

# Curso de Desenvolvimento Android com Flutter

## 1 - Linguagem Dart

**NEMOBIS**

Vinicius Takeo Friedrich Kuwaki



## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

# ● Introdução

- Antes de começarmos a criar aplicativos no Flutter, precisamos aprender a fundo a linguagem base do framework: o **Dart**;
- O Dart é o **núcleo do Flutter**, precisamos dela para implementar toda a lógica da nossa interface gráfica, além da **lógica de negócio** do aplicativo;
- Veremos nessa aula os pontos mais importantes e que mais usaremos no Flutter;



**Figura 1:** Separação das lógicas em um aplicativo.

## ● Introdução

- Precisamos primeiro entender o que é a linguagem **Dart**;
- Dart é uma **linguagem de script**;
- Otimizada para **interfaces de usuário** (tais como o Flutter);
- Compila para a arquitetura do qual o programador quer utilizar;
- Otimizando o código onde é possível, diferente de outras linguagens do mesmo tipo;
- Dart pode ser **compilada ou executada**;
- Utilizaremos **Dart executada**, visto que nosso foco não é desenvolver algoritmos sofisticados.

## ● Introdução

- Lançada pela Google em 2011;
- Dart é **multiparadigma**:
  - Orientada a objetos;
  - Paralela;
  - Funcional;
- Fortemente influenciada por Java e Javascript;
- Tipagem estática, forte e inferida;



**Figura 2:** Logo da Linguagem Dart.

## ● Introdução

- Utilizaremos o Dart instalado para executar nossos códigos;
- É possível também acessar o compilador online [Dartpad](#);
- Para executar um código Dart, basta executar no terminal **dart codigo.dart**;
- Como utilizaremos o VSCode, basta abrir um terminal na pasta do projeto executar o comando;

## ● Introdução

- Arquivos para a linguagem Dart terão a extensão **.dart**;
- Além de seguirem o padrão **snake\_case**:
  - Todos caracteres minúsculos;
  - Separados por underline “\_”;
- Todos os comandos em Dart, assim como C, Java e outras linguagens terminam com **ponto e vírgula**;
- **Comentários** são trechos de código seguidos de **duas barras**;

## ● Introdução

- O trecho de código a seguir é o “Hello World” em Dart;
- Para executá-lo, basta escrever o código em um arquivo **main.dart**;
- E executar o comando **dart main.dart** em um terminal estando na mesma pasta;

```
void main(){  
    print("Hello World!");  
}
```

- Se a extensão Dart estiver instalada no VSCode, um botão **run** aparecerá em cima da função **main()**, permitindo executar o código;



## ● Seções

Introdução

**Tipos de Dados**

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

## ● Tipos de Dados

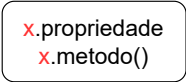
- Começaremos falando a respeito dos **tipos de dados** em Dart;
- Dart na verdade possui vários tipos de dados nativos;
- Veremos apenas os **principais** e que mais usaremos no curso:

## ● Tipos de Dados

- **int**: representam números exatos;
- **double**: representam números decimais;
- **bool**: representam condições lógicas: verdadeiro ou falso;
- **String**: representa um conjunto de caracteres;
- **List<T>**: agrupam elementos;
- **Map<K,V>**: mapeiam elementos;
- **dynamic**: permitem descobrir o tipo de dado em tempo de execução;
- **Future<T>**: permite trabalhar com operações assíncronas;
- **Function**: trata funções como se fossem variáveis;

## Tipos de Dados

- É interessante comentar que todos esses tipos de dados são **objetos**;
- Veremos a definição de objeto a frente em detalhes;
- Entretanto, é interessante comentar que objetos possuem **métodos** e **atributos/propriedades**;



A diagram consisting of a rounded rectangle with a thin black border. Inside the rectangle, the text 'x.propriedade' is on the top line and 'x.metodo()' is on the bottom line. Both lines are left-aligned. The text is black, but the 'x' at the beginning of each line is red.

**Figura 3:** Acessando propriedades e métodos de uma variável/objeto.

## Tipos de Dados - int

- O tipo **int** representa inteiros;
- Isto é, valores exatos;
- Exemplo de declaração de inteiros:
- Note o uso da palavra reservada **var**, através dela é possível inferir o tipo de dado;
- É possível utilizar as “funções” prontas dos inteiros;
- Na [documentação](#) é possível ver a lista completa;

```
int x = 3;  
var y = 4;
```

## ● Tipos de Dados - int

- A documentação **auxilia** muito na hora da **escrita de código**;
- Além de a documentação do Flutter/Dart ser bem mais **agradável** que a de muitas linguagens;
- Note que é comum encontrar na documentação **propriedades e métodos**;
- Veremos na parte de classes e objetos o que cada um significa a fundo;

## Tipos de Dados - int

- Porém, no momento é interessante saber que:

Propriedade  
variável interna

`x.isNegative`  
true se o inteiro é menor que zero

Método  
função interna

`x.gdc(3)`  
retorna o MDC entre x e 3

**Figura 4:** Propriedades e métodos e exemplos para inteiros.

## ● Tipos de Dados - int

- Exemplo de uso de **propriedades**:

```
int x = 10;  
print(x.isNegative);
```

- Ao executar o código acima o resultado será **false**, visto que x é um número positivo;



## ● Tipos de Dados - int

- Exemplo de uso de **métodos**:

```
int x = 20;  
print(x.gcd(50));
```

- Ao executar o código acima o resultado será **10**, visto que o MDC entre 20 e 50 é 10;

## ● Tipos de Dados - int

- Dart possui os seguintes operadores que utilizaremos:
  - Adição (+);
  - Subtração (-);
  - Multiplicação (\*)
  - Divisão (/);
  - Resto da divisão (%);
  - ... (ver documentação)
- Possui também um método bem importante que utilizaremos a frente;
  - **int.parse(String x)**: transforma uma String em um inteiro;
  - `int x = int.parse("1309");`

## ● Tipos de Dados - double

- Para representar números com casas decimais vamos utilizar a classe **double**;
- Ela também possui uma série de **propriedades e métodos** (veja na [documentação](#)) bem similares aos inteiros;
- Diferentemente de outras linguagens, Dart não possui o tipo **float**, apenas o **double**!

```
double x = 1.3;  
var y = 0.9;
```

## ● Tipos de Dados - bool

- Variáveis **booleanas** podem assumir dois valores:
  - **true**;
  - **false**;
- Tipos booleanos são utilizados em estruturas de **seleção** e **repetição**;

```
bool x = true;  
var y = false;
```

## ● Tipos de Dados - String

- `String`'s representam um **conjunto de caracteres**;
- Diferentemente de outras linguagens, Dart não possui o tipo **char**, apenas `String`;
- Em Dart, as `String`'s podem ser declaradas usando tanto **aspas duplas** como **aspas simples**;

```
String a = "uma palavra";  
String b = 'outra palavra';  
var c = 'mais uma palavra';
```

## ● Tipos de Dados - String

- String's em Dart podem **concatenar** valores dentro, utilizando o simbolo dólar (\$):
- No trecho de código ao lado, a variável x será substituída por 3 dentro da String y;

```
int x = 3;  
String y = "O valor da variavel x eh $x";
```

## ● Tipos de Dados - String

- String's também possuem propriedades e métodos interessantes que utilizaremos, veja a [documentação](#) para a lista completa;
- Alguns que utilizaremos:
  - **int length**: número de caracteres da String;
  - **bool isEmpty**: verifica se o número de caracteres é 0;
  - ...
- Utilizaremos muito String's visto que a **entrada de dados** no Dart e no Flutter é através de **String**;

```
String nome = "Vinicius";  
print(nome.length);
```

- Sempre precisaremos convertê-las para int, double, etc.

