
void main(List<String> arguments) {
var portugues = {'Brasil', 'Portugal'};
var europa = {'Alemanha', 'Portugal', 'Espanha'};
}
```

2.5 Mapas Um mapa é uma coleção de pares chave/valor. Em outras linguagens, são muitas vezes chamados de dicionários.

Nota. Tanto chave quanto valor podem ser de tipos quaisquer.

```
void main(List<String> arguments) {
  var pessoa = {
    'nome': 'Ana',
    'idade': 22,
    'altura': 1.8
  };
  // Map<String, Object>
  print(pessoa.runtimeType);

  var lembretes = {
    1: 'comprar café',
    2: 'ver um filme'
  };
  // Map<int, String>
  print(lembretes.runtimeType);
}
```

Um mapa não pode conter chaves iguais, embora isso não seja um erro em tempo de compilação/execução. Se digitarmos um valor literal de mapa com duas chaves iguais, somente um deles prevalecerá, em geral, o último.

```
void main(List<String> arguments) {
  var pessoa = {
    'nome': 'Pedro',
    'nome': 'Ana'
  };
  print(pessoa);
}
```

Mapas podem ser declarados sem utilizar a inferência de tipo e também utilizando type annotations.

```

void main(List<String> arguments) {
    //sem inferência de tipo
    Map <String, Object> pessoa = {
        'nome': 'Pedro',
        'idade': 22
    };
    print(pessoa);

    //com type annotation
    var pessoa2 = <String, Object> {
        'nome': 'Ana',
        'idade': 22
    };
    print(pessoa2);
}

```

Utilizamos o operador [] para acessar os elementos de um mapa. Como “índice”, utilizamos a chave de interesse. O resultado é o valor associado a ela. O acesso a um mapa utilizando uma chave que ele não possui produz o valor null.

```

void main(List<String> arguments) {
    var pessoa = {
        'nome': 'Pedro',
        'idade': 22
    };
    //não dá
    //print(pessoa.nome);
    //ok
    print(pessoa['nome']);
    //ok
    print(pessoa['idade']);
    //null
    print(pessoa['altura']);
}

```

Suponha que tenhamos um mapa cujos valores são do tipo “dynamic”. Qual o resultado do seguinte programa?

```

void main(List<String> arguments) {
  var pessoa = <String, dynamic>{
    'nome': 'Pedro',
    'idade': 22
  };
  //String, tipo conhecido em tempo de execução
  print(pessoa['nome'].runtimeType);
}

```

Observe, entretanto, que o ambiente em tempo de desenvolvimento não conhece o tipo do objeto e não consegue nos ajudar completando quando digitamos o operador ponto. Digite a palavra nome e o símbolo ponto logo a seguir. Observe que a lista mostra apenas métodos da classe Object. Não aparece nenhum da classe String.

bin > colecoes.dart > main

```

Run | Debug
1 void main(List<String> arguments) {
2   var pessoa = <String, dynamic>{
3     'nome': 'Pedro',
4     'idade': 22
5   };
6   //String, tipo conhecido em tempo de execução
7   print(pessoa['nome'].runtimeType);
8
9   var nome = pessoa['nome'];
10  //é String mas em tempo de compilação não se sabe disso
11  nome.

```

The screenshot shows a code editor with the following code:

```

void main(List<String> arguments) {
  var pessoa = <String, dynamic>{
    'nome': 'Pedro',
    'idade': 22
  };
  //String, tipo conhecido em tempo de execução
  print(pessoa['nome'].runtimeType);

  var nome = pessoa['nome'];
  //é String mas em tempo de compilação não se sabe disso
  nome.

```

A completion dropdown is open at the line `nome.`. The list includes:

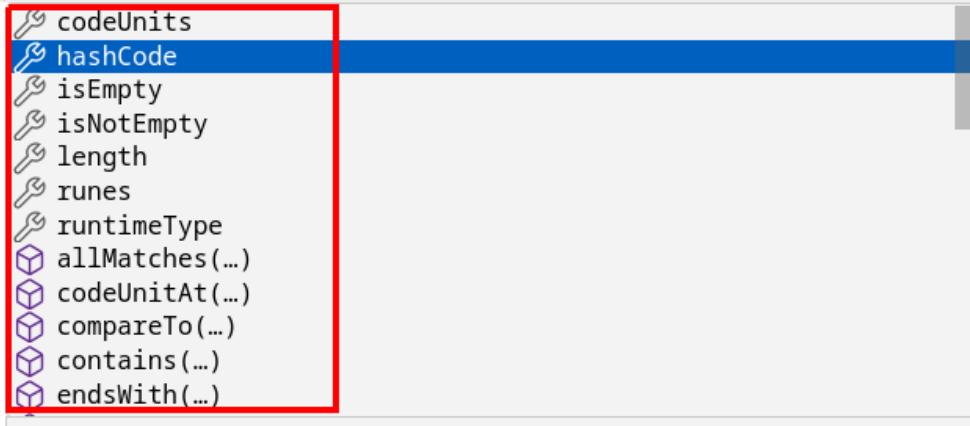
- hashCode
- runtimeType
- toString()
- isDefinedAndNotNull
- isNull
- isUndefined
- isUndefinedOrNull
- noSuchMethod(...)
- switch expression

Podemos usar o operador **as** neste caso. Observe que a lista inclui os métodos da classe String.

bin > colecoes.dart > main

```

Run | Debug
1 void main(List<String> arguments) {
2     var pessoa = <String, dynamic>{
3         'nome': 'Pedro',
4         'idade': 22
5     };
6     //String, tipo conhecido em tempo de execução
7     print(pessoa['nome'].runtimeType);
8
9     var nome = pessoa['nome'] as String;
10    //agora sim, falamos o tipo explicitamente
11    nome,
12
13 }
```



Observe que precisamos ter cuidado ao utilizar o operador **as**. Seu uso pode causar erros em tempo de execução.

```

void main(List<String> arguments) {
    var pessoa = <String, dynamic>{
        'nome': 'Pedro',
        'idade': 22
    };
    //String, tipo conhecido em tempo de execução
    print(pessoa['nome'].runtimeType);

    //agora sim, falamos o tipo explicitamente
    var nome = pessoa['nome'] as String;

    //errado mas o compilador não sabe
    var idade = pessoa['idade'] as String;

    //int não tem toUpperCase
    //erro em tempo de execução
    print(idade.toUpperCase());
}
```

Não podemos iterar sobre um mapa, utilizando um for each, por exemplo. Mas podemos iterar sobre as chaves de um mapa. Para obter as chaves, usamos o método **keys**. Também podemos iterar sobre os valores diretamente, usando o método **values** para obter a coleção de valores. Além disso, podemos iterar sobre os pares chave/valor tendo acesso a cada um deles a cada iteração, utilizando o método **entries**.

```
void main(List<String> arguments) {
    var pessoa = <String, dynamic>{
        'nome': 'Pedro',
        'idade': 22
    };

    //errado
    //for (var prop in pessoa){
    //print(prop);
    //}

    //Iterable de String
    var chaves = pessoa.keys;
    print(chaves.runtimeType);

    for (final propriedade in pessoa.keys){
        print(pessoa[propriedade]);
    }

    //Iterable de dynamic
    var valores = pessoa.values;
    print(valores.runtimeType);
    for (final valor in valores){
        print(valor);
    }

    //Iterable de
    var entries = pessoa.entries;
    //MappedIterable<String, MapEntry<String, dynamic>>
    print(entries.runtimeType);
    for (final entry in pessoa.entries){
        print(entry);
        print(entry.key);
        print(entry.value);
    }
}
```

Exercício. Escreva um programa que permita ao usuário armazenar a sua lista de contatos.

- Contatos possuem um nome e um número de telefone
- Deve ser possível realizar as quatro operações básicas de um CRUD
- O armazenamento deve ser feito em um mapa
- Deve haver um menu: 1-C 2-R 3-U 4-D 5-Sair.
- Um par chave/valor tem como chave o nome do contato e seu valor associado é o seu número.

2.6 Coleções de coleções Uma coleção pode armazenar outras coleções. No exemplo a seguir, temos uma lista de filmes. Cada filme tem

- título
- gênero
- notas

Cada filme é representado por um mapa. Além disso, a sua coleção de notas é uma lista de inteiros.

```
import 'dart:io';
void main(List<String> arguments) {
    //lista de mapas
    //cada item na lista é um mapa com chave String e valor dynamic
    var filmes = < Map<String, dynamic> > [];
    print(filmes.runtimeType);
    print("Titulo?");
    String? titulo = stdin.readLineSync();
    print("Genero?");
    String? genero = stdin.readLineSync();
    var notas = [5, 5];
    filmes.add({'titulo': titulo, 'genero': genero, 'notas': notas});
    print(filmes);
}
```

2.7 Collection-if Podemos fazer a adição condicional de elementos a uma lista. Observe.

```

void main(List<String> arguments) {
    var idadePedro = 17;
    var idadeCristina = 18;
    var maioresDeIdade = [
        'Ana',
        'João',
        if (idadePedro >= 18) 'Pedro',
        if (idadeCristina >= 18) 'Cristina'
    ];
    //Ana, João e Cristina
    print(maioresDeIdade);
}

```

2.8 Collection-for Essa construção é semelhante. Observe. Aqui estamos adicionando todos os itens de uma coleção a outra já no momento em que ela é inicializada.

```

void main(List<String> arguments) {
    var nomes1 = ['Ana', 'Pedro'];
    var nomes2 = [
        'Cristina',
        for (var nome in nomes1)
            nome
    ];
    //Cristina, Ana, Pedro
    print(nomes2);
}

```

2.9 Operador spread (spread significa algo como espalhar) O operador spread nos permite extrair os elementos de uma coleção. Ele representa todos os elementos dela, porém fora dela. Poderíamos tentar adicionar os elementos de uma lista a outra da seguinte forma.

```

void main(List<String> arguments) {
    var nomes1 = ['Ana', 'Pedro'];
    var nomes2 = [
        'Cristina',
        nomes1
    ];
    //Cristina, [Ana, Pedro]
    print(nomes2);
}

```

Porém, observe que o resultado obtido é uma lista que contém o nome Cristina e uma lista contendo os nomes Ana e Pedro. Não é exatamente o que desejamos. Precisamos extrair Ana e Pedro da lista antes de adicionar. Precisamos “espalhar” os elementos da lista. A

expressão resultante representa os elementos fora da lista. Aí podemos fazer a adição. Observe.

```
void main(List<String> arguments) {
  var nomes1 = ['Ana', 'Pedro'];
  var nomes2 = [
    'Cristina',
    ...nomes1 //operador spread
  ];
  //Cristina, Ana, Pedro
  print(nomes2);
}
```

2.10 Cópia de coleções No exemplo a seguir, temos a intenção de fazer uma cópia da lista. Observe, entretanto, que não é isso exatamente o que acontece. Depois da atribuição, ambas as variáveis fazem referência ao mesmo objeto.

```
void main(List<String> arguments) {
  var nomes = ['Ana', 'Pedro'];
  var copia = nomes;
  //alterando a copia, também alteramos a original
  //porque na verdade somente há uma lista
  //e duas variáveis fazendo referência a ela
  copia[0] = "Ana Maria";
  print(nomes);
  print(copia);
}
```

Uma cópia de fato pode ser criada de diferentes formas. Podemos usar

- Collection-for
- operador spread

```
void main(List<String> arguments) {
    var nomes = ['Ana', 'Pedro'];
    var copiaComCollectionFor = [
        for (var nome in nomes)
            nome
    ];
    copiaComCollectionFor[0] = 'Ana Maria';
    var copiaComOperadorSpread = [
        ...nomes
    ];
    copiaComOperadorSpread[0] = 'Cristina';

    // [Ana, Pedro]
    print(nomes);
    // [Ana Maria, Pedro]
    print(copiaComCollectionFor);
    // [Cristina, Pedro]
    print(copiaComOperadorSpread);
}
```

Referências

Dart programming language | Dart. Google, 2023. Disponível em <<https://dart.dev/>>. Acesso em agosto de 2023.

1 Introdução

Flutter é um framework criado pelo Google que permite a criação de aplicações “**bonitas**” para múltiplas plataformas. Veja algumas características.

Veja a sua página oficial.

<https://flutter.dev/>

A partir de um único “codebase”, podemos gerar aplicações para múltiplas plataformas como

- Linux
- Windows
- MacOS
- Android
- iOS
- Web

Compilação **nativa** para arquiteturas **ARM** e **Intel**. Também compila para **Javascript**.

Interfaces gráficas construídas utilizando-se **Widgets**. Um Widget é um componente visual como um botão, um campo textual, um gerenciador de layout etc.

Cada **Widget** é implementado como uma classe **Dart**.

Flutter possui um vasto catálogo de Widgets. Ele oferece pode ser encontrado a seguir

<https://docs.flutter.dev/ui/widgets>

É uma boa ideia manter esse link na sua barra de favoritos enquanto desenvolve.

Também podemos criar nossos próprios Widgets.

Neste material, vamos desenvolver uma aplicação que exibe fotos obtidas por meio de requisições HTTP. Ela nos permitirá aprender o uso básico de Widgets Flutter bem como alguns recursos interessantes, tais como o processamento assíncrono funciona neste ambiente.

2 Desenvolvimento

(Workspace, novo projeto Flutter e VS Code) Comece criando uma pasta para desempenhar o papel de Workspace. Ou seja, uma pasta que abriga subpastas, cada qual representando um projeto Flutter. No Windows, você pode usar algo assim:

```
C:\Users\seuUsuario\Documents\dev\
```

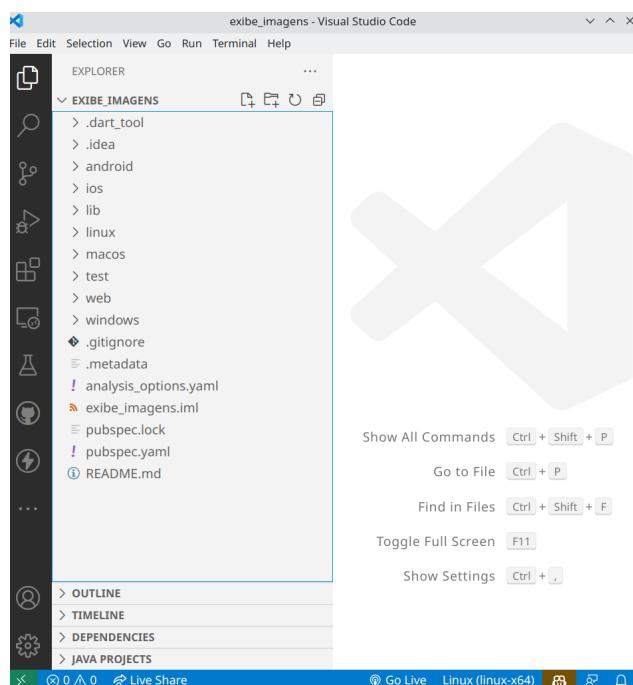
A seguir, abra um terminal e vincule-o ao diretório que acabou de criar com

```
C:\Users\seuUsuario\Documents\dev
```

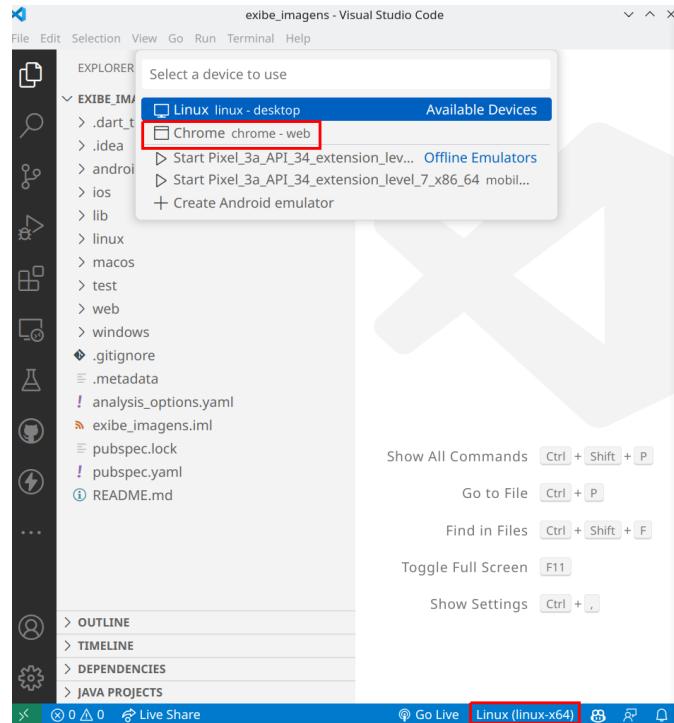
Crie o projeto Flutter com

```
flutter create exibe_imagens
```

Abra o VS Code e clique em **File > Open Folder**. Navegue até a pasta recém-criada. Veja o resultado esperado.



Observe que no canto inferior direito há a possibilidade de escolher a plataforma em que a aplicação será colocada em execução. No exemplo a seguir, o padrão selecionado é Linux. Dependendo de seu ambiente, o padrão pode ser diferente. Clique sobre esta opção e escolha o Chrome para executarmos a aplicação na Web.



No VS Code, clique em **Terminal >> New Terminal**. Use

flutter run

para executar a aplicação. É possível que você tenha de selecionar a plataforma alvo. Digite o número associado ao Google Chrome.