## Tipos de Dados - List

- Dart não possui **arrays** de tamanho fixo como muitas linguagens;
- Ao invés, possui a classe **List** que agrupa elementos;
- Note que a notação da List normalmente é **List<T>**, o T significa que ela pode ser de **qualquer tipo de objeto**;
- Logo, podemos ter lista de inteiros, String's, etc;

"Macarrão"	"Batata"	"Presunto"	"Repolho"
0	1	2	3

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

**Figura 5:** Exemplos de listas de String's e inteiros.



## ● Tipos de Dados - List

```
List<int> vazia = [];  
List<int> inteiros = [1,2,3];  
List letras = ["a","b","c"];  
var booleanos = [true,false,false];
```

- Note que podemos declarar **listas vazias** ou com elementos;
- Elementos são sempre **separados por virgulas**;

## ● Tipos de Dados - List

- A classe possui uma série de **propriedades e métodos** interessantes de se comentar;
- Veremos algumas:
  - **T first**: acessa o primeiro elemento;
  - **bool isEmpty**: retorna true se a lista não possui elementos;
  - **T last**: acessa o último elemento;
  - **add(T valor)**: adiciona um elemento ao final da lista;
  - **void clear()**: elimina todos os elementos da lista;
  - **bool contains(T objeto)**: verifica se um objeto está na lista, retornando true caso esteja;
  - **T elementAt(int posicao)**: retorna o elemento na posição passada como parâmetro;
  - **void insert(int posicao, T elemento)**: adiciona um elemento em uma posição específica da lista;

## ● Tipos de Dados - List

- Veremos algumas:
  - **void remove(T objeto)**: remove o objeto da lista;
  - **void removeAt(int posicao)**: remove o objeto de uma posição específica da lista;

## Tipos de Dados - List

"Macarrão"	"Batata"	"Presunto"	"Repolho"
0	1	2	3



`add("Hamburger")`

"Macarrão"	"Batata"	"Presunto"	"Repolho"	"Hamburger"
0	1	2	3	4

**Figura 6:** Adicionando um item a uma lista com o método `add()`.

## Tipos de Dados - List

"Macarrão"	"Batata"	"Presunto"	"Repolho"
0	1	2	3



removeAt(2)

"Macarrão"	"Batata"	"Repolho"
0	1	3

**Figura 7:** Removendo um item da lista com o método **removeAt()**.

## ● Tipos de Dados - List

- Exemplificando o uso de algumas dessas propriedades e métodos:

```
List<int> inteiros = [1,2,3,4];  
inteiros.add(5);  
inteiros.remove(3);  
inteiros.removeAt(0);  
print(inteiros.last);
```

## Tipos de Dados - List

- Para acessar um elemento em uma posição específica da lista, utilize a notação dos colchetes
- Lembre-se que a lista começa a contar em 0 e vai até  $n-1$ ;

```
List<int> inteiros = [1,2,3,4];  
print(inteiros[2]);
```

"Macarrão"	"Batata"	"Presunto"	"Repolho"
0	1	2	3



lista[2]

"Presunto"

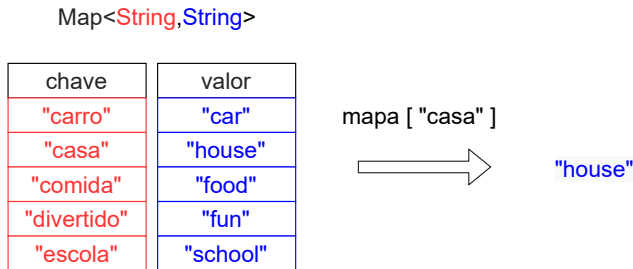
**Figura 8:** Acessando uma posição da lista.

## ● Tipos de Dados - Map

- Um dos tipos de dados que utilizaremos é o **Map**;
- A notação do Map é: **Map<K,V>**;
- A ideia central do Map é, como o próprio nome já diz, mapear valores;
- Cada K (key ou chave) é associado a um V (value ou valor);

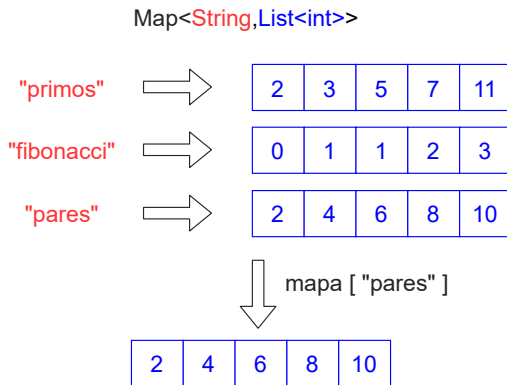


## Tipos de Dados - Map



**Figura 9:** Exemplo de mapa que mapeia Strings para Strings, agindo como se fosse de fato um "tradutor".

## Tipos de Dados - Map



**Figura 10:** É possível mapear qualquer tipo de dado para qualquer tipo de dado, por exemplo, String para listas.

## ● Tipos de Dados - Map

- No exemplo ao lado, mapearemos no **Dart** os classificados de um campeonato;
- Note que cada inteiro está a esquerda do dois pontos e cada String está a direita;
- Logo, cada inteiro foi mapeado para uma String;

```
Map<int,String> campeoes = {  
  1 : "Gabriela",  
  2 : "Marcos",  
  3 : "Julia"  
};
```

## Tipos de Dados - Map

- A utilidade de se fazer isso é poder **acessar valores**;
- O Map pode acessar valores passando a chave como se fosse uma List (usando a sintaxe dos colchetes);
- Veja no exemplo ao lado:
- Ao executar o código temos que apenas a lista associada a chave 'primos' é exibida;

```
Map<String,List<int>> mapa = {  
    "primos" : [2,3,5,7,11],  
    "fibonacci" : [0,1,1,2,3],  
    "pares" : [2,4,6,8,10],  
};  
print(mapa["primos"]);
```

## ● Tipos de Dados - Map

- Map's são usados muito no Dart e no Flutter;
- Veja na [documentação](#) a lista completa de propriedades e métodos.
- Exemplo de alguns:
  - **bool isEmpty:** retorna true se não há elementos no Map;
  - **int length:** retorna o número de pares (chave/valor) armazenados no Map;
  - **void clear():** remove todos os elementos do Map;
  - **void remove(T chave):** remove a chave e o elemento associado a essa chave do Map;
- Para declarar Map's em Dart, assim como feito nos slides anteriores, basta colocar os elementos entre **chaves**, separados por **virgula**;
- Já para adicionar um novo elemento, basta acessar uma chave nova e colocar o valor lá:

## Tipos de Dados - Map

- Note que declaramos um Map contendo dois pares chave/valor;
- E em seguida **sobreescrevemos** o segundo (trocamos o Marcos pela Julia);
- Após, colocamos a Letícia na terceira posição;

```
Map<int,String> campeoes = {  
    1 : "Gabriela",  
    2 : "Marcos",  
};  
campeoes[2] = "Julia";  
campeoes[3] = "Leticia";
```

## ● Tipos de Dados - dynamic

- O próximo tipo de dados que veremos é **dynamic**;
- Basicamente é um tipo de dados que é **inferido** em tempo de execução;
- Ou seja, se uma função recebe algum parâmetro do tipo dynamic, ele pode receber **qualquer tipo de dados**;
- Porém, se tentar acessar um **método/propriedade** que aquele tipo **não possui**, irá **lançar um erro** e parar a execução do programa;

```
dynamic x = 3;  
dynamic y = 7.4;  
dynamic z = "Vinicius";  
print(z.length);
```

## ● Tipos de Dados - dynamic

- As vezes será necessário transformar um tipo de dados **dynamic** para outro;
- Para isso utilizaremos o operador **as**;
- Tal operador permite realizar o chamado **casting**:

```
dynamic x = 3;  
int y = x as int;
```



## ● Seções

Introdução

Tipos de Dados

**Estruturas da Linguagem**

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

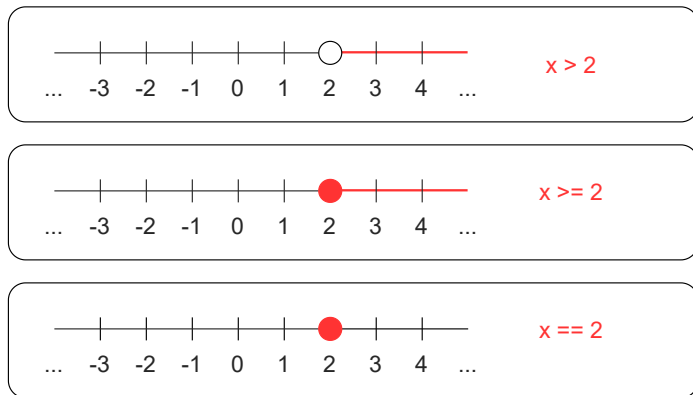
## ● Estruturas da Linguagem

- Agora que falamos dos tipos de dados, vamos ver as **estruturas da linguagem**;
- Como o pré-requisito do curso é linguagem de programação/algoritmos, a grande maioria já está familiarizado com as estruturas de pelo menos uma linguagem, seja C, Python, etc;
- Dart mistura um pouco elas, veremos agora cada uma delas:
  - **Estruturas de seleção** (if, else, else if, switch case);
  - **Estruturas de repetição** (while, for, for in, forEach);
  - **Funções**;

## ● Estruturas da Linguagem - Estruturas de Seleção

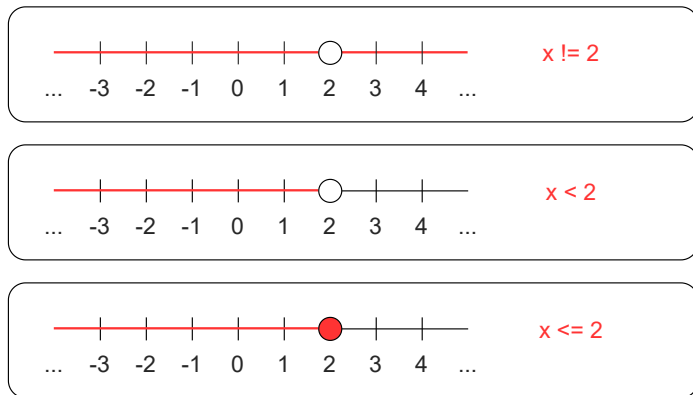
- Começaremos pelas **estruturas de seleção**;
- Antes de falarmos apropriadamente delas, precisamos revisar as operações lógicas e **operadores lógicos**;
- Dart possui os operadores:
  - **>**: verifica se o lado esquerdo é **maior** que o direito;
  - **>=**: verifica se o lado esquerdo é **maior ou igual** que o direito;
  - **==**: verifica se o lado esquerdo é **igual** ao direito;
  - **!=**: verifica se o lado esquerdo **diferente** do direito;
  - **<**: verifica se o lado esquerdo é **menor** que o direito;
  - **<=**: verifica se o lado esquerdo é **menor ou igual** que o direito;
- É a partir desses operadores que podemos realizar comparações entre valores;

## ● Estruturas da Linguagem - Estruturas de Seleção



**Figura 11:** Alguns operadores lógicos.

## ● Estruturas da Linguagem - Estruturas de Seleção

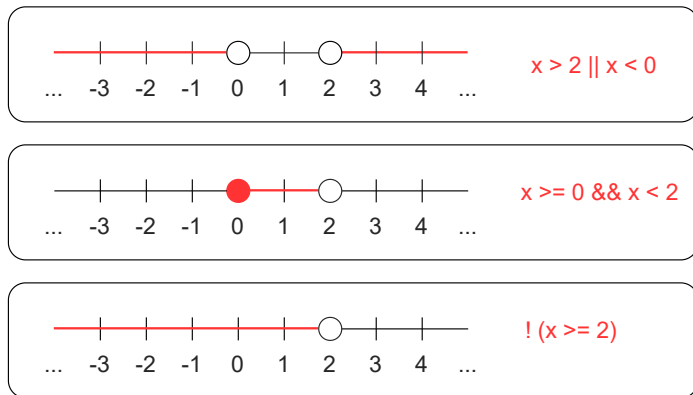


**Figura 12:** Alguns operadores lógicos.

## ● Estruturas da Linguagem - Estruturas de Seleção

- Também é possível combinar expressões usando os operadores:
  - || (or): retorna true se um dos lados é true;
  - && (and): retorna false se um dos lados é false;
  - ! (not): inverte o valor da expressão;

## ● Estruturas da Linguagem - Estruturas de Seleção



**Figura 13:** Alguns operadores lógicos.

## ● Estruturas da Linguagem - Estruturas de Seleção

- Muitas vezes é possível evitar usar expressões lógicas muito grandes, basta **agrupar** em vários if's;
- Veremos agora como usá-las em Dart;
- Basta indicar uma **condição lógica** entre **parenteses** e os **comandos** a ser executados **entre chaves**;

```
if( x == 3 ){  
    print(' x eh igual a 3 ');  
}
```

```
if ( CONDICAO ){  
  
    comandos...  
  
}
```

**Figura 14:** Comando if.



## ● Estruturas da Linguagem - Estruturas de Seleção

- É possível usar o comando **else** em Dart, se a condição dentro do **if** for falsa, o comando entre chaves do **else** será executado:

```
if( x == 3 ){  
    print(' x eh igual a 3 ');  
}else{  
    print(' x eh diferente de 3');  
}
```


```
if ( CONDICAO ) {  
    comandos...  
  
} else {  
    comandos...  
  
}
```

**Figura 15:** Comando else.

## ● Estruturas da Linguagem - Estruturas de Seleção

- Dart também implementa o **operador ternário**, inclusive ele é muito usado no Flutter;

```
if ( CONDICAO ) {  
    comandos...  
} else {  
    comandos...  
}
```



```
CONDICAO ? comando : comando
```

**Figura 16:** Operador ternário.

## ● Estruturas da Linguagem - Estruturas de Seleção

- Veja a comparação entre o **if/else** e o **operador ternário**;
- As linhas 0-4 são equivalentes a linha 6;
- Veremos mais exemplos a frente de seu uso prático;

```
if( x == 3 ){  
    print(' x eh igual a 3 ');  
}else{  
    print(' x eh diferente de 3');  
}  
  
x == 3 ? print(' x eh igual a 3 ') : print(' x eh  
diferente de 3');
```

## ● Estruturas da Linguagem - Estruturas de Seleção

- É possível também incluir comandos intermediários entre o **if** e o **else**;
- As cláusulas **else if**;
- Basta incluir **outra condição entre parenteses** e **comandos entre chaves**:

```
if ( CONDICAO ) {  
    comandos...  
} else if ( CONDICAO2 ) {  
    comandos...  
} else {  
    comandos...  
}
```

**Figura 17:** Comando else if.

## ● Estruturas da Linguagem - Estruturas de Seleção

```
if( x == 3 ){  
    print( ' x eh igual a 3 ' );  
}else if( x == 4 ){  
    print( ' x eh igual a 4 ' );  
}else{  
    print( ' x eh diferente de 4 ' );  
}
```