```
○ imagens$ flutter run
Connected devices:
Linux (desktop) • linux • linux-x64      • Ubuntu 22.04.3 LTS
6.2.0-26-generic
Chrome (web)     • chrome • web-javascript • Google Chrome
116.0.5845.96
[1]: Linux (linux)
[2]: Chrome (chrome)
Please choose one (or "q" to quit): [ ]
```

A aplicação resultante apenas mostra um botão que, quando clicado, incrementa um contador exibido na parte central da tela.



Observe que temos uma pasta chamada **lib** e que, dentro dela, há um arquivo chamado **main.dart**. O código Dart da aplicação exemplo se encontra ali. O ponto de partida da aplicação se encontra neste arquivo. Seu nome **deve ser main.dart**.

```

main.dart - exibe_imagens - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
  EXIBE_IMAGENS
    > .dart_tool
    > .idea
    < .vscode
      settings.json
    > android
    > build
    > ios
    < lib
      main.dart
    > linux
    > macos
    > test
    > web
    > windows
    < .gitignore
  metadata
main.dart
lib/main.dart
  MyApp
    build
    main.dart
      // THIS WORKS FOR code too, not just values: MOST
      // code changes can be
      // tested with just a hot reload.
      colorScheme: ColorScheme.fromSeed(seedColor:
        Colors.deepPurple),
      useMaterial3: true,
      ), // ThemeData
      home: const MyHomePage(title: 'Flutter Demo Home
      Page'),
    ); // MaterialApp
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});

  // This widget is the home page of your application. It
  // is stateful, meaning
  // that it has a State object (defined below) that
  // contains fields that affect
  // how it looks.

  // This class is the configuration for the state. It
  // holds the values (in this
  // case the title) provided by the parent (in this case
  // the App widget) and
  // used by the build method of the State. Fields in a
  // Widget subclass are
  // always marked "final".
  final String title;
}

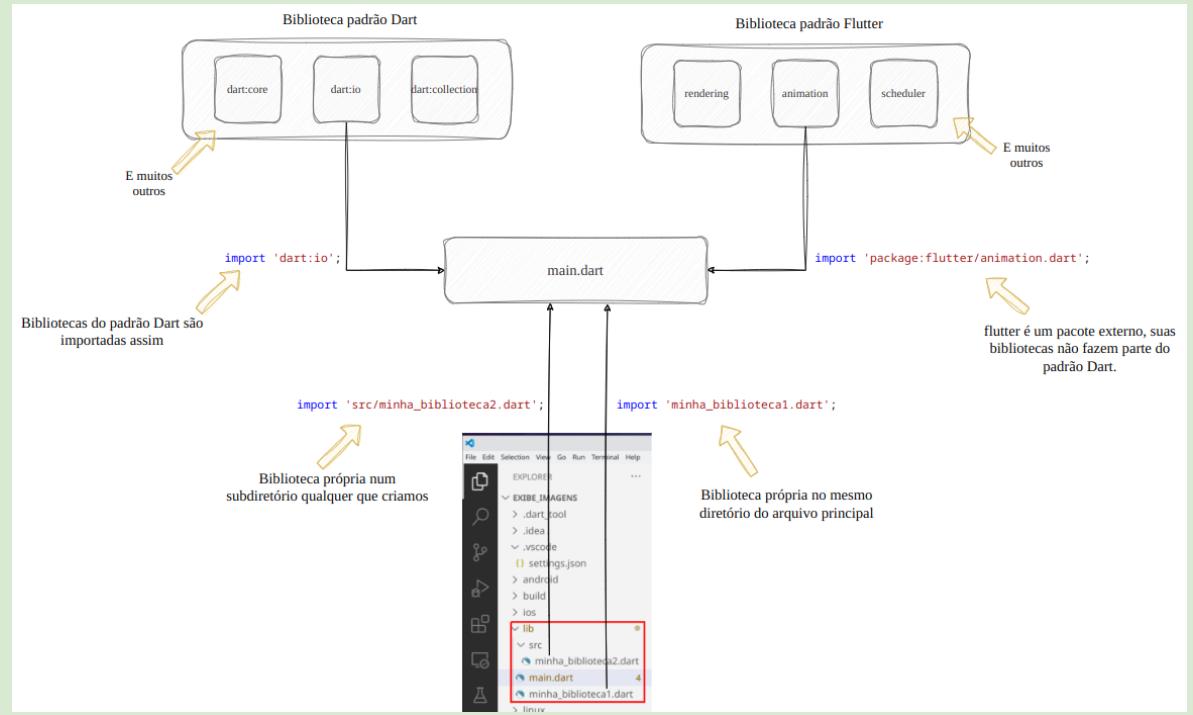
```

Apague todo o conteúdo do arquivo **main.dart** para começarmos a trabalhar em nossa aplicação do zero.

(Exibindo um componente textual) Vamos começar apenas exibindo um componente textual. Para isso, precisamos

- importar o pacote que contém o Widget desejado
- escrever uma função main, que é o ponto em que a aplicação começa
- criar o Widget capaz de exibir conteúdo textual
- exibir o Widget na tela

Nota. Há três formas de se importar conteúdo num arquivo dart. Para entender cada um, observe a seguinte figura.



Assim, podemos importar bibliotecas do ambiente padrão dart, bibliotecas de pacotes externos - como o flutter - e bibliotecas próprias que criamos em qualquer diretório de nossa aplicação que seja subdiretório de lib.

Para utilizar componentes visuais do Flutter, vamos importar o pacote material.

Nota. O pacote material traz a implementação da especificação conhecida como **Material Design**, do Google. Trata-se de um sistema de design com instruções para o desenvolvimento envolvendo UX, a implementação de componentes visuais, combinação de cores, feedback visual mediante interação do usuário etc. Para saber mais, visite

<https://m3.material.io/>

Veja o import. Estamos no arquivo **lib/main.dart**.

```
import 'package:flutter/material.dart';
```

A seguir, escrevemos a função main, como de costume.

```
import 'package:flutter/material.dart';

void main() {

}
```

No momento, nosso objetivo é apenas exibir um componente textual. Vamos construir um Widget do tipo **MaterialApp**, para começar.

Nota. O Widget **MaterialApp** se encarrega de realizar configurações básicas, como a possibilidade de **navegação entre diferentes telas**. Seu uso é bastante comum.

```
import 'package:flutter/material.dart';

void main() {
    var app = MaterialApp();
}
```

Observe que há uma sugestão de aplicar o modificador **const** à chamada do construtor. Isso acontece pois esse é um construtor que foi declarado como const. De maneira simplificada, isso quer dizer que ele constrói um objeto imutável. Quando aplicamos const à chamada de um construtor, o compilador nos entrega código otimizado.

```
import 'package:flutter/material.dart';

void main() {
    var app = const MaterialApp();
}
```

Visite a documentação do Widget **MaterialApp** a seguir.

<https://api.flutter.dev/flutter/material/MaterialApp-class.html>

Observe que seu construtor possui um parâmetro nomeado **home**.

Flutter > material > MaterialApp class

Search API Docs

CLASSES

- AboutDialog
- AboutListTile
- AbsorbPointer
- Accumulator
- Action
- ActionChip
- ActionDispatcher
- ActionIconTheme
- ActionIconThemeData
- ActionListener
- Actions
- ActivateAction
- ActivateIntent
- AdaptiveTextSelectionT...
- ...

Constructors

MaterialApp(*{Key? key, GlobalKey<NavigatorState>? navigatorKey, GlobalKey<ScaffoldMessengerState>? scaffoldMessengerKey, Widget? home, Map<String, WidgetBuilder> routes = const <String, WidgetBuilder>{}, String? initialRoute, RouteFactory? onGenerateRoute, InitialRouteListFactory? onGenerateInitialRoutes, RouteFactory? onUnknownRoute, List<NavigatorObserver> navigatorObservers = const <NavigatorObserver>[], TransitionBuilder? builder, String title = '', GenerateAppTitle? onGenerateTitle, Color? color, ThemeData? theme, ThemeData? darkTheme, ThemeData? highContrastTheme, ThemeData? highContrastDarkTheme, ThemeMode? themeMode = ThemeMode.system, Duration themeAnimationDuration = kThemeAnimationDuration, Curve themeAnimationCurve = Curves.linear, Locale? locale, Iterable<LocalizationsDelegate>? localizationsDelegates, LocaleListResolutionCallback? localeListResolutionCallback, LocaleResolutionCallback? localeResolutionCallback, Iterable<Locale> supportedLocales = const <Locale>[Locale('en', 'US')], bool debugShowMaterialGrid = false, bool showPerformanceOverlay = false, bool checkerboardRasterCachelImages = false, bool checkerboardOffscreenLayers = false, bool showSemanticsDebugger = false, bool debugShowCheckedModeBanner = true, Map<ShortcutActivator, Intent?>? shortcuts, Map<Type, Action<Intent>>? actions, String? restorationScopeld, ScrollBehavior? scrollBehavior, @Deprecated('Remove this parameter as it is now ignored. 'MaterialApp never introduces its own MediaQuery; the View widget takes care of that. 'This feature was deprecated after v3.7.0-29.0.pre.') bool useInheritedMediaQuery = false}]*)

Creates a MaterialApp.
const

Desejamos associar um novo Widget a esse parâmetro. Ele será exibido na região principal da tela. Neste momento, vamos apenas utilizar um Widget textual.

Nota. Também poderíamos aplicar **const** à chamada do construtor Text. Isso seria redundante pelo fato de ele estar sendo utilizado como parâmetro de MaterialApp e por já termos aplicado const a esse construtor.

```
import 'package:flutter/material.dart';

void main() {
  var app = const MaterialApp(
    home: Text('Hello, Dart')
  );
}
```

Nota. Observe a forma como ajustamos o código. Salve o arquivo. É provável que ele seja reorganizado da seguinte forma.

```
import 'package:flutter/material.dart';

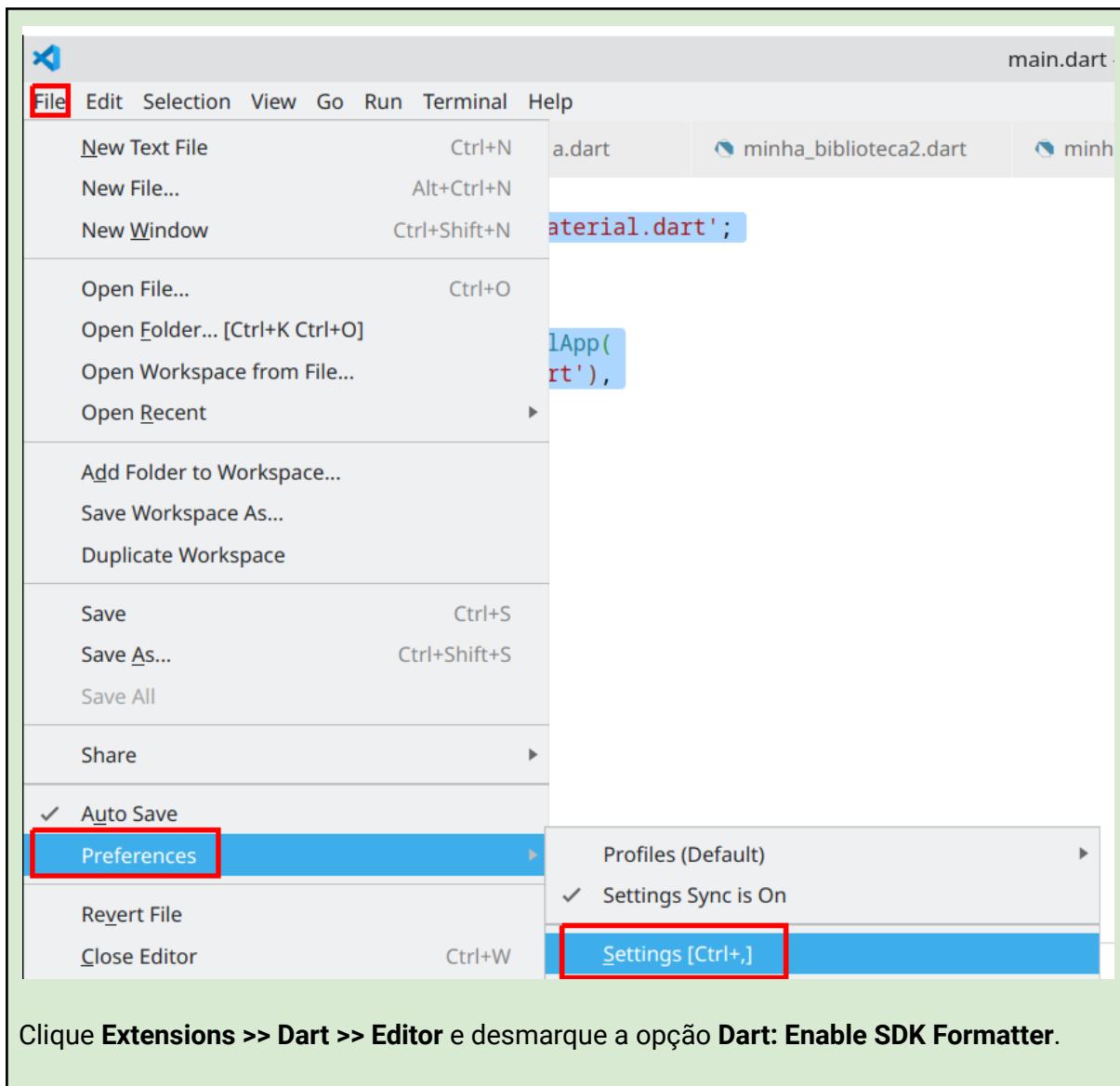
void main() {
    var app = const MaterialApp(home: Text('Hello, Dart'));
}
```

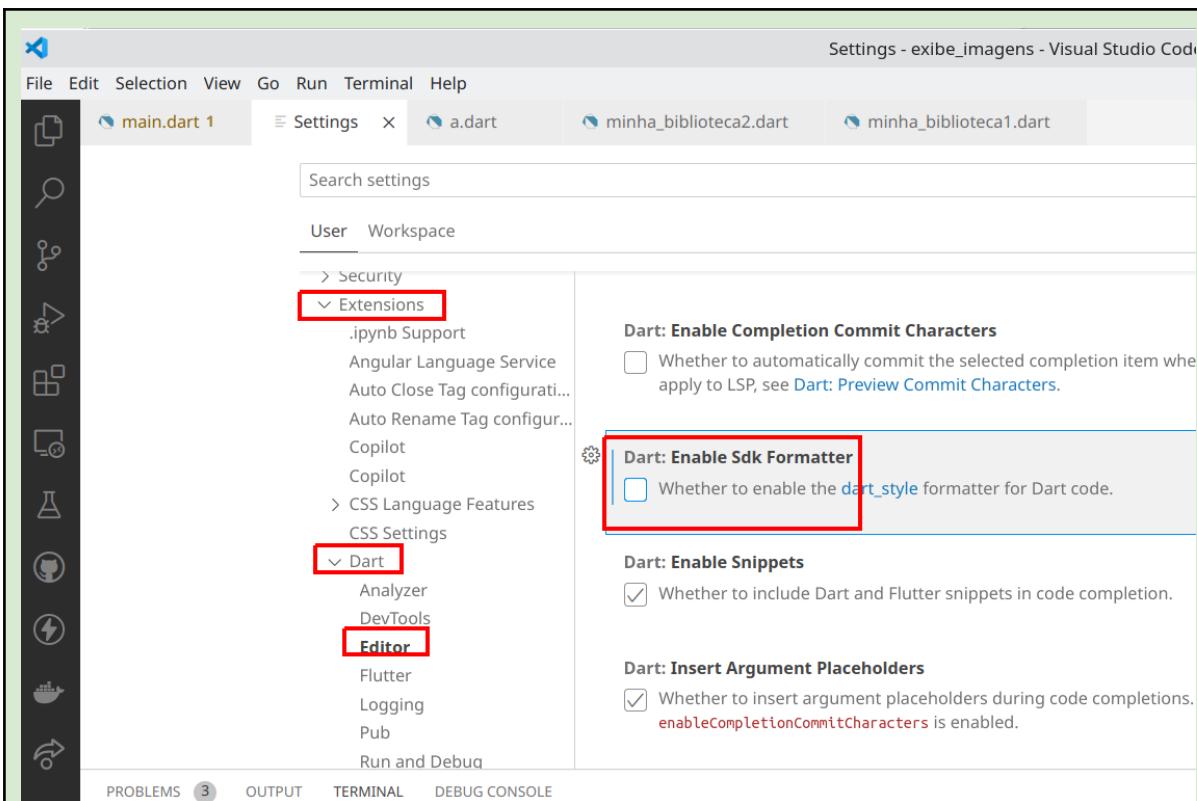
Se desejar evitar esse comportamento, você pode adicionar uma vírgula no final da definição do último componente. Observe.