## ● Estruturas da Linguagem - Estruturas de Seleção

- A última estrutura de seleção que veremos é o **switch case**;
- Basicamente ela vai determinar como tratar cada caso;
- Todos os casos terminam com um comando **break**;

```
int mes = 9;

switch(mes){
    case 1:
        print('Janeiro');
        break;
    case 9:
        print('Setembro');
        break;
    default:
        print('Outro mes');
        break;
}
```

## ● Estruturas da Linguagem - Estruturas de Seleção

- Note o uso do **default**, ele atua como se fosse o **else**, caso não entre em nenhum **case**, o comando especificado no **default** será executado;
- Agora veremos as estruturas de repetição;

## ● Estruturas da Linguagem - Estruturas de Repetição

- Na programação o tempo todo precisamos **repetir processos**;
- Uma das formas de realizar isso é através de **estruturas de repetição**;
- Dart possui as estruturas **while** e **for**;
- Começaremos pelo **while**;
- O **while** pode ser visto como um “**if que se repete até se tornar falso**”;
- Em Dart sua sintaxe é exatamente igual a do **if**, só que com a palavra **while**:

```
int i = 0;
while ( i < 10 ){
    print(i);
    i++;
}
```

- É importante definir um “passo” para garantir que o **while** atinja a sua condição de parada, evitando um **loop infinito**;
- No código acima, é o comando **i++**;



## ● Estruturas da Linguagem - Estruturas de Repetição

- Já o for possui três “argumentos”:
  - Condição inicial;
  - Condição de parada;
  - Passo;
- Cada um desses “argumentos” é separado por um ponto e vírgula no Dart;

```
for ( INICIO ; CONDICAO ; PASSO ) {  
    comandos...  
}
```

**Figura 18:** Sintaxe do comando for.

## ● Estruturas da Linguagem - Estruturas de Repetição

- Veja um exemplo equivalente ao **while** apresentado anteriormente:

```
int i = 0;
while ( i < 10 ){
    print(i);
    i++;
}
```

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

- A diferença crucial ali é o **escopo** da variável **i**;
- Note que no caso do **for** ela existe apenas para o **for**;
- Já no caso do **while** ela existe fora dele;

## ● Estruturas da Linguagem - Estruturas de Repetição

- Agora um exemplo **iterando** sobre uma lista;
- O tamanho da lista pode ser acessado usando a propriedade **length**:

```
List<int> lista = [1, 2, 3, 4, 5];  
  
for (int i = 0; i < lista.length; i++) {  
    print(lista[i]);  
}
```

## ● Estruturas da Linguagem - Estruturas de Repetição

- Existem também outras formas de iterar sobre uma lista;
- Apresentaremos mais duas delas;
- A primeira é usando o **for in**;
- Ao invés de termos uma variável para controlar a *i-ésima* posição da lista, vamos ter uma variável *x* para **assumir o valor** da *i-ésima* posição:

```
List<int> lista = [1, 2, 3, 4, 5];  
  
for (int x in lista) {  
    print(x);  
}
```

## ● Estruturas da Linguagem - Estruturas de Repetição

- A segunda for é usando o método **forEach()** da classe List;
- A classe **Map** também possui tal método;
- Porém, utilizar esse método as vezes pode causar alguns **problemas**;
- Visto que ele tentará executar vários passos do **for** ao mesmo tempo;
- Utilizando o conceito de **assincronismo**;
- Usando tal método, não há garantia de que os passos serão executados em sequência;
- Para aplicações complexas, isso pode ser um problema;

## ● Estruturas da Linguagem - Estruturas de Repetição

- Veremos agora dois exemplos usando listas e mapas;
- Começaremos pelas listas;
- O método **forEach()** pede como parâmetro uma **função para ser executada**;
- Podemos declará-la no momento da chamada:
- Também é possível declarar a função a ser passada para o **forEach()** de outra forma, porém veremos isso quando falarmos de funções;

```
List<int> lista = [1, 2, 3, 4, 5];
```

```
lista.forEach((x) {  
    print(x);  
});
```

## ● Estruturas da Linguagem - Estruturas de Repetição

- Para iterarmos sobre mapas usando o **forEach()** precisamos de uma função com dois parâmetros (key,value):

```
Map<String, int> mapa = {  
    'primeiro': 1,  
    'segundo': 2,  
    'terceiro': 3,  
};  
  
mapa.forEach((key, value) {  
    print('Mapa[$key] = $value');  
});
```

## ● Estruturas da Linguagem - Funções

- Agora que vimos como usar as partes fundamentais de uma linguagem de programação, vamos **declarar funções**;
- Funções tornam o código mais **organizado** e muitas vezes são utilizadas para **reaproveitar código**;
- Funções **podem ou não possuir um tipo de retorno**, é uma boa prática indicar o tipo de retorno mesmo quando não há (quando o retorno é void);
- Funções também **podem ou não possuir parâmetros de entrada**;

```
TipoRetorno nome (tipo1 nome1, ... , tipoN nomeN) {  
    comandos...  
    return nomeX;  
}
```

**Figura 19:** Sintaxe de funções.



## ● Estruturas da Linguagem - Funções

- Veja um exemplo de uma função de soma:

```
int soma(int x, int y){  
    return x + y;  
}
```

## ● Estruturas da Linguagem - Funções

- Essa mesma função pode ser **omitida os tipos de dados**:

```
soma(x,y){  
  return x + y;  
}
```

## ● Estruturas da Linguagem - Funções

- Variáveis declaradas **dentro** de funções **só existirão lá**;
- Funções também podem ser **usadas como variáveis**;
- Um exemplo disso ao usar o **forEach()**;
- Poderíamos declarar a função fora do **forEach()**:

```
void main(){  
    List<int> lista = [1, 2, 3, 4, 5];  
  
    lista.forEach(imprime);  
}  
  
void imprime(int x){  
    print(x);  
}
```

## ● Estruturas da Linguagem - Funções

- Porém, sabíamos a priori que a função **forEach()** da classe List precisava de uma função do tipo **void** que recebe um inteiro;
- Para a classe Map é necessário uma função que recebe a **chave e o valor**:

```
void main(){  
    Map<String, int> mapa = {  
        'primeiro': 1,  
        'segundo': 2,  
        'terceiro': 3,  
    };  
  
    mapa.forEach((key, value) {  
        print('Mapa[$key] = $value');  
    });  
}  
  
void imprime(String chave, int valor){  
    print('Chave: $chave — Valor: $valor');  
}
```

## ● Estruturas da Linguagem - Funções

- No Dart, assim como no Javascript é possível declarar **arrow functions**;
- Basicamente são funções que **suprimem** o uso da palavra **return** e as **chaves**:

```
TipoRetorno nome (tipo1 nome1, ... , tipoN nomeN) {  
    comandos...  
    return nomeX;  
}  
↓  
TipoRetorno nome (tipo1 nome1, ... , tipoN nomeN) => return COMANDO;
```

**Figura 20:** Sintaxe das Arrow Functions.

```
int soma(x,y) => x + y;
```

## ● Estruturas da Linguagem - Funções

- Funções em Dart também podem possuir **parâmetros opcionais**;
- Porém, estes **devem possuir valores padrões**;

```
main(List<String> args) {  
  print(soma(3, 4));  
  print(soma(3));  
}  
  
int soma(int x, [int y = 8]) => x + y;
```