```
import 'package:flutter/material.dart';

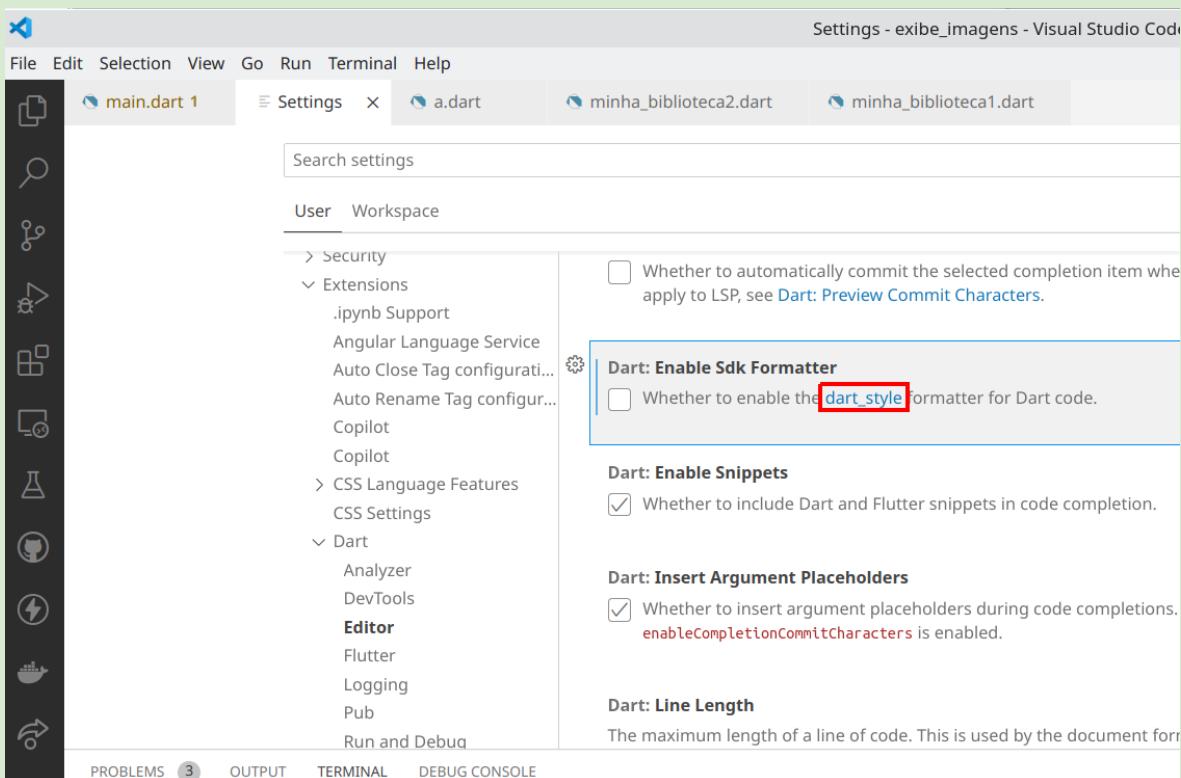
void main() {
    var app = const MaterialApp(
        home: Text('Hello, Dart'),
    );
}
```

Salve o arquivo novamente e repare que o código permanece da forma como você deixou. Se preferir, também é possível desabilitar esse funcionamento utilizando uma opção da extensão Dart, embora ela também envolva outros aspectos. No VS Code, clique **File >> Preferences Settings**.

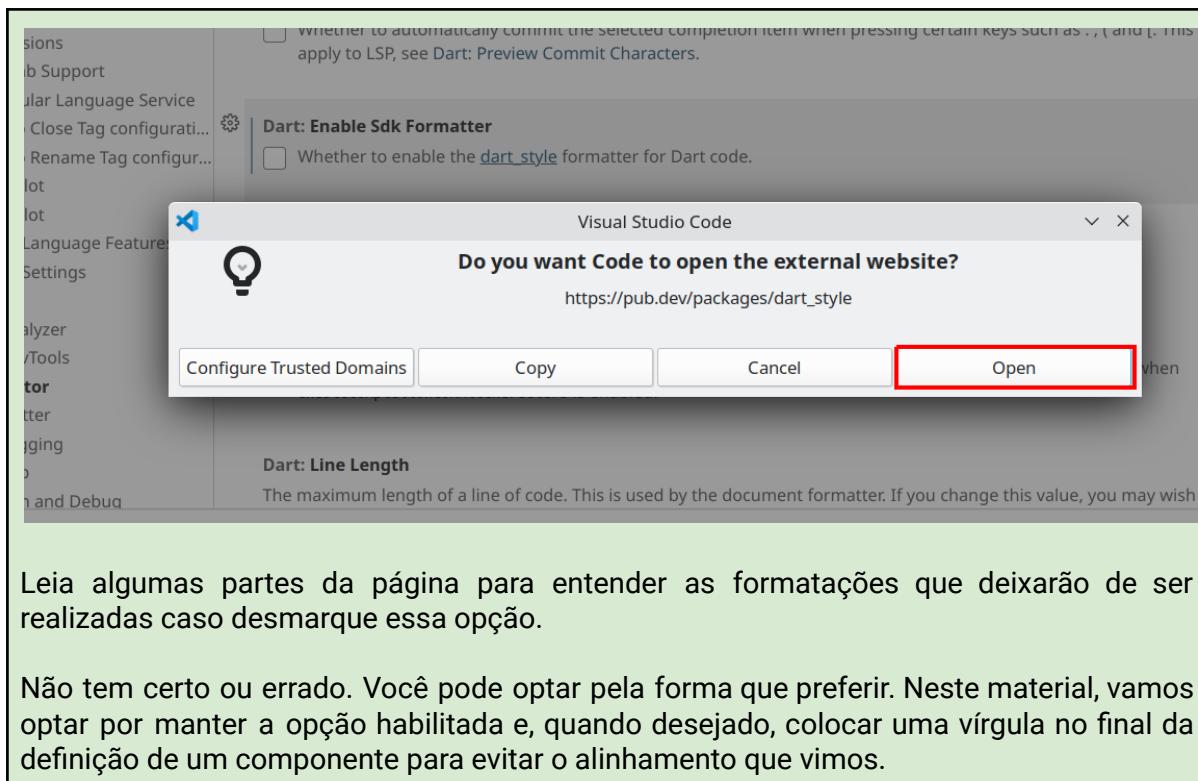




Observe, entretanto, que desmarcar esta opção, envolve diversos outros tipos de ajustes feitos automaticamente e que estão de acordo com as recomendações oficiais Dart. Ainda nesta tela, clique **`dart_style`**.



Se necessário, clique em **Open**.



Leia algumas partes da página para entender as formatações que deixarão de ser realizadas caso desmarque essa opção.

Não tem certo ou errado. Você pode optar pela forma que preferir. Neste material, vamos optar por manter a opção habilitada e, quando desejado, colocar uma vírgula no final da definição de um componente para evitar o alinhamento que vimos.

Por enquanto, o código está assim.

```
import 'package:flutter/material.dart';

void main() {
    var app = const MaterialApp(
        home: Text('Hello, Dart'),
    );
}
```

A simples definição da função main não é suficiente para que possamos colocar a aplicação em execução. Aplicações Flutter devem ser colocadas em execução de um jeito especial: **utilizando uma função chamada runApp**. Na função main, vamos chamar a função runApp entregando a ela o Widget “raiz” da aplicação.

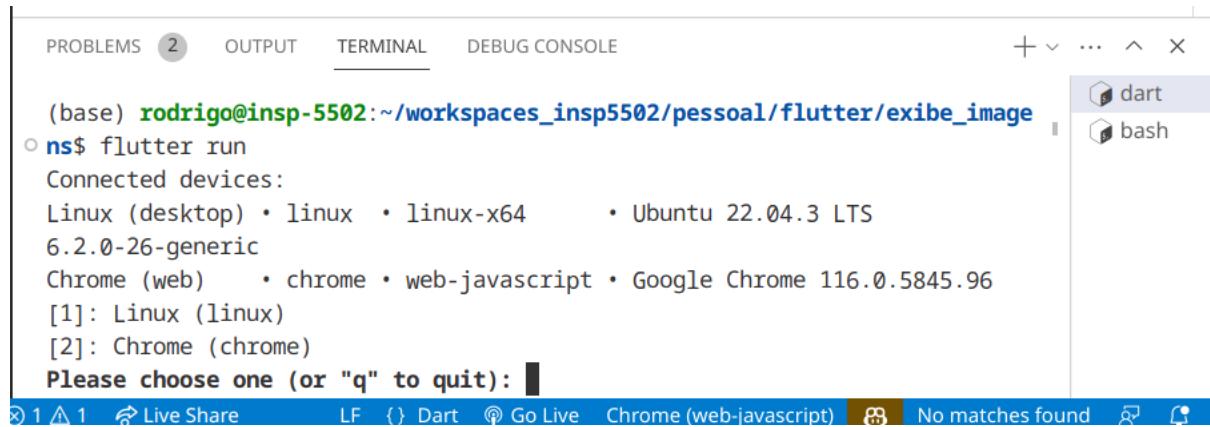
```
import 'package:flutter/material.dart';

void main() {
    var app = const MaterialApp(
        home: Text('Hello, Dart'),
    );
    runApp(app);
}
```

No VS Code, clique em **Terminal >> New Terminal** para abrir um terminal interno, caso ainda não possua um. Use

flutter run

para executar a aplicação. No terminal, escolha a plataforma para execução digitando seu número associado. No exemplo a seguir, digitamos **2** para executar a aplicação no Google Chrome.



The screenshot shows the VS Code interface with the terminal tab selected. The terminal window displays the following text:

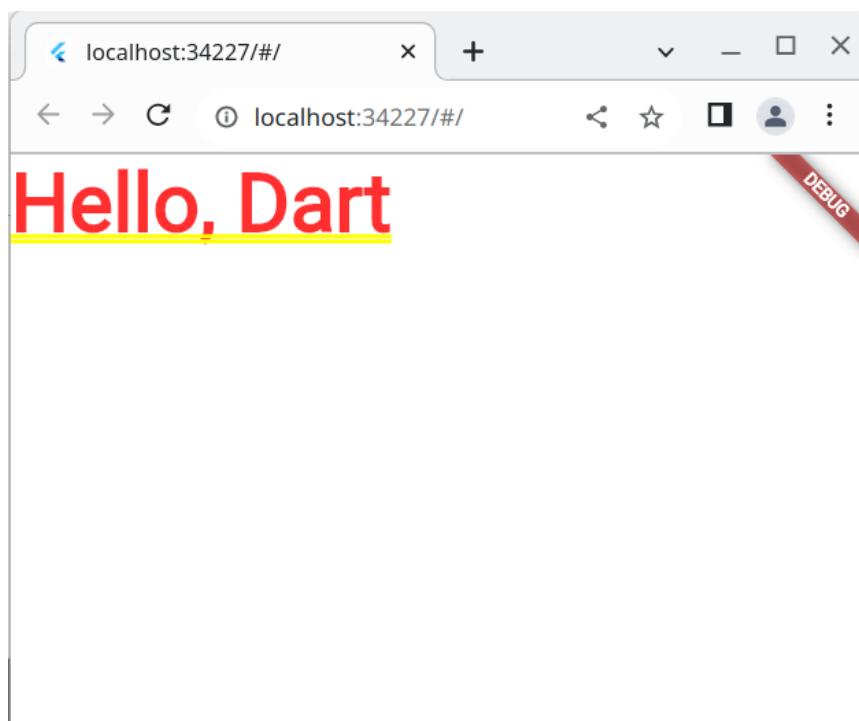
```
(base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/flutter/exibe_image
$ flutter run
Connected devices:
Linux (desktop) • linux • linux-x64      • Ubuntu 22.04.3 LTS
6.2.0-26-generic
Chrome (web)     • chrome • web-javascript • Google Chrome 116.0.5845.96
[1]: Linux (linux)
[2]: Chrome (chrome)

Please choose one (or "q" to quit):
```

The sidebar on the right shows two options: 'dart' and 'bash', with 'dart' currently selected. The bottom of the terminal window shows various status icons and text.

Nota. É natural que este processo demore algo entre 30 segundos a 1 minuto. Ou talvez um pouco mais.

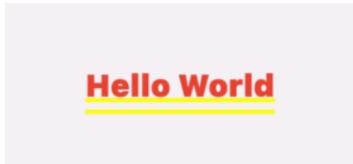
A esperança é que uma aba do navegador seja aberta e que a aplicação seja exibida. Observe.



Percebeu como o estilo do texto, incluindo as cores, não é lá dos melhores? Isso é de propósito. Veja o que a documentação da classe `MaterialApp` fala sobre isso.

Why is my app's text red with yellow underlines?

`Text` widgets that lack a `Material` ancestor will be rendered with an ugly red/yellow text style.



The typical fix is to give the widget a `Scaffold` ancestor. The `Scaffold` creates a `Material` widget that defines its default text style.

```
const MaterialApp(
  title: 'Material App',
  home: Scaffold(
    body: Center(
      child: Text('Hello World'),
    ),
  ),
)
```

<https://api.flutter.dev/flutter/material/MaterialApp-class.html>

A ideia é lembrar o desenvolvedor de fazer uso adequado de componentes que apliquem temas do Material Design.

(Hot reload, hot restart e full restart) Conforme testamos nossa aplicação, estamos interessados em visualizar as modificações. Para isso, podemos usar as seguintes opções:

- **hot reload**: atualiza o trecho de código alterado direto na Dart VM e reconstrói a árvore de Widgets. **Preserva o estado da aplicação. Não a atualiza executando novamente o método main**. No terminal do VS Code em que você digitou `flutter run`, aperte a tecla **r** para utilizar.

- **hot restart**: atualiza o trecho de código alterado direto na Dart VM e **reinicia a aplicação, o que faz com o que o estado seja perdido**. No terminal do VS Code em que você digitou `flutter run`, aperte **SHIFT + r** para utilizar.

- **full restart**: reinicia a aplicação completamente. Demora mais pois o código também é recompilado. Não há atalho no teclado. Basta encerrar o processo de execução da aplicação no terminal (**CTRL + C**) e executar novamente.

Faça um teste alterando o texto da aplicação e fazendo um hot reload logo a seguir. Testaremos outras situações ao longo do desenvolvimento.

```

import 'package:flutter/material.dart';

void main() {
  var app = const MaterialApp(
    home: Text('Helloooo, Dart'),
  );
  runApp(app);
}

```

Lembre-se de apertar r no terminal em que executou o comando **flutter run**.

Nota. Há casos em que o **hot reload e/ou o hot restart podem não funcionar**.Leia mais no link a seguir. Veja alguns exemplos da documentação.

An app is killed

Hot reload can break when the app is killed. For example, if the app was in the background for too long.

Compilation errors

When a code change introduces a compilation error, hot reload generates an error message similar to:

```

Hot reload was rejected:
'/path/to/project/lib/main.dart': warning: line 16 pos 38: unbalanced '{' open:
  Widget build(BuildContext context) {
      ^
'/path/to/project/lib/main.dart': error: line 33 pos 5: unbalanced ')'
  );
  ^

```

In this situation, simply correct the errors on the specified lines of Dart code to keep using hot reload.

Enumerated types

Hot reload doesn't work when enumerated types are changed to regular classes or regular classes are changed to enumerated types.

For example:

Before the change:

```
enum Color {
    red,
    green,
    blue,
}
```

After the change:

```
class Color {
    Color(this.i, this.j);
    final int i;
    final int j;
}
```

Generic types

Hot reload won't work when generic type declarations are modified. For example, the following won't work:

Before the change:

```
class A<T> {
    T? i;
}
```

After the change:

```
class A<T, V> {
    T? i;
    V? v;
}
```

Native code

If you've changed native code (such as Kotlin, Java, Swift, or Objective-C), you must perform a full restart (stop and restart the app) to see the changes take effect.

Estes exemplos foram tirados da documentação no momento em que este documento está sendo escrito. Para mais detalhes, visite a página a seguir.

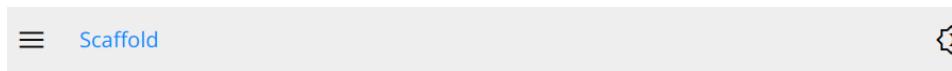
<https://docs.flutter.dev/tools/hot-reload>

(Usando um Widget Scaffold) A fim de organizar a estrutura da aplicação, vamos fazer uso de um Widget chamado Scaffold. Veja a sua documentação a seguir.

<https://api.flutter.dev/flutter/material/Scaffold-class.html>

Nota. "Scaffold" significa algo como “**esqueleto**” ou “**andaime**”.

O código dos exemplos da documentação pode envolver conceitos ainda não vistos neste momento. Não se preocupe com ele. Mas observe que um Scaffold permite especificarmos barras inferiores (**BottomAppBar**), botões flutuantes (**FloatingActionButton**) e assim por diante. Role a página até encontrar a parte que ilustra o construtor. A seguir, ilustramos dois exemplos de Widgets que podem ser entregues ao construtor de Scaffold.



Inheritance
Object > DiagnosticableTree > Widget > StatefulWidget > Scaffold

Constructors

```
Scaffold({Key? key, PreferredSizeWidget? appBar, Widget? body, Widget?
floatingActionButton, FloatingActionButtonLocation? floatingActionButtonLocation,
FloatingActionButtonAnimator? floatingActionButtonAnimator, List<Widget>?
persistentFooterButtons, AlignmentDirectional persistentFooterAlignment =
AlignmentDirectional.centerEnd, Widget? drawer, DrawerCallback? onDrawerChanged,
Widget? endDrawer, DrawerCallback? onEndDrawerChanged, Widget?
bottomNavigationBar, Widget? bottomSheet, Color? backgroundColor, bool?
resizeToAvoidBottomInset, bool primary = true, DragStartBehavior
drawerDragStartBehavior = DragStartBehavior.start, bool extendBody = false, bool
extendBodyBehindAppBar = false, Color? drawerScrimColor, double?
drawerEdgeDragWidth, bool drawerEnableOpenDragGesture = true, bool
endDrawerEnableOpenDragGesture = true, String? restorationId})  

Creates a visual scaffold for Material Design widgets.  

const
```

Nota. O símbolo ? que aparece colado ao tipo dos parâmetros indica que eles são opcionais.

Nota. Os parâmetros que se encontram entre {} na lista de parâmetros do construtor são **parâmetros nomeados**. Eles são especificados na chamada do construtor assim: ConstrutorQualquer (nomeDoParametro: objetoDesejado). Abra o **DartPad** e faça o teste a seguir.

Não altere seu código no VS Code. Esse é apenas um teste no DartPad.

<https://dartpad.dev/>

```
//pode ser null
//e implicitamente é null, inicializado
//pelo compilador
void opcional({int? numero}) {
    print(numero);
}

//assim não funciona pois int não pode ser null
//e o valor implícito é null
// void obrigatorio({int numero}) {
//     print(numero);
// }

void obrigatorio({int numero = 5}) {
    print(numero);
}

void comParametrosSemNome(int semNome, {int? comNome}) {
    print(semNome);
    print(comNome);
}

//não funciona
// nomeados (separados por { }) têm de aparecer depois
// de todos os sem nome
// void comParametrosSemNome(int? comNome}, int semNome) {
//     print(semNome);
//     print(comNome);
// }
```

```

void main(){
    //exibe null
    opcional();
    //exibe 2
    opcional(numero: 2);
    //exibe 5
    obrigatorio();
    //exibe 10
    obrigatorio(numero: 10);
    //exibe 2 null
    comParametrosSemNome(2);
    //exibe 2 2
    comParametrosSemNome(2, comNome: 2);
}

```

Nota. Não há sobrecarga de métodos/funções em Dart. Lembre-se que Dart é a linguagem de programação que estamos utilizando para programar com o Framework Flutter, que é escrito em Dart.

Nota. Observe que o primeiro parâmetro é do tipo **Key**. Esse parâmetro está associado à atualização de estado da aplicação, o que pode envolver a substituição de Widgets. Veja o que a documentação diz.

≡ key

key property

Key? key

(final)

Controls how one widget replaces another widget in the tree.

If the `runtimeType` and `key` properties of the two widgets are `operator==`, respectively, then the new widget replaces the old widget by updating the underlying element (i.e., by calling `Element.update` with the new widget). Otherwise, the old element is removed from the tree, the new widget is inflated into an element, and the new element is inserted into the tree.

Leia mais na página a seguir.

<https://api.flutter.dev/flutter/widgets/Widget/key.html>

Exercício. Substitua o Widget Text por um Widget Scaffold da seguinte forma:

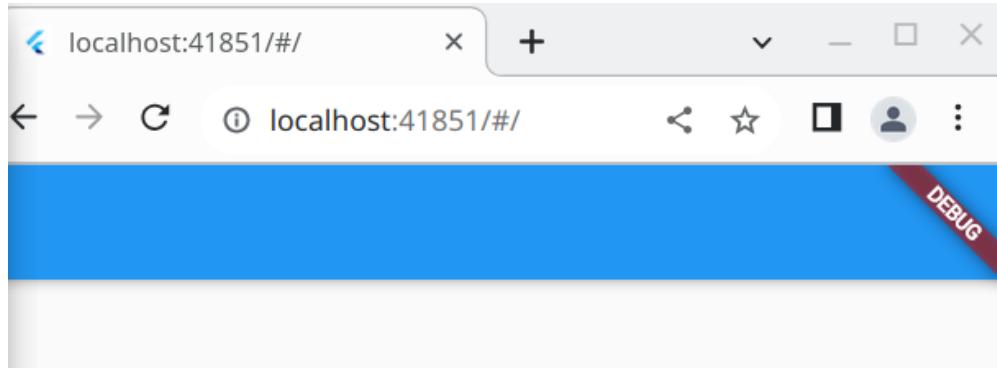
- Deve ser associado à propriedade **home** de MaterialApp.
- Deve ter um parâmetro nomeado **appBar** do tipo **AppBar**
- Observe que o construtor de AppBar não é **const**. Remova const do construtor de MaterialApp.
- Faça hot reload e veja o resultado na tela. Você deverá ver uma barra superior.
- Faça push ao repositório Git.

Veja como fica.

```
import 'package:flutter/material.dart';

void main() {
    var app = MaterialApp(
        home: Scaffold(
            appBar: AppBar(),
        ),
    );
    runApp(app);
}
```

No navegador, o resultado deve ser o seguinte.

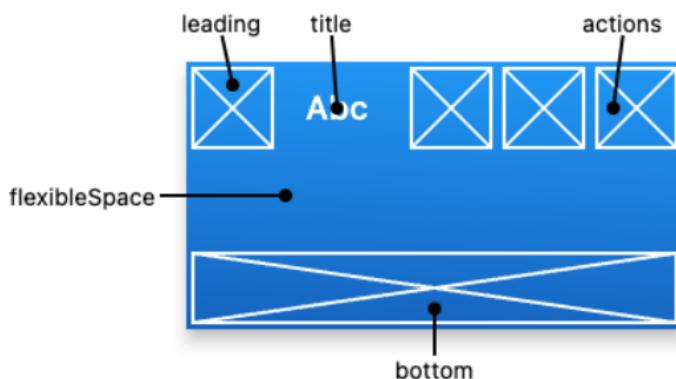


Estudemos um pouco sobre o Widget AppBar. Sua documentação se encontra em

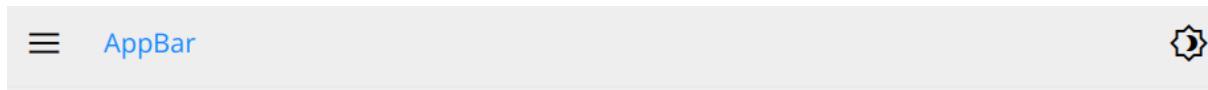
<https://api.flutter.dev/flutter/material/AppBar-class.html>

A figura a seguir, retirada da documentação, mostra o nome de Widgets importantes que podemos encaixar dentro de um AppBar.

The `AppBar` insets its content based on the ambient `MediaQuery`'s padding, to avoid system UI intrusions. It's taken care of by `Scaffold` when used in the `Scaffold.appBar` property. When animating an `AppBar`, unexpected `MediaQuery` changes (as is common in `Hero` animations) may cause the content to suddenly jump. Wrap the `AppBar` in a `MediaQuery` widget, and adjust its padding such that the animation is smooth.



Veja também a lista de parâmetros de seu construtor.



Constructors

```
AppBar({Key? key, Widget? leading, bool automaticallyImplyLeading = true, Widget? title,
List<Widget>? actions, Widget? flexibleSpace, PreferredSizeWidget? bottom, double?
elevation, double? scrolledUnderElevation, ScrollNotificationPredicate
notificationPredicate = defaultScrollNotificationPredicate, Color? shadowColor, Color?
surfaceTintColor, ShapeBorder? shape, Color? backgroundColor, Color?
foregroundColor, IconThemeData? iconTheme, IconThemeData? actionsIconTheme,
bool primary = true, bool? centerTitle, bool excludeHeaderSemantics = false, double?
titleSpacing, double toolbarOpacity = 1.0, double bottomOpacity = 1.0, double?
toolbarHeight, double? leadingWidth, TextStyle? toolbarTextStyle, TextStyle?
titleTextStyle, SystemUiOverlayStyle? systemOverlayStyle, bool
forceMaterialTransparency = false, Clip? clipBehavior})}
```

Creates a Material Design app bar.

Exercício.

- Lendo a documentação de AppBar, descubra o nome do Widget que podemos utilizar para adicionar um título ao Widget AppBar. Adicione o título “Minhas Imagens”.
- Faça hot reload e verifique o resultado no navegador.
- Faça push ao repositório Git.

Veja como fica.

```
import 'package:flutter/material.dart';

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
    ),
  );
  runApp(app);
}
```

A seguir, vamos lidar com um botão que, quando clicado, deve realizar uma tarefa. Essa tarefa é realizada por uma função. Por isso, vamos estudar um pouco sobre elas em Dart. Faça os seguintes exemplos no DartPad.

```
void main() {
    final f1 = () {
        print('f1');
    };
    final f2 = () {
        print('f2');
        return "f2";
    };
    final f3 = () => print("f3");
    //parece uma função que pode ter várias linhas
    //mas é uma função que devolve um set ou map
    //delimitado por {}
    final f4 = () => {
        //esse print não vale pois {} não é o corpo de
        //uma função
        // print ("E agora?");
    };
    //devolve um Set com 1 e 2
    final f5 = () => {
        1, 2
    }
    //devolve um mapa com nome: 'Ana'
    final f6 = () => {
        'nome': 'Ana'
    };

    //sem inferência de tipo: funções são do tipo Function
    Function f7 = () {
        print('f7');
    };
    //funções podem receber funções
    final f8 = (f) {
        print('f8');
        //chamando a função recebida
        f();
    };
}
```

```
//com parâmetro
final f9 = (int? a){
    print(a);
}
//f1 devolve null
print(f1());
//f2 devolve seu nome
print(f2());
//f3 devolve o que o print que ela chama
//devolve void neste caso
//por isso, não podemos imprimir o retorno de f3
//print(f3());
//devolve um map vazio
print(f4());
//devolve um set com 1 e 2
print(f5());
//devolve um mapa com nome: Ana
print(f6());
//normal, mas sem inferência de tipo
print(f7());
//f8 recebe f1 e a coloca em execução
print(f8(f1));
//f8 recebe uma arrow function sem nome
print(f8(() => print('oi')));
//recebe 2 como parâmetro
print(f9(2));
}
```

Exercício.

- Lendo a documentação de **Scaffold**, descubra o nome do Widget que podemos utilizar para adicionar um botão flutuante ao Scaffold.
- Adicione um botão flutuante do tipo **FloatingActionButton**.
- Você terá obtido um erro, indicando que FloatingActionButton requer um parâmetro. Descubra qual é e corrija, lendo a documentação de FloatingActionButton.

<https://api.flutter.dev/flutter/material/FloatingActionButton-class.html>

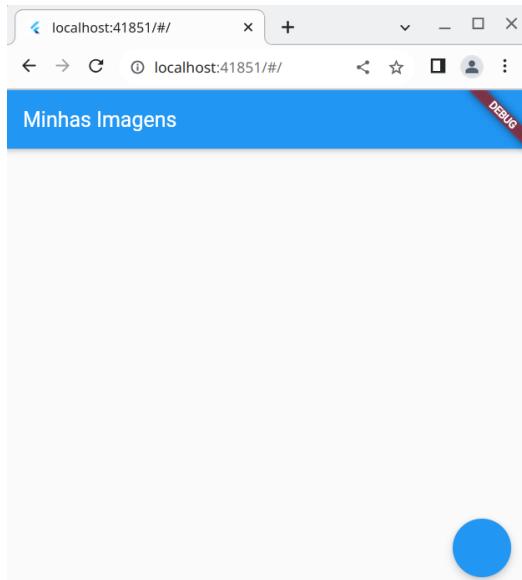
- Faça um hot reload e veja a saída no terminal e no console do navegador (Chrome Dev Tools: CTRL + SHIFT + I).
- Faça push ao repositório Git.

Veja como fica.

```
import 'package:flutter/material.dart';

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          print("Hello!");
        },
      ),
    ),
  );
  runApp(app);
}
```

No navegador, o resultado esperado é o seguinte.



O botão ainda não exibe conteúdo algum. Precisamos descobrir como fazê-lo mostrar algo. Talvez um símbolo “+” neste caso. Para tal, vamos inspecionar a sua documentação uma vez mais.

<https://api.flutter.dev/flutter/material/FloatingActionButton-class.html>

Constructors

```
FloatingActionButton({Key? key, Widget? child, String? tooltip, Color? foregroundColor, Color? backgroundColor, Color? focusColor,
  Color? hoverColor, Color? splashColor, Object? heroTag = const _DefaultHeroTag(), double? elevation, double? focusElevation,
  double? hoverElevation, double? highlightElevation, double? disabledElevation, required VoidCallback? onPressed, MouseCursor?
  mouseCursor, bool mini = false, ShapeBorder? shape, Clip clipBehavior = Clip.none, FocusNode? focusNode, bool autofocus =
  false, MaterialTapTargetSize? materialTapTargetSize, bool isExtended = false, bool? enableFeedback})  

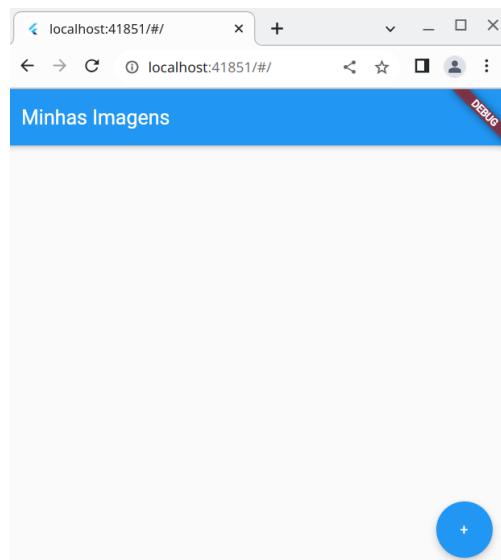
Creates a circular floating action button.  

const
```

Para exibir o conteúdo textual, vamos tentar fazer uso do parâmetro nomeado **child**.

```
import 'package:flutter/material.dart';  
  
void main() {  
  var app = MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: const Text("Minhas Imagens"),  
      ),  
      floatingActionButton: FloatingActionButton(  
        child: const Text("+"),  
        onPressed: () {  
          print("Hello!");  
        },  
      ),  
    ),  
  );  
  runApp(app);  
}
```

Embora funcione, o resultado visual mostra um símbolo “+” muito pequeno.



Ocorre que não estamos exibindo o ícone “+” corretamente. Observe o destaque. Ele é um ícone, não apenas um texto. Precisamos encontrar o mecanismo apropriado para fazer a sua exibição. Ela pode ser feita usando a classe **Icon**.

Exercício.

- Estude a documentação da classe **Icon** e descubra como exibir o ícone “+”.

<https://api.flutter.dev/flutter/widgets/Icon-class.html>

Constructors

```
Icon(IconData? icon, {Key? key, double? size, double? fill, double? weight, double? grade, double? opticalSize, Color? color, List<Shadow>? shadows, String? semanticLabel, TextDirection? textDirection})
Creates an icon.
const
```

Dica. Neste caso, vamos fazer uso de um parâmetro não nomeado.

Dica. Tente passar uma constante da classe **Icons** ao construtor da classe **Icon**. Para isso, estude também a documentação da classe **Icons**.

<https://api.flutter.dev/flutter/material/Icons-class.html>

- Faça hot reload e veja o resultado no navegador.
- Faça push ao repositório Git.

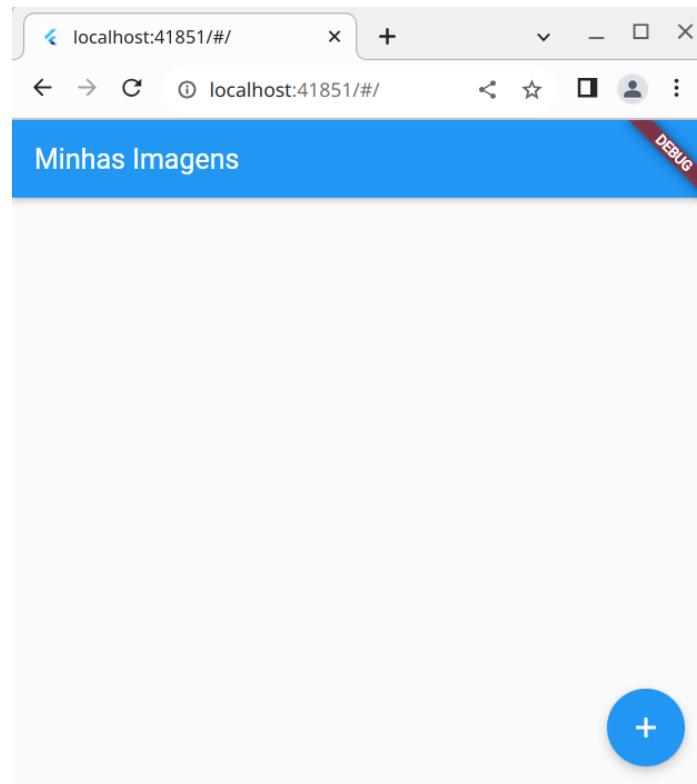
Veja como fica.