## ● Estruturas da Linguagem - Funções

- Em Dart funções podem possuir **parâmetros nomeados**;
- Utilizaremos muito o conceito de parâmetros nomeados no **Flutter**;
- Quase todas as **classes utilizam** em suas funções;
- Basicamente parâmetros nomeados permitem que a função seja chamada informando para **qual parâmetro o valor passado é**;
- Permitindo que os parâmetros sejam passados até **fora de ordem**;

## ● Estruturas da Linguagem - Funções

- Entretanto, caso um parâmetro seja nomeado, ele precisa de **um valor padrão**, pois se torna **opcional**;
- É possível tornar um parametro obrigatorio, usando a palavra **required**:

```
int soma({required int x, int y = 7}) => x + y;
```



## ● Estruturas da Linguagem - Funções

- Para chamar uma função com parâmetros nomeados é necessário **especificá-los como se fosse um Map**:
- Note que há uma virgula a mais após o parâmetro **y**;
- Tal fato, possibilita formatar o código (usando as extensões do VSCode) para torná-lo mais legível ao desenvolvedor.

```
main(List<String> args) {  
    print(soma(  
        x: 3,  
        y: 4,  
    ));  
}
```

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

**Null-Safety**

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

## ● Null-Safety

- Null-Safety é um recurso adicionado na versão 2 do Dart;
- Basicamente seu propósito é reduzir o número de erros com variáveis null no Dart;
- Parte-se do princípio agora que toda variável deve possuir um valor;
- Em outras palavras, não pode ser null:

```
String x = "Ola";
```

```
String y = null; // Erro de compilacao
```

## ● Null-Safety

- Porém, e se precisarmos utilizar um valor null?
- Para isso, o Dart introduziu o operador *question mark* (?):

```
String x = "Ola";
```

```
String? y = null; // Agora y pode ser null sem ser um erro de compilacao
```

## ● Null-Safety

- É importante salientar que:
  - `String`  $\rightarrow$  `String?` OK!
  - `String?`  $\rightarrow$  `String` ERRO!
- Para realizar a segunda conversão, Dart introduziu o operador *exclamation mark*;
- Tal operador “transforma” uma variável que pode ser null em uma null-safety:

```
String? x;  
String y = x!;
```

- Porém, utilize-o somente em último recurso, pois deve-se ter certeza que `x` não é null, caso contrário uma exceção será levantada.

## ● Null-Safety

- Abordagens clássicas ainda funcionam:

```
String? x;
```

```
if (x != null) String y = x;
```

## ● Null-Safety

- Dart ainda apresenta um terceiro operador “??”;
- Tal operador pode ser dito como: “atribua se diferente de null”;

```
String? x;
```

```
String y = x ?? "Ola";
```

## ● Null-Safety

- Há também uma nova-palavra reservada no Dart, chamada **late**;
- Porém, não a utilizaremos muito;
- Caso deseje saber mais, veja uma explicação na [documentação oficial](#).



## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

**Exercícios**

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

## Exercícios

1. Crie uma função do tipo **int** chamada **expo** que receba dois **parâmetros nomeados** “a” e “b” sendo “a” obrigatório e “b” opcional. A função deve retornar o valor “a” elevado “b”, calcule usando um **laço de repetição**. Caso o parâmetro “b” não seja informado, utilize o valor 2.

2. Crie uma função do tipo **int** que receba três **parâmetros nomeados** e obrigatórios “a”, “b” e “op”. A função deve retornar, de acordo com o valor “op”, o resultado da soma, subtração, multiplicação e resto de divisão entre “a” e “b”. Utilize **switch case** para isso. Considere que:
- 1 - soma;
  - 2 - subtração;
  - 3 - multiplicação;
  - 4 - resto da divisão;

3. Crie uma função do tipo **void** chamada `traduz` que receba um **Map<String,String>** e um **booleano** “traduzir” que representa palavras em português mapeadas para o inglês. Dependendo do valor do booleano, ela irá imprimir todas as palavras do Map em português ou todas as palavras do Map em inglês. No slide seguinte é possível ver a execução da função e a **assinatura** da função.

## Exercícios

```
void traduz(Map<String, String> palavras, bool traduzir) {  
    // Sua implementacao  
}  
  
main(List<String> args) {  
    Map<String, String> frutas = {  
        'Banana': 'Banana',  
        'Strawberry': 'Morango',  
        'Orange': 'Laranja',  
        'Watermelon': 'Melancia'  
    };  
    print('Original:');  
    traduz(frutas, false);  
    print('\nTraduzido:');  
    traduz(frutas, true);  
}
```

Original :

Banana

Morango

Laranja

Melancia

Traduzido :

Banana

Strawberry

Orange

Watermelon

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

**Classes, Objetos e Enumeráveis**

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

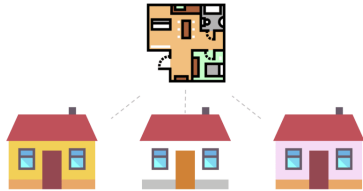
Singleton

## ● Classes e Objetos

- Assim como mencionamos nos slides anteriores, em Dart tudo são **classes** e **objetos**;
- Também falamos que Dart é **multi paradigma** e um desses paradigmas é o **orientado a objetos**;
- Isso significa na prática que é possível agrupar **objetos** e **gerar seus próprios tipos de dados**;
- Veremos um pouco nessa aula sobre os **principais conceitos** da orientação a objetos;
- Cujo uma das premissas é **reduzir e reaproveitar código**;

## ● Classes e Objetos

- Começaremos vendo o conceito de **classes**;
- Basicamente uma classe é um **modelo**;
- Pode ser definida como uma “planta”, uma **receita**;
- Ela define o que os **objetos podem possuir**;
- Sendo cada objeto uma “instancia” de uma classe;



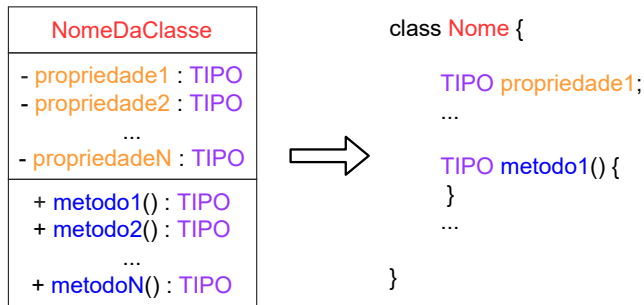
**Figura 21:** Analogia a uma *blueprint*.  
Fonte: (CHARANRAJ, 2018)



## ● Classes e Objetos

- De fato vimos esses conceitos na prática anteriormente;
- Por exemplo a classe **List**;
- Que possuía suas **propriedades e métodos**;
- Nós programadores apenas criávamos objetos (nossas variáveis) e utilizávamos-as;
- Agora veremos como **declarar** nossas próprias **classes**, contendo suas propriedades(atributos) e métodos;

## ● Classes e Objetos



**Figura 22:** Estrutura básica de uma classe e sua sintaxe de declaração.

## ● Classes e Objetos

- Toda classe em **Dart** irá seguir o padrão **Camel Case**:
  - Início de palavra com letra maiúscula;
  - Sem espaços;
- O **nome do arquivo** continua seguindo o padrão **snake case**:
  - Letras minúsculas;
  - Espaços substituídos por *underline*;

## ● Classes e Objetos

- É comum na orientação a objetos **classificarmos** as classes de acordo com suas **funcionalidades**;
- Cada “pasta” de um projeto será para uma finalidade;
- No momento vamos criar classes apenas para aprender os conceitos, então vamos modelar apenas **entidades/modelos**;
- Vamos colocá-las dentro de uma pasta chamada **models**;

## ● Classes e Objetos

- Dentro dessa pasta criaremos nossas classes que representam **entidades**;
- Começaremos com o exemplo mais clássico de todos, uma classe para representar uma **pessoa**;
- São infinitos as propriedades que podemos pensar de uma pessoa:
  - Nome;
  - Idade;
  - Altura;
  - CPF;
  - etc...