```
import 'package:flutter/material.dart';

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () {
          print("Hello!");
        },
      ),
    ),
  );
}
```

```
        runApp(app);  
    }  
}
```

O resultado no navegador é o seguinte.



(Criando um Widget Stateless) Observe que a aplicação inteira está definida em um único arquivo. Claro, esse é um aspecto muito ruim para qualquer ambiente de desenvolvimento de software. Por isso, vamos criar um Widget próprio nosso, com a finalidade de isolar toda a definição que fizemos até então. Este é um bom momento para estudarmos os diferentes tipos de Widgets em Flutter.

Widget: Um Widget é uma descrição imutável de parte de uma interface gráfica. Documentação: <https://api.flutter.dev/flutter/widgets/Widget-class.html>

Widget class abstract



Describes the configuration for an [Element](#).

Widgets are the central class hierarchy in the Flutter framework. A widget is an immutable description of part of a user interface. Widgets can be inflated into elements, which manage the underlying render tree.

Widgets themselves have no mutable state (all their fields must be final). If you wish to associate mutable state with a widget, consider using a [StatefulWidget](#), which creates a [State](#) object (via [StatefulWidget.createState](#)) whenever it is inflated into an element and incorporated into the tree.

A given widget can be included in the tree zero or more times. In particular a given widget can be placed in the tree multiple times. Each time a widget is placed in the tree, it is inflated into an [Element](#), which means a widget that is incorporated into the tree multiple times will be inflated multiple times.

The [key](#) property controls how one widget replaces another widget in the tree. If the [runtimeType](#) and [key](#) properties of the two widgets are [operator==](#), respectively, then the new widget replaces the old widget by updating the underlying element (i.e., by calling [Element.update](#) with the new widget). Otherwise, the old element is removed from the tree, the new widget is inflated into an element, and the new element is inserted into the tree.

StatelessWidget: Descreve parte de uma interface gráfica utilizando outros Widgets. "Sem estado" significa que aquilo que ele exibe depende apenas de sua configuração própria e do objeto [BuildContext](#) que recebe no método [build](#). Documentação:

<https://api.flutter.dev/flutter/widgets/ StatelessWidget-class.html>

A stateless widget is a widget that describes part of the user interface by building a constellation of other widgets that describe the user interface more concretely. The building process continues recursively until the description of the user interface is fully concrete (e.g., consists entirely of [RenderObjectWidgets](#), which describe concrete [RenderObjects](#)).



Stateless widget are useful when the part of the user interface you are describing does not depend on anything other than the configuration information in the object itself and the [BuildContext](#) in which the widget is inflated. For compositions that can change dynamically, e.g. due to having an internal clock-driven state, or depending on some system state, consider using [StatefulWidget](#).

StatefulWidget: Descreve parte de uma interface gráfica utilizando outros Widgets. "Com estado" significa que aquilo que ele exibe depende de informações que podem ser obtidas externamente quando ele é construído e que também podem ser alteradas enquanto ele está sendo exibido. Documentação:

<https://api.flutter.dev/flutter/widgets/ StatefulWidget-class.html>

StatelessWidget class abstract



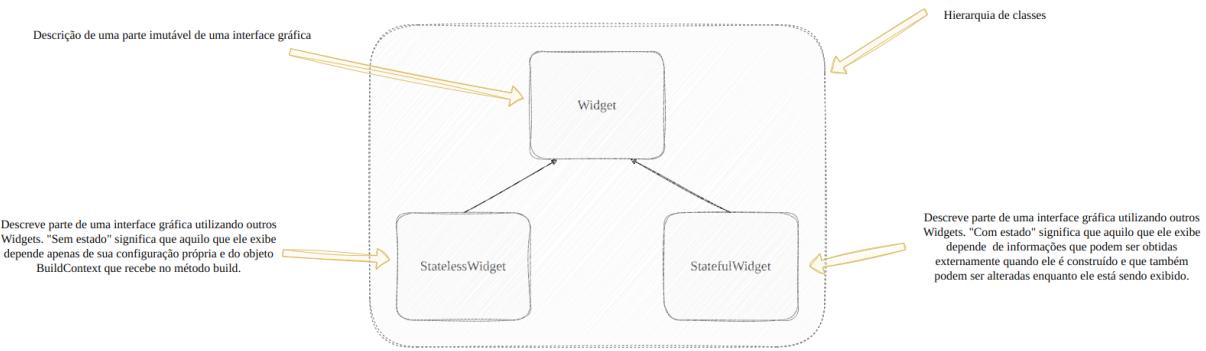
A widget that has mutable state.

State is information that (1) can be read synchronously when the widget is built and (2) might change during the lifetime of the widget. It is the responsibility of the widget implementer to ensure that the [State](#) is promptly notified when such state changes, using [State.setState](#).

A stateful widget is a widget that describes part of the user interface by building a constellation of other widgets that describe the user interface more concretely. The building process continues recursively until the description of the user interface is fully concrete (e.g., consists entirely of [RenderObjectWidgets](#), which describe concrete [RenderObjects](#)).

Stateful widgets are useful when the part of the user interface you are describing can change dynamically, e.g. due to having an internal clock-driven state, or depending on some system state. For compositions that depend only on the configuration information in the object itself and the [BuildContext](#) in which the widget is inflated, consider using [StatelessWidget](#).

Veja a hierarquia de classes.



Exercício.

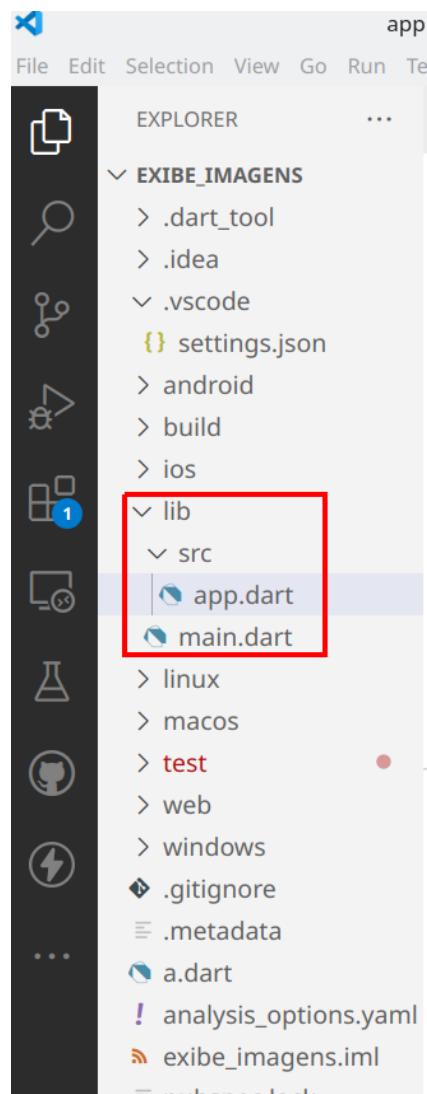
- Crie uma subpasta de **lib** chamada **src**.
- Crie um arquivo chamado **app.dart** na pasta **src**.
- No arquivo **app.dart**, importe a biblioteca material do Flutter.
- No arquivo **app.dart**, crie uma classe chamada **App** e faça com que ela herde de **StatelessWidget**.

Dica. O operador de herança em Dart é o **extends**.

- Escreva um método chamado **build** cujo tipo de retorno é **Widget**. Ele deve receber um parâmetro do tipo **BuildContext**.
- Recorte o conteúdo do arquivo **main.dart**, a partir da definição de **MaterialApp** e cole no método **build**, fazendo com que ele devolva o Widget **MaterialApp**. No arquivo **main.dart**, mantenha a declaração da variável **app** sem nada atribuído a ela neste momento.
- Faça o método **print** exibir outro texto, algo como “Estou n o arquivo **app.dart**”.
- Salve o arquivo **app.dart** para que o código seja indentado automaticamente.

- No arquivo **main.dart**, importe o arquivo **app.dart**.
- Construa uma instância de App no arquivo **main.dart** e faça com que a variável **app** a referencie.
- Faça hot reload e confira o resultado, clicando no FloatingActionButton.
- Faça push ao repositório Git.

Veja como fica.



No arquivo **app.dart**, a definição inicial fica assim.

```
import 'package:flutter/material.dart';

class App extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    }
}
```

Depois disso, o método **build** passa devolver o Widget **MaterialApp** outrora definido no arquivo **main.dart**. Recorte a sua definição do arquivo **main.dart** e cole no arquivo **app.dart**.

```
import 'package:flutter/material.dart';

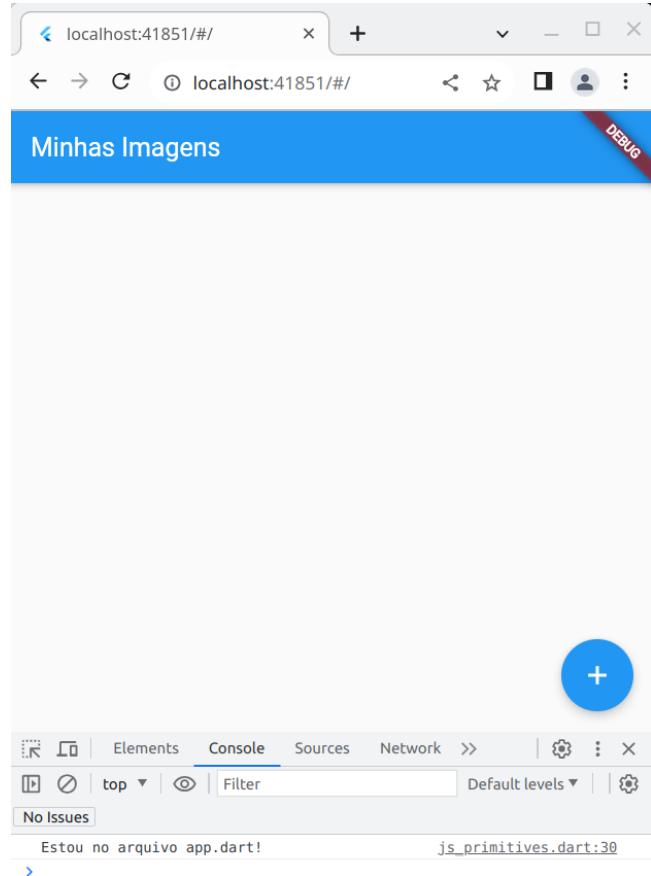
class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text("Minhas Imagens"),
        ),
        floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.add),
          onPressed: () {
            print("Estou no arquivo app.dart!");
          },
        ),
        ),
      );
  }
}
```

No arquivo **main.dart**, importamos e passamos a utilizar o novo Widget.

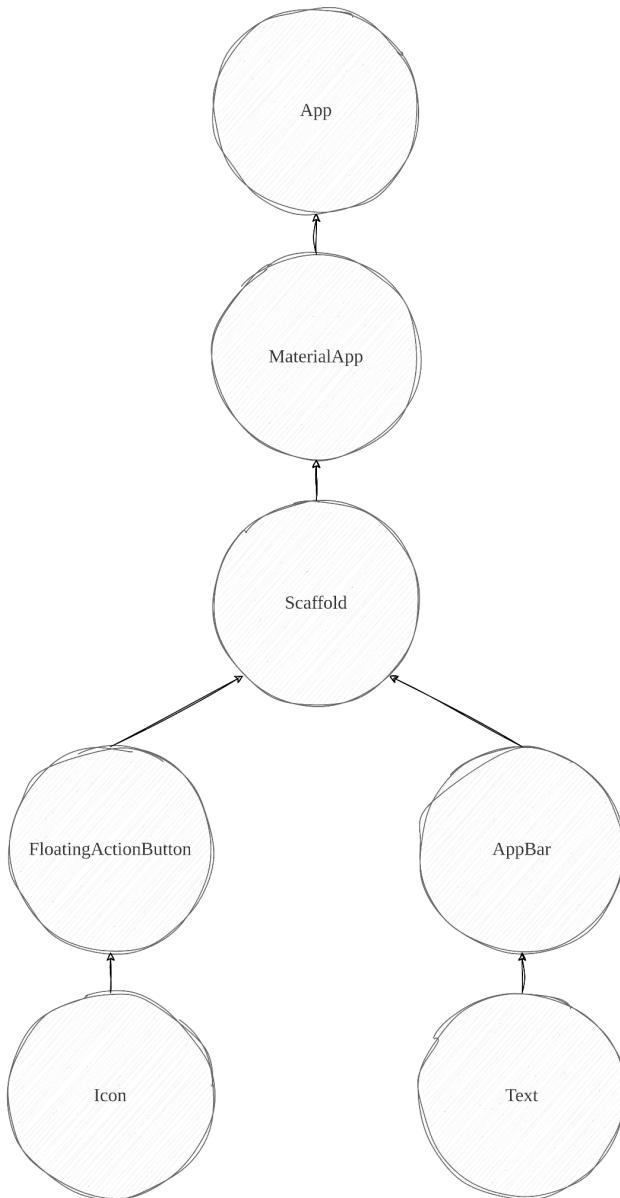
```
import 'package:flutter/material.dart';
import 'src/app.dart';

void main() {
  var app = App();
  runApp(app);
}
```

Aperte SHIFT + R no terminal em que executou flutter run. Clique no FloatingActionButton e veja se funcionou. Veja o resultado esperado.



(A árvore de Widgets) A interface gráfica de uma aplicação Flutter é representada por uma árvore de Widgets. Veja a árvore que temos até este momento.



(Estado. Exibindo o número de imagens: Utilizando um Widget Stateful) Vamos ajustar a aplicação para que ela exiba o número de imagens exibidas, além das próprias imagens. Essa é uma informação que, evidentemente, será atualizada sempre que o usuário clicar no botão para adicionar nova imagem. Ela faz parte do **estado** do Widget. Por isso, vamos precisar de um Widget com estado. Assim, vamos refatorar a aplicação, deixando de utilizar um **StatelessWidget** e passando a utilizar um **StatefulWidget**.

A refatoração envolve os seguintes passos.

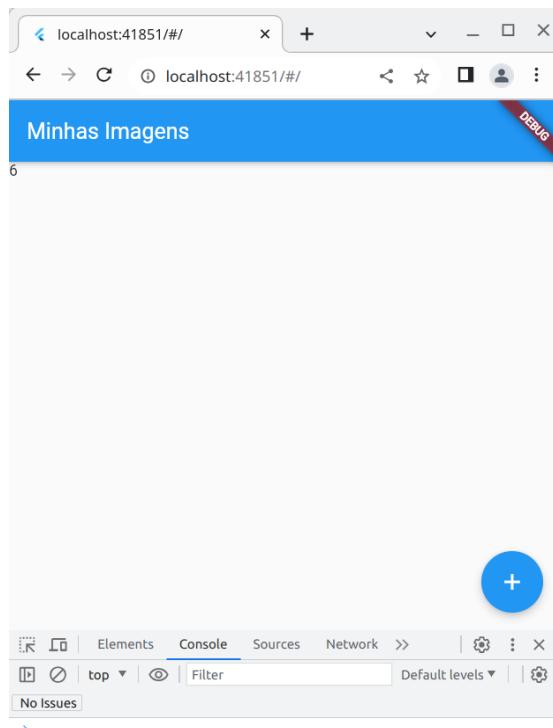
- Escrever duas classes: uma representa o estado da aplicação e a outra representa o Widget.
- A classe que representa o Widget deve possuir um método chamado **createState**. Ele devolve uma instância da classe que representa o estado do Widget.
- A classe que representa o estado da aplicação deve possuir um método chamado **build** que devolve a descrição do Widget de interesse.
- A classe que representa o estado da aplicação deve possuir variáveis de instância. Cada uma representa uma “fatia” do estado de interesse.
- Quando os dados forem alterados, ou seja, quando o usuário clicar no botão, vamos chamar o método **setState**.

Exercício.

- No arquivo **app.dart**, renomeie a classe **App** para **AppState**.
- No arquivo **app.dart** crie uma classe chamada **App** que herda de **StatefulWidget**.
- Faça com que a classe **AppState** herde de **State <App>**. Repare no generics.
- Defina um método chamado **createState** na classe **App**. Seu tipo de retorno deve ser **State <App>**. Ele não recebe coisa alguma como parâmetro. Ele deve devolver uma instância de **AppState**.
- Adicione um inteiro como variável de instância à classe **AppState**. Ela deve se chamar **numerolimagens** e deve ser inicializada com zero.
- Quando o usuário clicar no botão, incremente o contador de imagens.
- Chame o método **setState** depois de incrementar. Passe como parâmetro uma arrow function. Ela deve incrementar o contador. Assim, deixe de incrementá-lo como havia feito anteriormente e apenas incremente na função entregue a **setState**.
- Estude a documentação de Scaffold e descubra como exibir um “corpo”.

<https://api.flutter.dev/flutter/material/Scaffold-class.html>
- Use a propriedade encontrada como parâmetro nomeado de Scaffold. Associe a ela um Widget Text. Seu conteúdo deve ser o contador de imagens. Utilize interpolação para exibir seu valor.
- Faça push ao repositório Git.

Veja como fica. Clique algumas vezes no botão para testar.



Primeiro renomeamos a classe **App**.

```
import 'package:flutter/material.dart';

class AppState extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    ...
  }
}
```

Depois disso, criamos uma classe chamada **App** que herda de **StatefulWidget**. Ela possui o método **createState**. Ele devolve uma instância de AppState. Para tal, precisamos fazer com que **AppState** passe a herdar de **State<App>**.

```

import 'package:flutter/material.dart';

class App extends StatefulWidget {
    @override
    State<App> createState() {
        return AppState();
    }
}

class AppState extends State<App> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            ...
        );
    }
}

```

A seguir, adicionamos o contador de imagens à classe AppState.