## ● Classes e Objetos

- Vamos colocar essas quatro em primeiro momento;
- Começaremos criando dentro da pasta **models** um arquivo **pessoa.dart**;
- **Sempre o nome das classes estará no singular**, visto que modelam uma entidade;

## ● Classes e Objetos

- Para declarar uma classe em Dart, vamos utilizar a palavra-reservada **class**;
- Dentro das chaves iremos declarar as suas propriedades/atributos:
- Em muitas linguagens orientadas a objeto, incluindo Dart, é comum o uso de atributos **privados**;
- Tais atributos podem ser acessados **apenas** por funções **dentro da classe**;

```
class Pessoa{  
    String nome = "";  
    int idade = 0;  
    double altura = 0;  
    int cpf = 0;  
}
```

## ● Classes e Objetos

- Poderíamos criar um atributo chamado **senha** na classe Pessoa;
- Que gostaríamos que **apenas a classe visualizasse** tal valor;
- Para isso, utilizamos o **underline** antes do nome do atributo:

```
class Pessoa{  
    String nome = "";  
    int idade = 0;  
    double altura = 0;  
    int cpf = 0;  
    String _senha = "";  
}
```



## ● Classes e Objetos

- Porém, como alguém poderia **alterar** o valor da propriedade senha se ela não acessível de **fora da classe**?
- Para isso vamos utilizar um conceito chamado **setter**;
- Em Dart existe uma palavra reservada **set** para isso;
- Entretanto seu objeto é apenas “fazer com que seja possível atribuir um valor ao atributo senha como se ela fosse pública”;
- Para declarar um **setter** é necessário especificar um nome e o corpo da função:

```
class Pessoa{  
  ...  
  String _senha = "";  
  
  void set senha(String nova) {  
    _senha = nova;  
  }  
}
```

## ● Classes e Objetos

- Agora é possível alterar a **propriedade privada** senha de **fora da classe**;
- Vamos instanciar uma Pessoa e testar;
- Para isso, precisamos **importar** a classe Pessoa em nosso arquivo **main.dart**;
- Normalmente o **VSCode** aponta um erro e **sugere a importação**;
- Para instanciar uma Pessoa podemos usar o operador **new**;
- Porém, em Dart **seu uso é opcional**:

```
void main(){  
    Pessoa p = new Pessoa();  
}
```

## ● Classes e Objetos

- Tendo instanciado uma pessoa, podemos alterar suas propriedades:

```
void main(){  
  
    Pessoa p = new Pessoa();  
    p.nome = "Vinicius";  
    p.idade = 20;  
    p.altura = 1.78;  
    p.cpf = 11111111111;  
    p.senha = "123";  
  
}
```

## ● Classes e Objetos

- Linguagens orientadas a objeto mais puristas, como Java e C++ preferem que **todos os atributos sejam privados**;
- Tendo em vista isso, surge o conceito de **getter**, o método oposto ao **setter**;
- Tal método serve apenas para **acessar o valor** de uma variável privada de **fora da classe**;
- Para declarar um **getter** em Dart basta usar a palavra **get**:

```
class Pessoa{  
  
    String _senha = "";  
  
    String get senha {  
        return _senha;  
    }  
  
}
```

## ● Classes e Objetos

- Para utilizá-lo, basta usá-lo como se fosse um atributo público:

```
void main(){  
    Pessoa p = new Pessoa();  
    p.nome = 'Vinicius';  
    p.idade = 20;  
    p.altura = 1.78;  
    p.cpf = 11111111111;  
    p.senha = '123';  
  
    print(p.senha);  
}
```

## ● Classes e Objetos

- Como você pode ter notado, os **getters** e **setters** são métodos da classe Pessoa;
- Entretanto é **possível declarar** qualquer **função** dentro de uma classe e ela **se tornará método** dela;
- Se o nome começar com **underline**, tal função **será privada**;

```
class Pessoa{  
    ...  
    bool ehDeMaior() {  
        return idade > 18;  
    }  
}
```

## ● Classes e Objetos

- Em classes existem alguns **métodos especiais**;
- Abordaremos alguns deles:
  - **Construtores**;
  - **toString()**;
- Quando utilizamos o operador **new** ou chamamos o método de mesmo nome da classe (começando com letra maiúscula), estamos chamando **o construtor**;
- A função principal do construtor é **alocar memória**, porém não entraremos em detalhes nessas questões;
- Para o curso é importante saber que **um construtor pode pedir parâmetros**, atuando como se fosse um **setter**;

## ● Classes e Objetos

- Por exemplo, ao definir uma classe Pessoa contendo apenas nome e idade, poderíamos pedir esses parâmetros no construtor:
- (Não se preocupe com o **this**, basicamente ele se refere ao objeto que está recebendo o valor, toda vez que se usar o **this**, significa que estamos usando um atributo ou método);

```
class Pessoa{  
  
    String nome;  
    int idade;  
  
    Pessoa(this.nome,this.idade);  
  
}
```



## ● Classes e Objetos

- Assim, ao **instanciar** uma pessoa, somos obrigados a **passar o nome e a idade**, e tais valores já serão **salvos nos atributos** da classe:

```
void main(){  
    Pessoa p = Pessoa('Vinicius',20);  
}
```

- Como o construtor é uma função, tudo que aprendemos sobre funções é válido, inclusive a questão de parâmetros opcionais, etc.

## ● Classes e Objetos

- Dart implementa uma funcionalidade chamada **construtores nomeados**;
- Que permite ao programador declarar **múltiplos construtores**;
- Imagine que queremos um construtor apenas para setar o nome da Pessoa, mas não queremos apagar o outro;
- Para isso, podemos **dar um nome** ao construtor:

```
class Pessoa{  
  
    String nome;  
    int idade;  
  
    Pessoa(this.nome,this.idade);  
  
    Pessoa.soNome(this.nome);  
  
}
```

## ● Classes e Objetos

- Se **nenhum** construtor **for declarado** explicitamente é como se existisse um **construtor vazio**.
- Também é possível criar um **construtor privado**, em alguns casos como veremos a frente, isso é bastante útil;
- Para declarar um **construtor privado**, basta fazer um **construtor nomeado** que comece com **underline**;

## ● Classes e Objetos

- O próximo método que veremos é o método **toString()**;
- Sua função é a de **exibir as informações** importantes de uma classe, **retornando uma String**;
- Basicamente **todos as classes já possuem** ele em Dart;
- Ele é **invocado automaticamente** quando um objeto é colocado dentro da função **print()**;
- Porém, a implementação “original de fábrica” trás apenas informações relevantes ao compilador;
- É necessário **sobrescreve-lo**;

## ● Classes e Objetos

- Veja uma implementação para a classe Pessoa:

```
class Pessoa{  
    String nome;  
    int idade;  
  
    @override  
    String toString(){  
        return 'Nome: $nome — Idade: $idade';  
    }  
}
```

## ● Classes e Objetos

- A **extensão** do Dart no **VSCode** possui um *snippet* para isso;
- Basta digitar **dentro da classe** “to” e será sugerido **toString()**;

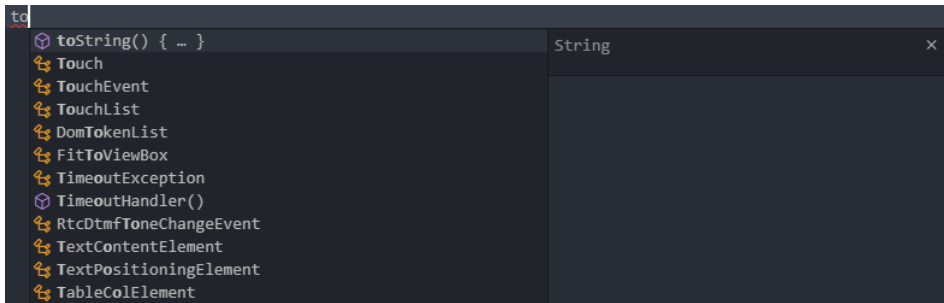


Figura 23: Caption

## ● Classes e Objetos

- Dart possui um conceito chamado **sobrecarga de operador**;
- Utilizando tal conceito é possível sobrescrever alguns operadores básicos da linguagem para classes:
  - `==`: verifica se dois objetos são iguais;
  - `>`: verifica se um objeto é maior que o outro;
  - `>=`: verifica se um objeto é maior ou igual a outro;
  - `<`: verifica se um objeto é menor que o outro;
  - `<=`: verifica se um objeto é menor ou igual a outro;
  - ...