```

...
class AppState extends State<App> {
    int numeroImagens = 0;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            ...
        );
    }
}

```

Atualizamos o contador. Num primeiro momento, direto na função que é chamada quando o usuário clica no botão.

```

...
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: () {
        numeroImagens++;
        print("Estou no arquivo app.dart!");
    },
),
...

```

Chamamos o método setState quando o usuário clicar no botão. Ele recebe uma função e incrementa o contador de imagens. O incremento passa a ser feito apenas ali. Também podemos remover a instrução print, que estávamos utilizando apenas para um teste inicial.

```
...
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: () {
    setState(() => numeroImagens++);
  },
),
...

```

Adicionamos o parâmetro home Scaffold e a ele associamos uma instância de Text. Como parâmetro, ela recebe o número de imagens. Repare na interpolação.

```
...
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () {
          setState(() => numeroImagens++);
        },
      ),
      body: Text('$numeroImagens'), //parâmetro entregue ao construtor de Scaffold
    ),
  );
}
...

```

(Obtenção de imagens: Pexels) Vamos obter imagens a partir do site Pexels. Para isso, comece acessando a sua página inicial e criando uma conta para você.

<https://www.pexels.com/>

O acesso ao conteúdo de Pexels é gratuito. Porém, há alguns limites para acessos realizados por meio de requisições direcionadas à sua API.

Segundo a sua documentação, que pode ser acessada a partir do link a seguir, o limite atual é

- 200 requisições por hora
- 20000 requisições por mês

É possível entrar em contato com o suporte e solicitar um aumento gratuito. Basta apresentar razões e exemplos concretos. Se for de interesse da Pexels, você pode até mesmo obter acesso ilimitado.

<https://www.pexels.com/api/documentation/>

Guidelines

Whenever you are doing an API request make sure to show a **prominent link to Pexels**. You can use a text link (e.g. "Photos provided by Pexels") or a link with our logo.

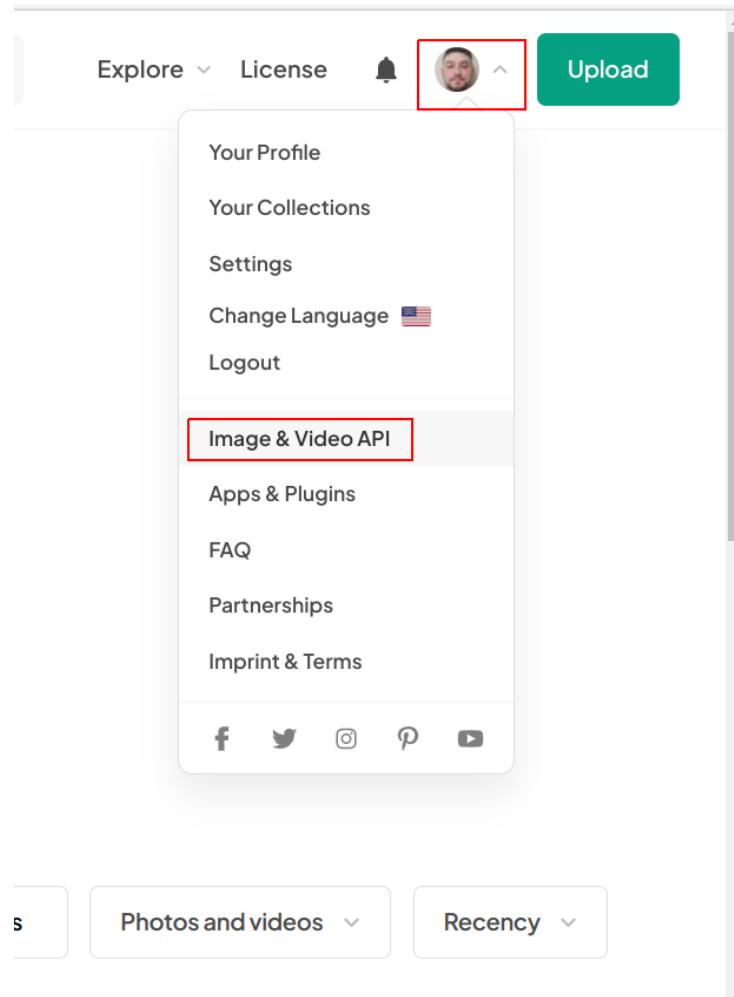
Always credit our photographers when possible (e.g. "Photo by John Doe on Pexels" with a link to the photo page on Pexels).

You may not copy or replicate core functionality of Pexels (including making Pexels content available as a wallpaper app).

Do not abuse the API. By default, the API is rate-limited to 200 requests per hour and 20,000 requests per month. **You may contact us to request a higher limit**, but please include examples, or be prepared to give a demo, that clearly shows your use of the API with attribution. If you meet our API terms, you can get unlimited requests for free.

Abuse of the Pexels API, including but not limited to attempting to work around the rate limit, will lead to termination of your API access.

Depois de criar a conta e fazer login, clique no seu avatar no canto superior direito e escolha **Image & Video API**.



Clique em **Your API Key** para visualizar a sua chave de API.

The screenshot shows the Pexels homepage. At the top, there is a search bar with the placeholder 'Search for free photos' and a magnifying glass icon. Below the search bar are navigation links for 'Explore', 'License', and 'Upload'. A small user icon is also present. In the center, there is a large call-to-action button labeled 'Start building with the power of Pexels'. Below this button, there is a paragraph of text about the API and two buttons: 'Your API Key' (which is highlighted with a red box) and 'Documentation'. At the bottom, there is a note for high-traffic platforms and a link to contact support. To the right, a browser window displays a sample API response for a search query. The response includes a JSON object with fields like 'page', 'per_page', 'total_results', 'url', 'next_page', and 'photos'. The 'photos' field contains an array of image objects, each with properties such as 'width', 'height', 'url', 'photographer', and 'src'. Below the JSON, several sample images are shown, including a close-up of orange mushrooms, a waterfall, a mountain peak, and a shark underwater.

(Trabalhando com objetos JSON) JSON significa Javascript Object Notation. É uma notação bastante utilizada para a representação de dados que precisam ser enviados e recebidos por aplicações distintas, eventualmente escritas em diferentes linguagens de programação. Veja a sua especificação oficial.

<https://www.json.org/json-en.html>

Visite a página e estude os grafos sintáticos.

Veja alguns exemplos práticos.

Uma pessoa se chama Ana e tem 22 anos.

```
{
  "nome": "Ana",
  "idade": 22
}
```

Uma pessoa se chama João, tem 30 anos e mora na Rua B, número 10, Vila J.

```
{
  "nome": "João",
  "idade": 30,
  "endereco": {
    "logradouro": "Rua B",
    "numero": 10,
    "bairro": "Vila J"
  }
}
```

Uma coleção de pessoas. Cada pessoa tem apenas nome.

```
[
  "Ana",
  "João",
  "Pedro"
]
```

Uma coleção de pessoas. Cada pessoa tem nome e idade.

```
[
  {
    "nome": "Ana",
    "idade": 22
  },
  {
    "nome": "Pedro",
    "idade": 28
  }
]
```

Agora as pessoas são alunos de uma escola. A escola tem um nome.

```
{
  "nome": "Escola A",
  "alunos": [
    {
      "nome": "Ana",
      "idade": 22
    },
    {
      "nome": "Pedro",
      "idade": 28
    }
  ]
}
```

Exercício. Represente o seguinte com JSON. Uma concessionária tem um CNPJ e duas filiais. Cada filial tem nome, endereço (logradouro, numero e bairro) e uma coleção de veículos. Cada veículo tem modelo, placa e marca. Cada filial tem dois veículos.

Veja como fica.

```
{  
    "cnpj": "123456789/0001",  
    "filiais": [  
        // filial 1  
        {  
            "nome": "Filial 1",  
            "endereco": {  
                "logradouro": "Rua A",  
                "numero": 100,  
                "bairro": "Vila A"  
            },  
            "veiculos": [  
                {  
                    "modelo": "Gol",  
                    "marca": "VW",  
                    "placa": "ABC-1234"  
                },  
                {  
                    "modelo": "Celta",  
                    "marca": "Chevrolet",  
                    "placa": "DEF-5544"  
                }  
            ]  
        },  
        // filial 2  
        {  
            "nome": "Filial 2",  
            "endereco": {  
                "logradouro": "Rua B",  
                "numero": 100,  
                "bairro": "Vila B"  
            },  
            "veiculos": [  
                {  
                    "modelo": "Corsa",  
                    "marca": "Chevrolet",  
                    "placa": "ERF-1122"  
                },  
                {  
                    "modelo": "Fusca",  
                    "marca": "VW",  
                    "placa": "DEF-1111"  
                }  
            ]  
        }  
    ]  
}
```

(Manipulando JSON com Dart) Nesta seção, vamos abrir uma instância do DartPad para aprender como podemos manipular objetos JSON em Dart. Lembre-se que o DartPad está disponível a seguir.

<https://dartpad.dev/>

Comece escrevendo uma função main. Ela define um JSON que representa uma pessoa.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}
```

Observe que um JSON é uma criatura puramente textual. Se desejarmos extrair alguma informação a respeito da pessoa, como seu nome, teremos de aplicar algoritmos de manipulação de strings, o que pode ser muito repetitivo e trabalhoso. Seria muito mais prático se pudéssemos utilizar uma instrução como “me dê o valor associado à chave “nome””. De fato, podemos. Utilizando mapas. Para isso, basta convertermos o objeto JSON para uma estrutura do tipo mapa. Essa é uma operação muito comum em muitas linguagens de programação. Em Dart, podemos fazê-lo utilizando o pacote **dart:convert**. Ele oferece um objeto chamado json e, por meio dele, acessamos o método **decode**. Ele recebe um JSON e devolve um mapa. Veja.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';

  var pessoa = json.decode(pessoaJson);

  print(pessoa);
  // _JsonMap
  print(pessoa.runtimeType);

  print(pessoa['nome']);
  print(pessoa['idade']);

}
```

O mesmo vale para coleções. Observe como podemos iterar sobre elas com um `for each` e também com um `for "regular"`.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';

  var pessoa = json.decode(pessoaJson);

  print(pessoa);
  // _JsonMap
  print(pessoa.runtimeType);

  print(pessoa['nome']);
  print(pessoa['idade']);

  var veiculosJson = '''[
    {
      "marca": "VW",
      "modelo": "Gol"
    },
    {
      "marca": "Chevrolet",
      "modelo": "Fusca"
    }
]'''';

  var veiculos = json.decode(veiculosJson);
  print(veiculos);
  // _JSArray<dynamic>
  print(veiculos.runtimeType);
  for (final veiculo in veiculos) {
    print(veiculo);
    print(veiculo['marca']);
    print(veiculo['modelo']);
  }

  for (var i = 0; i < veiculos.length; i++) {
    print(veiculos[i]);
  }
}
```

Exercício. Neste exercício, vamos utilizar o objeto “concessionária” previamente ilustrado. Por isso, copie a sua definição para o DartPad, fazendo os ajustes necessários.

- Adicione um preço a cada veículo de cada filial.
- Calcule e mostre o preço médio dos veículos, por filial.
- Calcule e mostre o preço médio dos veículos da concessionária como um todo. Mostre também o total de veículos da concessionária.

Veja como fica.

```
import 'dart:convert';
void main() {
  var concessionariaJSON = '''
  "cnpj": "123456789/0001",
  "filiais": [
    {
      "nome": "Filial 1",
      "endereco": {
        "logradouro": "Rua A",
        "numero": 100,
        "bairro": "Vila A"
      },
      "veiculos": [
        {
          "modelo": "Gol",
          "marca": "VW",
          "placa": "ABC-1234",
          "preco": 20000
        },
        {
          "modelo": "Celta",
          "marca": "Chevrolet",
          "placa": "DEF-5544",
          "preco": 17000
        }
      ]
  },
  {
```

```

    "nome": "Filial 2",
    "endereco": {
        "logradouro": "Rua B",
        "numero": 100,
        "bairro": "Vila B"
    },
    "veiculos": [
        {
            "modelo": "Corsa",
            "marca": "Chevrolet",
            "placa": "ERF-1122",
            "preco": 12000
        },
        {
            "modelo": "Fusca",
            "marca": "VW",
            "placa": "DEF-1111",
            "preco": 15000
        }
    ]
}
}'''';

var concessionaria = json.decode(concessionariaJSON);
print(concessionaria.runtimeType);
var filiais = concessionaria['filiais'];
print(filiais.runtimeType);
double mediaTotal = 0;
int totalVeiculos = 0;
for (final filial in filiais){
    var veiculos = filial['veiculos'];
    double media = 0;
    for (final veiculo in veiculos){
        media += veiculo['preco'];
        mediaTotal += veiculo['preco'];
        totalVeiculos++;
    }
    print('Filial ${filial['nome']}: ${media/veiculos.length}');
}
print('Média total: ${mediaTotal / totalVeiculos}');
print('Total de veículos: ${totalVeiculos}');

```

```
}
```

(Classes de modelo) Os valores contidos na estrutura que a função **decode** nos entrega têm tipo conhecido somente em tempo de execução (dynamic). Podemos deixar o código mais seguro e previsível convertendo essa estrutura para uma classe de modelo. Esta classe contém os campos desejados e eles são definidos utilizando-se o sistema de tipos estático da linguagem. No DartPad, volte para o seguinte modelo simples.

```
import 'dart:convert';
void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}
```

Depois disso, escrevemos uma classe de Modelo. Ela tem as propriedades de interesse e um construtor.

```
import 'dart:convert';
void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}

class PessoaModel{
  String nome;
  int idade;
  //construtor
  PessoaModel(this.nome, this.idade);
}
```

Por fim, fazemos a conversão com o método **decode** e construímos um objeto de modelo. O que ganhamos essencialmente foi o uso do sistema de tipos estático e, portanto, o suporte do compilador.

```

import 'dart:convert';
void main() {
    var pessoaJson = '{"nome": "Ana", "idade": 18}';
    var pessoaDecoded = json.decode(pessoaJson);
    var pessoa = PessoaModel(
        pessoaDecoded['nome'],
        pessoaDecoded['idade']
    );
    //agora podemos acessar assim:
    print(pessoa.nome);
    print(pessoa.idade);
}
class PessoaModel{
    String nome;
    int idade;
    //construtor
    PessoaModel(this.nome, this.idade);
}

```

(Construtores nomeados) Nesta seção, vamos definir um construtor **nomeado**. Ele terá as seguintes propriedades:

- receberá o objeto JSON produzido pela função decode.
- devolverá um objeto construído a partir da classe de Modelo, já com os campos preenchidos. O objetivo é isolar e escrever uma única vez o seguinte trecho:

```

//não altere nada, apenas observe
var pessoa = PessoaModel(
    pessoaDecoded['nome'],
    pessoaDecoded['idade']
);

```

O trecho citado incomoda pois pode se tornar muito grande e repetitivo caso o objeto envolvido possua muitas propriedades.