## ● Classes e Objetos

- Para sobrecarregar o operador `==` é necessário implementar a função:

```
bool operator ==(Object other) {  
    // Implementacao da logica  
}
```

- Veja um exemplo para uma classe Pessoa:

```
class Pessoa {  
    int idade;  
    Pessoa(this.idade);  
  
    bool operator ==(Object other) {  
        if (other is Pessoa) {  
            return this.idade == other.idade;  
        }  
        return false;  
    }  
}
```



## ● Classes e Objetos

- Note que a função recebe um **Object** e retorna false caso o objeto **other** não seja Pessoa;
- É necessário tratar esse caso pois há vezes em que o programador pode acabar verificando se dois objetos de tipos diferentes são iguais;
- Por exemplo, ao verificar se uma String é igual a uma Pessoa.

## ● Classes e Objetos

- Para sobrecarregarmos os outros operadores listados anteriormente, vamos utilizar a palavra-reservada **covariant**;
- Tal palavra representa instancias de subclasses (veremos esse conceito em seguida);
- Veja a implementação do operador  $>$  para a classe Pessoa do slide anterior:

```
bool operator >(covariant Pessoa other) => this.idade > other.idade;
```

## ● Enumeráveis

- Dart implementa um **tipo especial de classe** chamado **enum**;
- A idéia de um **enum** é enumerar uma série de **valores**;
- Podem ser usados também dentro de **switch cases**;
- Veja um exemplo de declaração de um enumerável que representa os dias da semana:

```
enum DiasSemana {  
    Segunda,  
    Terca,  
    Quarta,  
    Quinta,  
    Sexta,  
    Sabado,  
    Domingo,  
}
```

## Enumeráveis

- Veja o exemplo de instanciação e uso no **switch case**:

```
main(List<String> args) {  
    DiaSemana dia = DiaSemana.Quarta;  
  
    switch (dia) {  
        case DiaSemana.Sabado:  
            print("Fim de semana!");  
            break;  
        case DiaSemana.Domingo:  
            print("Fim de semana!");  
            break;  
        default:  
            print("Dias uteis");  
            break;  
    }  
}
```

## Enumeráveis

- A partir de um enumerável é possível acessar um **atributo** especial **values**;
- Tal atributo acessa um **List<T>** contendo todos os valores que o enumerável pode **assumir**;
- Cada valor do enumerável também possui um atributo **index**, que possibilita saber qual a posição dele no atributo **values**:

```
DiaSemana quarta = DiaSemana.Quarta;  
  
DiaSemana todosDias = DiaSemana.values;  
  
int posicaoQuarta = quarta.index;
```

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

**Exercícios**

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

## Exercícios

1. Defina uma **classe Animal** contendo dois atributos de tipos diferentes, um deles deve ser privado, um método **construtor**, um método **get** e **set** e o método **toString()**. Instancie um **Animal**, defina seus atributos e exiba-o no terminal com a função **print()**.
2. Defina um **enumerável Regiao** que enumere os quatro pontos cardeais (norte, sul, leste e oeste). A partir desse enumerável, crie uma **classe Bairro** que contenha um atributo **nome** e uma **região**, além de um método **construtor** e um **toString()**.

3. A partir do item anterior, crie uma **classe Cidade** que contenha um atributo **nome** e uma **lista de bairros**. Crie um método na classe Cidade chamado **filtrarBairros()**. Tal método deve receber um objeto do enumerável **Regiao** e retornar uma lista de bairros da cidade que são dessa região. Instancie uma cidade e alguns bairros e adicione-os a cidade, em seguida busque por uma região utilizando a função criada.



## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

**Herança, Classes Abstratas e Mixins**

Exercícios

Sincronismo e Assincronismo

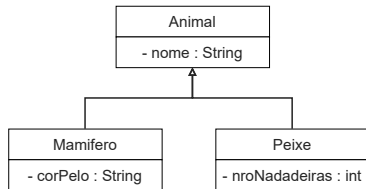
Exceções

Exercícios

Singleton

## Herança

- Agora que vimos o básico de classes e objetos, veremos conceitos um pouco mais avançados de orientação a objetos;
- Conceitos esses que usaremos bastante no **Flutter**;
- Veremos agora o conceito de **herança**;



**Figura 24:** Diagrama representando uma hierarquia de classes.

## ● Herança

- Basicamente o conceito permite criar uma classe com as **mesmas características** de outra, possuindo métodos ou atributos que a **outra classe não possuía**;
- Para isso, diz-se que uma classe **estende** a outra;
- A classe que é **estendida** é chamada de **superclasse**;
- E a classe que a **estendeu** é chamada de classe **filha**;
- Em Dart existe uma palavra-reservada para estender outra classe: **extends**;

## ● Herança

- Vamos utilizar a classe Pessoa que criamos anteriormente e estendê-la, criando uma **PessoaNorteAmericana**;
- A única diferença entre uma Pessoa e uma PessoaNorteAmericana é o **cálculo da maioridade**;
- Vamos assumir que pessoas que moram na América do Norte se tornam de maiores apenas aos 21 anos;
- Seguindo a nossa implementação da classe Pessoa:

```
class Pessoa{  
    String nome = "";  
    int idade = 0;  
  
    bool ehDeMaior() {  
        return idade > 18;  
    }  
}
```

## ● Herança

- Criaremos agora a classe PessoaNorteAmericana, estendendo a classe Pessoa:
- A partir do conceito de herança, todo objeto da classe PessoaNorteAmericana possui nome e idade;

```
class PessoaNorteAmericana extends Pessoa{  
  
}
```

- Vamos adicionar um atributo SSN que representa um documento presente nos EUA;

```
class PessoaNorteAmericana extends Pessoa{  
    String SSN = "";  
}
```

## ● Herança

- Agora todo objeto da classe PessoaNorteAmericana possui nome, idade e SSN;
- Precisamos agora sobrescrever o método **ehDeMaior()**, fazendo com que retorne a verificação de se a idade é a maior que 21:

```
class PessoaNorteAmericana extends Pessoa{  
    String SSN = "";  
  
    @override  
    bool ehDeMaior() {  
        return idade > 21;  
    }  
}
```

- Note que agora podemos instanciar PessoaNorteAmericana e alterar seus atributos:

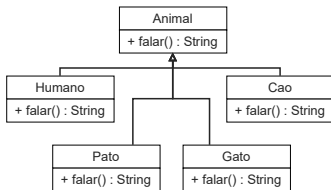
```
void main(){  
  PessoaNorteAmericana p = PessoaNorteAmericana();  
  p.nome = 'Sophia';  
  p.idade = 22;  
  p.SSN = '309-77-XXXX';  
}
```