Começamos definindo o construtor nomeado.

```
...
class PessoaModel {
    String nome;
    int idade;

    //construtor
    PessoaModel(this.nome, this.idade);

    //construtor nomeado
    PessoaModel.fromJson(decodedJSON) {
        nome = decodedJSON['nome'];
        idade = decodedJSON['idade'];
    }
}
```

Observe que o código causa um erro.

```
21
22     //construtor nomeado
23 ▼   PessoaModel.fromJson(decodedJSON) {
24     nome = decodedJSON['nome'];
25     idade = d Non-nullble instance field 'nome'
26   };
27 }
```

Ocorre que, quando o construtor nomeado entra em execução, o ambiente Dart já espera que os campos tenham sido inicializados. Podemos ajustar marcando-os como opcionais ou “prometendo” que eles terão sido inicializados posteriormente, antes de serem acessados. Escolhamos a segunda opção. Para isso, marcamos as variáveis com **late**.

```

...
class PessoaModel {
    late String nome;
    late int idade;

    //construtor
    PessoaModel(this.nome, this.idade);

    //construtor nomeado
    PessoaModel.fromJSON(decodedJSON) {
        nome = decodedJSON['nome'];
        idade = decodedJSON['idade'];
    }
}

```

Passamos a construir o objeto assim.

```

import 'dart:convert';
void main() {
    var pessoaJson = '{"nome": "Ana", "idade": 18}';
    var pessoaDecoded = json.decode(pessoaJson);
    var pessoa = PessoaModel.fromJSON(pessoaDecoded);
    //agora podemos acessar assim:
    print(pessoa.nome);
    print(pessoa.idade);
}

class PessoaModel{
    late String nome;
    late int idade;
    //construtor
    PessoaModel(this.nome, this.idade);
    //construtor nomeado
    PessoaModel.fromJSON(decodedJSON) {
        nome = decodedJSON['nome'];
        idade = decodedJSON['idade'];
    }
}

```

(Modelo para a imagem) De volta ao projeto, vamos criar uma classe de modelo para as imagens que a Pexels nos entregará. Veja um exemplo de JSON que ela nos entrega.

```

2   "page": 1,
3   "per_page": 1,
4   "photos": [
5     {
6       "id": 5637733,
7       "width": 3840,
8       "height": 5760,
9       "url": "https://www.pexels.com/photo/man-in-yellow-jacket-and-black-pants-walking-on-sidewalk-5637733/",
10      "photographer": "RDNE Stock project",
11      "photographer_url": "https://www.pexels.com/@rdne",
12      "photographer_id": 3149039,
13      "avg_color": "#AD9A4",
14      "src": {
15        "original": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg",
16        "large2x": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&dpr=2&h=650&w=940",
17        "large": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=650&w=940",
18        "medium": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=350",
19        "small": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=130",
20        "portrait": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&fit=crop&h=1200&w=800",
21        "landscape": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&fit=crop&h=627&w=1200",
22        "tiny": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&dpr=1&fit=crop&h=200&w=280"
23      },
24      "liked": false,
25      "alt": "Man in Yellow Jacket and Black Pants Walking on Sidewalk"
26    }
27  ],
28  "total_results": 8000,
29  "next_page": "https://api.pexels.com/v1/search/?page=2&per_page=1&query=people"
-- .
```

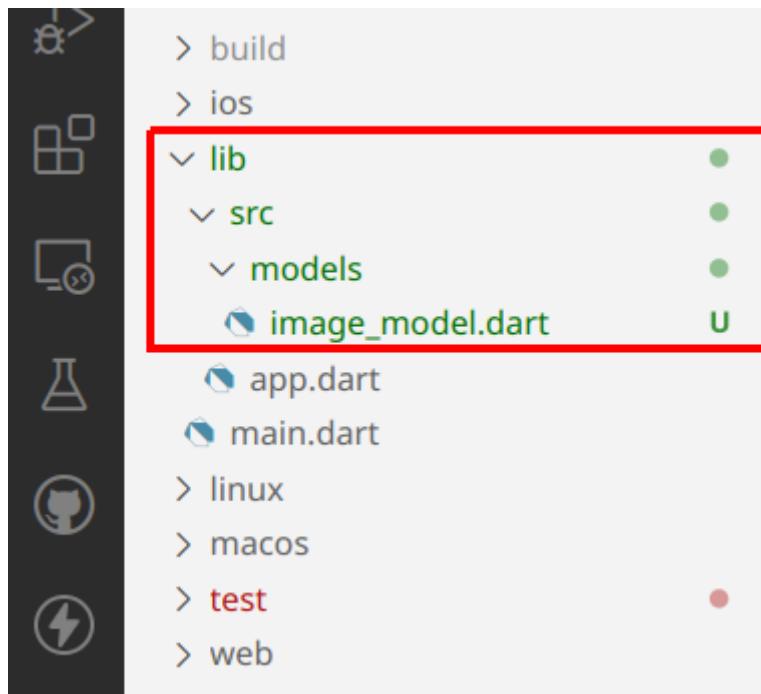


Observe que o resultado possui uma coleção chamada **photos**. Cada posição contém uma foto. Cada foto tem algumas propriedades, como:

- **url**: endereço dela que nos leva para uma página do site Pexels. Não é esse que desejamos
- **src**: um JSON com propriedades como original, large2x, medium, small etc. É daí que escolheremos a URL a utilizar na aplicação. A foto é sempre a mesma, o que muda é o tamanho.
- **alt**: é uma descrição textual da foto.

Entre outras propriedades. Inspecione o resultado para conhecer mais.

Para criar o modelo de imagem, crie uma pasta chamada **models** na pasta **src** de seu projeto. A seguir, crie um arquivo chamado **image_model.dart**. Observe.



Digamos que o modelo, a princípio, tem as seguintes propriedades: url e alt.

Exercício. Escreva a classe de modelo com url e alt, considerando a estrutura do objeto devolvido pela Pexels.

Veja como fica.

```
class ImageModel {
  late String url;
  late String alt;

  ImageModel(this.url, this.alt);

  ImageModel.fromJSON(Map <String, dynamic> decodedJSON) {
    //o resultado tem uma coleção de fotos
    //vamos pegar sempre a primeira
    //por isso, posição zero
    //dela, pegamos o tamanho médio (propriedade do objeto src)
    url = decodedJSON['photos'][0]['src']['medium'];
    alt = decodedJSON['photos'][0]['alt'];
  }
}
```

Nota. Há uma outra forma de definição do corpo de um construtor. Veja.

```
class ImageModel {
    late String url;
    late String alt;

    ImageModel(this.url, this.alt);

    // ImageModel.fromJSON(Map <String, dynamic> decodedJSON) {
    //     // o resultado tem uma coleção de fotos
    //     // vamos pegar sempre a primeira
    //     // por isso, posição zero
    //     // dela, pegamos o tamanho médio (propriedade do objeto src)
    //     url = decodedJSON['photos'][0]['src']['medium'];
    //     alt = decodedJSON['photos'][0]['alt'];
    // }

    ImageModel.fromJSON(Map <String, dynamic> decodedJSON)
        : url = decodedJSON['photos'][0]['src']['medium'],
        alt = decodedJSON['photos'][0]['alt'];
}
```

Neste material, entretanto, vamos manter a primeira forma.

(Requisição HTTP para obter uma imagem) Quando o usuário clicar no botão, desejamos disparar uma requisição HTTP direcionada aos servidores da Pexels a fim de obter uma imagem. Para isso, vamos escrever uma função que tem essa responsabilidade. Estamos no arquivo **app.dart**.

```
...
class AppState extends State<App> {
    int numeroImagens = 0;

    void obterImagem() {

    }

    @override
    Widget build(BuildContext context) {
        ...
    }
}
```

A seguir, na função associada ao parâmetro `onPressed` do `FloatingActionButton`, vamos deixar de atualizar o estado e chamar a função criada.

```
...
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: () {
        obterImagen();
    },
),
...
```

Observe que, agora, temos a especificação de uma função que não faz nada além de chamar outra. Assim, podemos especificar apenas o nome da função interna. Veja.

```
...
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: obterImagen,
),
...
```

Cuidado. Especificar apenas o nome da função significa deixá-la associada ao parâmetro nomeado no sentido de que ela deve ser chamada quando o evento envolvido acontecer. Isso é completamente diferente de chamá-la explicitamente, como a seguir.

```
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: obterImagen(), //errado, não faça isso
),
```

Em Dart, isso é um erro em tempo de compilação, pois a nossa função devolve “void”. Assim que o método `build` é chamado, ela também é chamada (ao invés de apenas ser chamada quando o evento de clique no botão acontecer) e, assim, estamos tentando associar “void” ao parâmetro `onPressed`. É claro, não podemos fazer isso. Em outros ambientes, como no React, esse é um erro comum que faz com que a função seja chamada assim que o componente aparece na tela. Em geral, isso faz com que a tela seja atualizada e a função seja chamada de novo, causando uma recursão infinita.

Para realizar as requisições HTTP, vamos utilizar o pacote **http**. O primeiro passo é fazer a sua instalação. Num terminal vinculado à raiz da aplicação, execute

```
flutter pub add http
```

Investigue a seção **dependencies** de seu arquivo **pubspec.yaml**. A nova dependência deve ter sido adicionada ali.



```

app.dart M
! pubspec.yaml M X
New Request

! pubspec.yaml
27 # dependencies can be manually updated by char
numbers below to
28 # the latest version available on pub.dev. To
dependencies have newer
29 # versions available, run `flutter pub outdated`
30 dependencies:
31   flutter:
32     | sdk: flutter
33
34
35   # The following adds the Cupertino Icons for
36   # Use with the CupertinoIcons class for iOS
37   cupertino_icons: ^1.0.2
38   | http: ^1.1.0
39
40 dev_dependencies:
41   | flutter_test:
42     | ...

```

Visite também a documentação do pacote para conhecer mais sobre ele.

<https://pub.dev/packages/http>

O próximo passo é importar o pacote no arquivo **app.dart**.

```

import 'package:flutter/material.dart';
import 'package:http/http.dart';
...

```

O **import** que fizemos foi “sem nome”. Assim, as funções do pacote podem ser acessadas diretamente, sem qualificação. Veja como falamos **get** diretamente no código a seguir, sem qualificação.

```
...
void obterImagem() {
    get(Uri.parse(''));
}
...
```

Também podemos dar um nome ao pacote. Assim, o acesso a suas funções precisa ser qualificado. Observe.

```
...
import 'package:http/http.dart' as http;

...
void obterImagem() {
    http.get(Uri.parse(''));
}
...
```

Nota. Há também as construções **show** e **hide** que também podem ser envolvidas numa instruções import. A documentação a seguir explica isso.

<https://dart.dev/language/libraries>

Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Além disso, há um mecanismo conhecido como **lazy loading**. Quando utilizado, a aplicação apenas importa a biblioteca quando ela é utilizada. Observe que ele funciona apenas na **web**.

Lazily loading a library

Deferred loading (also called *lazy loading*) allows a web app to load a library on demand, if and when the library is needed. Here are some cases when you might use deferred loading:

- To reduce a web app's initial startup time.
- To perform A/B testing—trying out alternative implementations of an algorithm, for example.
- To load rarely used functionality, such as optional screens and dialogs.

⚠ Only `dart compile js` supports deferred loading. Flutter and the Dart VM don't support deferred loading. To learn more, see [issue #33118](#) and [issue #27776](#).

To lazily load a library, you must first import it using `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

Visite sempre a documentação a fim de obter informações mais atualizadas.

Neste material, vamos manter o import com nome e acesso qualificado.

```
...
import 'package:http/http.dart' as http;
...
void obterImagem() {
    http.get(Uri.parse(''));
}
....
```

(URL Pexels e seus parâmetros) A URL de acesso à Pexels é a seguinte.

<https://api.pexels.com/v1/search>

Para especificar o assunto da foto que desejamos, utilizamos o parâmetro `query`. Fica assim:

<https://api.pexels.com/v1/search?query=people>

Além disso, os resultados que a Pexels nos entrega são **paginados**.

Isso funciona da seguinte forma:

- Uma consulta por people pode resultar numa coleção muito grande de resultados, contendo milhares ou milhões de fotos.
- Na primeira resposta, a Pexels nos entrega apenas uma quantidade pequena de fotos, digamos 15. Além disso, ela nos diz que estamos na “página 1”.
- Na próxima requisição podemos obter mais 15 resultados, instruindo a Pexels a nos entregar os resultados da página 2.

E assim por diante.

Os parâmetros envolvidos se chamam **page** e **per_page**. Eles indicam, respectivamente, o número da página de interesse e o número de itens por página. A URL final fica assim:

https://api.pexels.com/v1/search?query=people&per_page=1&page=1

Para pedir uma nova foto, podemos acessar a URL

https://api.pexels.com/v1/search?query=people&per_page=1&page=2

Ou seja, incrementamos o número da página. Claro, também podemos aumentar o número de resultados por página. Depende da natureza da aplicação.

Assim, podemos realizar a requisição.

Exercício. No método `obterImagem`:

1. Construa um objeto url a partir de `Uri.https`.
 - primeiro parâmetro: host da pexels
 - segundo parâmetro: recurso
 - terceiro parâmetro: mapa de parâmetros da query
2. Construa um objeto Request (a partir do objeto `http`)
 - Primeiro parâmetro: `get` (é o nome do método de interesse)
 - segundo parâmetro: o objeto url
3. Acesse a propriedade **headers** do objeto Request e dela, chame o método `addAll`.
 - Primeiro parâmetro: um mapa com chave Authorization associada à sua chave Pexels.
4. Envie a requisição com `send`.

Veja como fica.

Nota. O valor de **page** está fixo em 1, neste momento. No futuro, a aplicação será atualizada para que este valor seja atualizado a cada requisição. Assim, teremos uma nova imagem a cada clique. No momento, cada clique nos produz exatamente a mesma imagem.

```
...
void obterImagem() {
    var url = Uri.https(
        'api.pexels.com',
        '/v1/search',
        {'query': 'people', 'page': '1', 'per_page': '1'},
    );
    var req = http.Request('get', url);
    req.headers.addAll(
    {
        'Authorization':
        'chave',
    },
    );
    req.send();
}
...
```

E agora, como verificar o resultado? Veja a documentação da função **get**.

<https://pub.dev/documentation/http/latest/http/get.html>

http package > documentation > http > get function Search AP

http library

CLASSES

- BaseClient
- BaseRequest
- BaseResponse
- ByteStream
- Client
- MultipartFile
- MultipartRequest
- Request
- Response
- StreamedRequest
- StreamedResponse

FUNCTIONS

- delete
- get

get function

```
Future<Response> get(
    Uri url,
    {Map<String, String>? headers}
)
Sends an HTTP GET request with the given headers to the given URL.
```

This automatically initializes a new [Client](#) and closes that client once the request is complete. If you're planning on making multiple requests to the same server, you should use a single [Client](#) for all of those requests.

For more fine-grained control over the request, use [Request](#) instead.

Implementation

```
Future<Response> get(Uri url, {Map<String, String>? headers}) =>
    _withClient((client) => client.get(url, headers: headers));
```

Veja também a documentação da função **send** de [Request](#).

<https://pub.dev/documentation/http/latest/http/BaseRequest/send.html>

http package > documentation > http > BaseRequest > send method Search AP

BaseRequest class

CONSTRUCTORS

- BaseRequest

PROPERTIES

- contentLength
- finalized
- followRedirects
- hashCode
- headers
- maxRedirects
- method
- persistentConnection
- runtimeType
- url

METHODS

send method

```
Future<StreamedResponse> send()
Sends this request.
```

This automatically initializes a new [Client](#) and closes that client once the request is complete. If you're planning on making multiple requests to the same server, you should use a single [Client](#) for all of those requests.

Implementation

```
Future<StreamedResponse> send() async {
    var client = Client();

    try {
        var response = await client.send(this);
        var stream = onDone(response.stream, client.close);
        return StreamedResponse(ByteStream(stream), response.statusCode,
            contentLength: response.contentLength,
            request: response.request
    }
}
```

Ambas devolvem um objeto do tipo **Future**. O que é um objeto do tipo Future?

Nota. Um Future é um objeto que fica associado a uma computação assíncrona, não bloqueante, potencialmente demorada. Ele nos permite obter o resultado calculado por esta computação, no futuro, quando ela terminar. Não bloqueante ou assíncrono quer dizer que o fluxo principal de nossa aplicação prossegue normalmente e a função demorada opera em background. Assim, não congelamos a interface com o usuário.

Nota. Um Future em Dart é equivalente a uma Promise em Javascript. Não é necessário conhecer Promises para entender o que são Futures. E vice-versa. Essa é apenas uma analogia que pode ser útil.

Já temos o objeto **Future** em mãos. Ele nos foi entregue pela função `send`. Agora precisamos descobrir como podemos acessar o resultado calculado pela função demorada, quando ela terminar. Neste caso, a função demorada é o acesso à Pexels e o resultado que desejamos é o objeto JSON. Podemos fazer uso da função **then** para obter o resultado.

Nota. Entenda a função **then** da seguinte forma: “execute a computação demorada associada a este Future e então, execute o código da função entregue por parâmetro à função `then`.

Vejamos um primeiro teste usando a função **then**.

```
...
void obterImagem() {
  var url = Uri.https(
    'api.pexels.com',
    '/v1/search',
    {'query': 'people', 'page': '1', 'per_page': '1'},
  );
  var req = http.Request('get', url);
  req.headers.addAll(
  {
    'Authorization':
    'chave',
  },
  );
  req.send().then((result) {
    print(result);
  });
}
...
```

Veja o resultado esperado.

Performing hot restart...
621ms
Restarted application in 621ms.
Instance of 'StreamedResponse'
□

Ainda não tem muita graça. Temos uma instância de **StreamedResponse** em mãos. Veja a sua definição, dada pela documentação.

<https://pub.dev/documentation/http/latest/http/StreamedResponse-class.html>

http package > documentation > http > StreamedResponse class

Search API Docs

http library

CLASSES

- BaseClient
- BaseRequest
- BaseResponse
- ByteStream
- Client

StreamedResponse class

An HTTP response where the response body is received asynchronously after the headers have been received.

Inheritance

Object > BaseResponse > StreamedResponse

Implementers

IOStreamedResponse

Temos um objeto cujos dados internos podem ser obtidos pouco a pouco, sob demanda. É um “fluxo(stream) de resposta”. Dele podemos pegar a propriedade **stream**.

```
...
req.send().then((result) {
  print(result.stream);
});
...
```

Veja o resultado.

40WMS
Restarted application in 460ms.
Instance of 'ByteStream'
□

Ainda não tem graça. Podemos verificar o código de status. Se ele for 200 (sucesso, segundo a especificação do protocolo HTTP), fazemos algo com o resultado. Neste caso, vamos converter nosso StreamedResponse para um Response. Um Response nos permite

obter os dados de interesse por meio de propriedades previamente definidas. Como essa conversão também pode ser demorada, temos de lidar com um novo objeto Future. Veja.

```
...
req.send().then((result) {
  if (result.statusCode == 200) {
    http.Response.fromStream(result).then((response) {
      print(response.body);
    });
  } else {
    print('Falhou!');
  }
});
...
...
```

Agora o resultado ficou mais interessante.

```
Performing hot restart...
408ms
Restarted application in 408ms.
{"page":1,"per_page":1,"photos":[{"id":5637733,"width":3840,"height":5760,"url":"https://www.pexels.com/photo/man-in-yellow-jacket-and-black-pants-walking-on-sidewalk-5637733/","photographer":"RDNE Stock project","photographer_url":"https://www.pexels.com/@rdne","photographer_id":3149039,"avg_color": "#ADA9A4","src":{"original":"https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg","large2x":"https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=650&w=1200"}]
```

Podemos até decodificar o JSON recebido e converter para um ImageModel, não é mesmo?

```
import 'package:flutter/material.dart';
import 'dart:convert';
import 'models/image_model.dart';
import 'package:http/http.dart' as http;
...
req.send().then((result) {
  if (result.statusCode == 200) {
    http.Response.fromStream(result).then((response) {
      var decodedJSON = json.decode(response.body);
    });
  }
});
```

```

        var imagem = ImageModel.fromJSON(decodedJSON);
        print(imagem);
    });
} else {
    print('Falhou!');
}
);
...

```

Antes de executar, pode ser boa ideia fazer a sobreescrita do método **toString** da classe **ImageModel**. Assim, obtemos uma representação textual mais interessante de seus objetos. Vá até o arquivo **image_model.dart**.

Exercício. Faça uma sobreescrita do método `toString` para a classe `ImageModel`. Ele deve produzir uma String contendo ambas as propriedades do objeto.

Veja uma possível sobreescrita. Claro, você pode variar se desejar.

```

class ImageModel {
    late String url;
    late String alt;

    ImageModel(this.url, this.alt);

    ImageModel.fromJSON(Map<String, dynamic> decodedJSON) {
        //o resultado tem uma coleção de fotos
        //vamos pegar sempre a primeira
        //por isso, posição zero
        //dela, pegamos o tamanho médio (propriedade do objeto src)
        url = decodedJSON['photos'][0]['src']['medium'];
        alt = decodedJSON['photos'][0]['alt'];
    }

    @override
    String toString() {
        return 'url:$url, alt:$alt';
    }
}

```

Execute mais uma vez para ver o resultado.

```

Performing hot restart...
  410ms
Restarted application in 410ms.
url:https://images.pexels.com/photos/5637733/pexels-photo-5637733.
jpeg?auto=compress&cs=tinysrgb&h=350, alt:Man in Yellow Jacket and
Black Pants Walking on Sidewalk

```

Bem mais interessante, né?

Embora funcione, o código que envolve o método **then** pode ser difícil de entender e de manter. Há uma construção chamada **async/wait** (também semelhante àquela de Javascript) que nos permite escrever código assíncrono num formato “sequencial”, mais fácil de entender, manter e depurar. Começamos marcando a função que fará uso da construção como **async**.

```

...
void obterImagem() async {
  var url = Uri.https(
  ...
}
```

A seguir, deixamos de chamar a função **then** e “aguardamos” pelo resultado da computação demorada, guardando seu resultado de modo aparentemente sequencial. Para isso, marcamos a chamada à função que produz um Future com **await**. Veja.

```

void obterImagem() async {
  var url = Uri.https(
    'api.pexels.com',
    '/v1/search',
    {'query': 'people', 'page': '1', 'per_page': '1'},
  );
  var req = http.Request('get', url);
  req.headers.addAll(
  {
    'Authorization':
    'chave',
  },
);

final result = await req.send();
if (result.statusCode == 200) {
  final response = await http.Response.fromStream(result);
  var decodedJSON = json.decode(response.body);
}
```

```

    var imagem = ImageModel.fromJson(decodedJSON);
    print(imagem);
} else {
    print('Falhou!');
}
}

```

(Criando uma lista de ImageModel) Sempre que tivermos uma nova imagem, ela será representada por uma instância de ImageModel e armazenada em uma lista. Essa atualização de dados envolve a atualização da tela. Isso quer dizer que a lista será uma **variável de estado**. Veja.

```

...
class AppState extends State<App> {
    int numeroImagens = 0;
    List<ImageModel> imagens = [];
...

```

A cada clique, adicionamos uma nova instância de ImageModel à lista.

```

...
final result = await req.send();
if (result.statusCode == 200) {
    final response = await http.Response.fromStream(result);
    var decodedJSON = json.decode(response.body);
    var imagem = ImageModel.fromJson(decodedJSON);
    imagens.add(imagem);
} else {
...

```

Lembre-se, entretanto, que estamos lidando com uma variável de estado. Assim, precisamos fazer a atualização de maneira adequada, usando o método **setState**. Ele se encarrega de atualizar a tela quando for a hora.

```

...
    final result = await req.send();
    if (result.statusCode == 200) {
        final response = await http.Response.fromStream(result);
        var decodedJSON = json.decode(response.body);
        var imagem = ImageModel.fromJSON(decodedJSON);
        setState(() {
            imagens.add(imagem);
        });
    } else {
        print('Falhou!');
    }
...

```

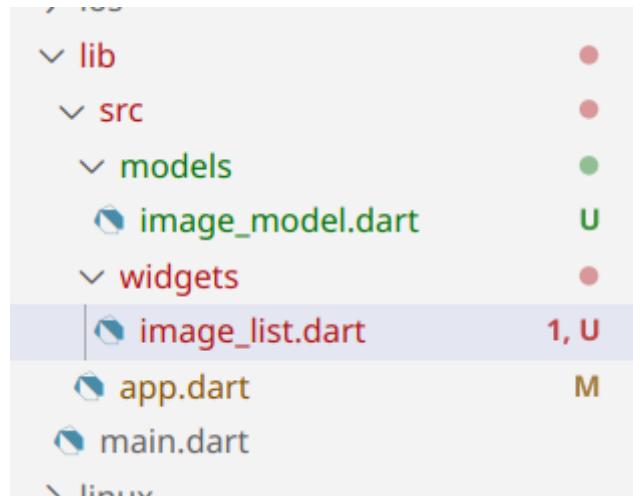
Neste ponto, a nossa aplicação merece nova refatoração. Observe como a classe **AppState** está bem grandinha.

Exercício.

1. Crie uma pasta **widgets**, subpasta de src.
2. Crie um arquivo chamado **image_list.dart** na pasta widgets.
3. Importe a biblioteca material de flutter.
4. Importe a classe **ImageModel**.
5. Defina uma classe chamada **ImageList**. Ela deve passar no teste É-UM **StatelessWidget**. Já já vemos por que ele não é um StatefulWidget.
6. Defina o método build: ele devolve Widget e recebe BuildContext.
7. No arquivo **app.dart**, importe o arquivo **image_list.dart**.
8. Ajuste a propriedade **body** de Scaffold. Ele deixa de exibir um Text e passa a exibir um ImageList.
9. Ainda em **app.dart**, ao construir um ImageList, passe como parâmetro a lista de ImageModel. Isso causará um erro em tempo de compilação.
10. No arquivo **image_list.dart** defina uma variável de instância chamada **imagens**. Ela deve ser do tipo **List<ImageModel>**. Já que o widget é sem estado, ela deve ser marcada final.
11. No arquivo **image_list.dart**, escreva um construtor que recebe uma lista de ImageModel.

Nota. O Widget que criamos é “sem estado” pois uma nova instância dele é construída a cada atualização de tela. Cada nova instância recebe, como parâmetro do construtor, a nova lista a ser exibida, já atualizada. A atualização da lista fica a cargo do Widget principal. Por essa razão, ele é “com estado”

Veja como fica.



```
//arquivo image_list.dart
import 'package:flutter/material.dart';
import '../models/image_model.dart';

class ImageList extends StatelessWidget {
  final List<ImageModel> imagens;
  ImageList(this.imagens);
  @override
  Widget build(BuildContext context) {
    ...
  }
}
```

Arquivo **app.dart**.

```
import 'models/image_model.dart';
import 'widgets/image_list.dart';
import 'package:flutter/material.dart';
import 'dart:convert';
import 'package:http/http.dart' as http;
...
@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: const Text("Minhas Imagens"),
            ),
            floatingActionButton: FloatingActionButton(
                child: const Icon(Icons.add),
                onPressed: obterImagen,
            ),
            body: ImageList(imagens),
        ) );
}
```

(Exibição de widgets com ListView) Um Widget do catálogo oficial capaz de exibir uma lista de widgets se chama ListView. Veja a sua documentação.

<https://api.flutter.dev/flutter/widgets/ListView-class.html>

Observe que ListView possui alguns construtores.

Flutter > widgets > ListView class

Search API Docs

CLASSES

- AbsorbPointer
- Accumulator
- Action
- ActionDispatcher
- ActionListener
- Actions
- ActivateAction
- ActivateIntent
- Align
- Alignment
- AlignmentDirectional
- AlignmentGeometry
- AlignmentGeometryTwe...
- AlignmentTween
- AlignTransition
- AlwaysScrollableScrollIP...
- AlwaysStoppedAnimation
- AndroidView
- AndroidViewSurface
- Animatable
- AnimatedAlign

Constructors

```
ListView({Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, bool addAutomaticKeepAlives = true, bool addRepaintBoundaries = true, bool addSemanticIndexes = true, double? cacheExtent, List<Widget> children = const <Widget>[], int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge})
```

Creates a scrollable, linear array of widgets from an explicit List.

```
ListView.builder({Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, required NullableIndexedWidgetBuilder itemBuilder, ChildIndexGetter? findChildIndexCallback, int? itemCount, bool addAutomaticKeepAlives = true, bool addRepaintBoundaries = true, bool addSemanticIndexes = true, double? cacheExtent, int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge})
```

Creates a scrollable, linear array of widgets that are created on demand.

```
ListView.custom({Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, required SilverChildDelegate childrenDelegate, double? cacheExtent, int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge})
```

Creates a scrollable, linear array of widgets with a custom child model.

const

Observe a descrição daquele que aparece primeiro na lista.

Creates a scrollable, linear array of widgets from an explicit List.

Já o segundo, chamado **builder**, é descrito assim.

Creates a scrollable, linear array of widgets that are created on demand.

O texto “on demand” faz toda a diferença. Se temos uma lista pequena, digamos com 5 elementos, o primeiro construtor resolve bem o problema. Ele constrói os 5 Widgets de para exibição de cada elemento de uma vez. Porém, se tivermos 5 milhões de itens para exibir, não podemos construir um Widget para cada item assim que a lista for construída. A melhor abordagem é fazer a construção **sob demanda**, conforme o usuário navega na lista. Por isso, vamos utilizar o construtor chamado **builder**.

No arquivo **image_list.dart**, vamos construir e devolver um ListView a partir do método build.

Os parâmetros que especificamos são os seguintes:

- **itemCount**: número de elementos na lista.
- **itemBuilder**: uma função que explica a forma como cada item da lista deve ser exibido. Ela recebe um objeto do tipo **BuildContext**, chamado **context**, e um objeto do tipo **int**, chamado **index**. Ela devolve um Widget que exibe o item na posição index da lista de imagens.

Exercício.

1. Faça o método build de ImageList devolver um ListView, construído pelo construtor chamado builder.
 2. Entregue os dois parâmetros a ele: itemCount e itemBuilder.
- itemCount deve ser o número de imagens existentes na lista.
 - itemBuilder deve ser uma função que recebe um BuildContext e um int.
 - Faça a função associada ao parâmetro itemBuilder devolver um Widget textual que exibe a representação textual do ImageModel contido na posição index.

Veja como fica.

```
//arquivo image_list.dart
import 'package:flutter/material.dart';
import '../models/image_model.dart';

class ImageList extends StatelessWidget {
    final List<ImageModel> imagens;

    ImageList(this.imagens);

    @override
    Widget build(BuildContext context) {
        return ListView.builder(
            itemCount: imagens.length,
            itemBuilder: (BuildContext context, int index) {
                return Text(imagens[index].toString());
            },
        );
    }
}
```

Teste a aplicação clicando algumas vezes no FloatingActionButton. Veja o resultado esperado.



Exercício. Observe que, a cada clique, a aplicação exibe o mesmo texto e a mesma URL. Faça o ajuste necessário para que ela exiba um novo item a cada clique. Lembre-se de usar o número de imagens, variável de estado da classe AppState.

Veja como fica.

```

class AppState extends State<App> {
    //começamos de 1 pois não há página 0 na Pexels
    int numeroImagens = 1;
    List<ImageModel> imagens = [];
    void obterImagem() async {
        var url = Uri.https(
            'api.pexels.com',
            '/v1/search',
            //número da página requisitada igual ao valor existente em
            numeroImagens
            {'query': 'people', 'page': '$numeroImagens', 'per_page': '1'},
        );
        var req = http.Request('get', url);
        req.headers.addAll(
        {
            'Authorization':
            'chave',
        },
    );
    final result = await req.send();
    if (result.statusCode == 200) {
        final response = await http.Response.fromStream(result);
        var decodedJSON = json.decode(response.body);
        var imagem = ImageModel.fromJSON(decodedJSON);
        setState(() {
            //incremento a cada clique
            numeroImagens++;
            imagens.add(imagem);
        });
    }
    else {
        print('Falhou!');
    }
}

```

(Exibindo imagens com o Widget Image) Há um widget chamado Image capaz de fazer a exibição de imagens. Veja a sua documentação.

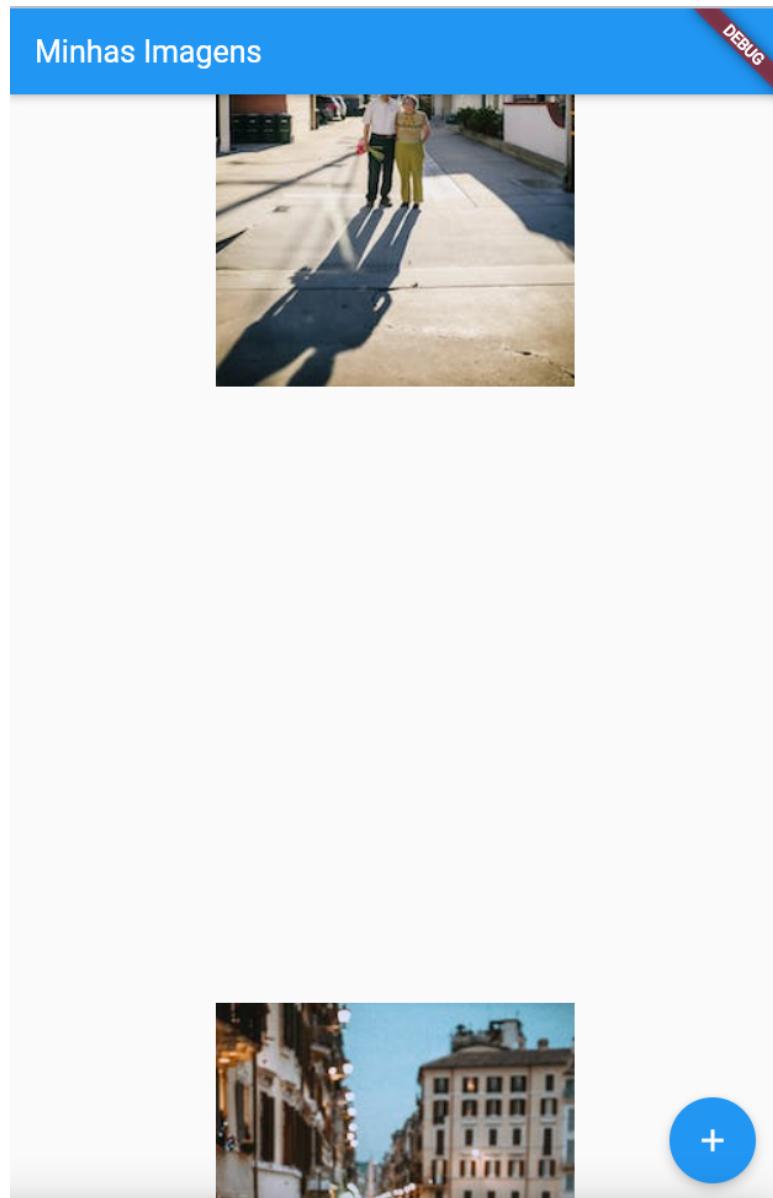
<https://api.flutter.dev/flutter/widgets/Image-class.html>

Exercício. Encontre o construtor apropriado para exibir uma imagem vinda da Internet. Lembre-se de utilizar a propriedade url do ImageModel.

Veja como fica. Estamos no arquivo **image_list.dart**.

```
...
@Override
Widget build(BuildContext context) {
    return ListView.builder(
        itemCount: imagens.length,
        itemBuilder: (BuildContext context, int index) {
            return Image.network(imagens[index].url);
        },
    );
}
...
```

Teste novamente a aplicação e veja o resultado, após clicar algumas vezes no FloatingActionButton.



Observe que as imagens aparecem mas elas estão distantes umas das outras. Há alguns ajustes de posicionamento, borda, margem, padding etc a serem realizados.

Exercício. Faça ajustes na aplicação para que as imagens fiquem “próximas” umas das outras. Também não permita que elas fiquem “coladas” às bordas da tela do dispositivo. Para isso, estude sobre leiautes.

<https://docs.flutter.dev/ui/layout>

Estude também sobre bordas.

<https://docs.flutter.dev/ui/layout#container>

Exiba a descrição de cada imagem logo abaixo dela.

Acima da lista de imagens, exiba o texto: Você tem n imagens, sendo n o número de imagens atual.

Troque o leiaute da aplicação. Passe a exibir duas imagens por linha.

Referências

Dart programming language | Dart. Google, 2023. Disponível em <<https://dart.dev/>>. Acesso em setembro de 2023.

Flutter - Build apps for any screen. Google, 2023. Disponível em <<https://flutter.dev/>>. Acesso em setembro de 2023.