- Veremos no Flutter que todas as classes estendem de uma única só;
- E a partir disso possuem os **mesmos métodos**, só que com **implementações diferentes**;

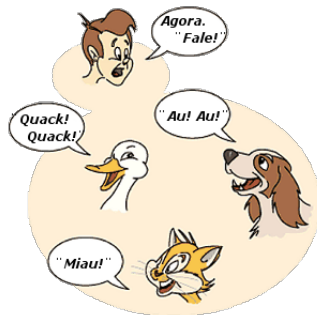


## Herança

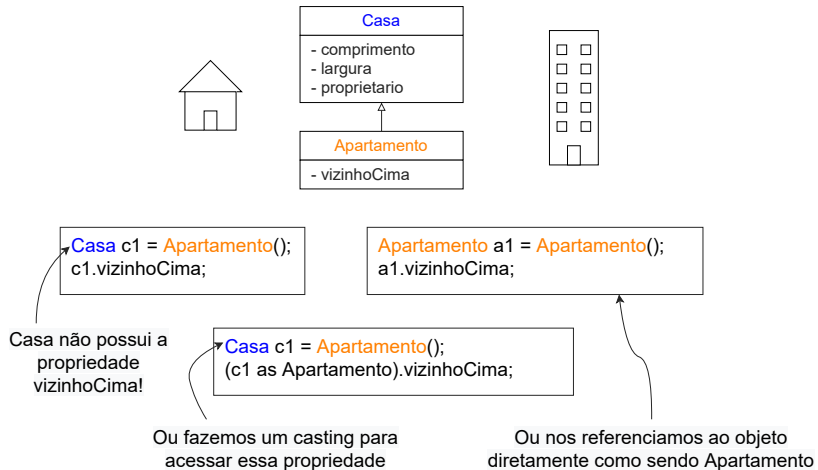
- Outro conceito muito importante que vamos utilizar é o de **polimorfismo**;
- A propriedade que um objeto tem de assumir várias formas;
- Cada forma com suas próprias características;



**Figura 25:** Exemplo de polimorfismo.



**Figura 26:** Polimorfismo. Fonte: (TECHBLOG, 2011).



**Figura 27:** Exemplo de polimorfismo ao manipular variáveis e usar métodos e propriedades.

## ● Herança

- Resumidamente pode-se dizer que como a classe PessoaNorteAmericana **estende** Pessoa, ela pode ser salva em uma **variável do tipo Pessoa**:
- Cuidado pois o contrário não é válido!

```
void main(){  
    PessoaNorteAmericana p1 = PessoaNorteAmericana();  
    Pessoa p2 = PessoaNorteAmericana();  
}
```

- Também é interessante comentar que os métodos de PessoaNorteAmericana não poderão ser usados enquanto a variável p2 for de Pessoa, será necessário fazer um **casting**.

## ● Herança

- Para poder fazer um casting é necessário usar o operador **as**;
- Como dito, no **Flutter** utilizaremos polimorfismo o tempo todo, então é necessário entender como as variáveis conseguem armazenar suas sub-classes;

## ● Herança

- É interessante salientar que não é possível estender duas classes no Dart;
- Tal conceito é chamado de **herança-múltipla**;
- Veremos estratégias para isso a seguir;

## ● Classes Abstratas

- O próximo conceito que veremos é o de **classes abstratas**;
- Sua definição é simples: classes que **não podem ser instanciadas**, apenas estendidas;
- **Servem de base** para outras classes;
- Para uma classe ser abstrata em Dart precisamos usar a palavra-reservada **abstract**;
- Definiremos uma classe abstrata Veiculo para nossos exemplos:

```
abstract class Veiculo{  
  
}
```

## ● Classes Abstratas

- Classes abstratas podem ter tudo que uma classe normal, exceto serem instanciadas;
- Além de poderem ter **métodos abstratos**;
- Tais métodos **devem ser implementados** pelas classes que a estendem;
- Para definir um **método abstrato** basta apenas declarar sua assinatura, **sem abrir e fechar chaves**;
- Para exemplificar isso, vamos definir um atributo **velocidade** e um método abstrato **acelerar()**:

```
abstract class Veiculo{  
    double velocidade;  
    void acelerar();  
}
```

## ● Classes Abstratas

- Criaremos agora duas classes que estenderão a classe Veiculo, as classes Carro e Moto;
- Cada uma delas irá implementar o método **acelerar()** de uma forma:
  - Carros aceleram de 10 em 10 metros;
  - Motos de 15 em 15;

```
class Carro extends Veiculo{  
    @override  
    void acelerar() => this.velocidade += 10;  
}
```

```
class Carro extends Moto{  
    @override  
    void acelerar() => this.velocidade += 15;  
}
```



## ● Classes Abstratas

- Se aproveitando do **polimorfismo** novamente podemos criar variáveis do tipo Veiculo;
- Entretanto agora não será necessário fazer um **casting**, visto que a classe é abstrata e as implementações estão na classe que a estende:

```
Veiculo carro = Carro();  
Veiculo moto = Moto();  
  
carro.acelerar();  
moto.acelerar();
```

## ● Classes Abstratas

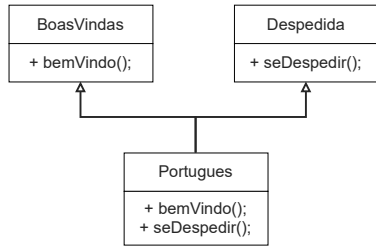
- No Flutter não criaremos novas classes abstratas (não com muita frequência), porém veremos muitas classes, inclusive de pacotes de terceiros que utilizam classes abstratas;
- Um dos **conceitos mais importantes** de se absorver e que utilizaremos bastante é o de **polimorfismo**, porque a toda hora vamos nos **referenciar a objetos de classes pai**;

## Mixins

- Para finalizar o conceito de **polimorfismo**, veremos um tipo especial de classe abstrata: os **mixins**;
- Inspirados nas interfaces do Java, mixins são **classes abstratas sem métodos**;
- É comum dizer que os mixins definem **comportamentos**;

## ● Mixins

- Mixins também permitem **herança-múltipla**;
- Para definir um mixin, basta usar a palavra-reservada **mixin**;
- Para implementar um mixin é necessário utilizar a palavra-reserva **with**;
- Geralmente se diz que mixins são “**contratos**” que a classe que a implementa é obrigada a fazer;



**Figura 28:** Herança múltipla usando mixins.

## Mixins

- Veremos um exemplo de um mixin BoasVindas e outro Despedida;
- Nele definiremos alguns métodos para se comunicar com o usuário:

```
mixin BoasVindas{  
    void bemVindo();  
}
```

```
mixin Despedida{  
    void seDespedir();  
}
```

## ● Mixins

- Criaremos agora as classes Portugues e Ingles, onde iremos implementar esses métodos;
- Para a classe Portugues, precisaremos imprimir as frases “Olá, seja bem-vindo!” e “Tchau, até mais!”;

```
class Portugues with BoasVindas,Despedida{  
    void bemVindo() => print('Ola, seja bem—vindo!');  
    void seDespedir() => print('Tchau, ate mais!');  
}
```

- Já para o Inglês, precisaremos imprimir as frases “Hi, welcome!” e “Bye, see you later!”;

```
class Ingles with BoasVindas,Despedida{  
    void bemVindo() => print('Hi, welcome!');  
    void seDespedir() => print('Bye, see you later!');  
}
```

## ● Mixins

- Um dos benefícios do polimorfismo é que podemos trabalhar agora usando os **dois mixins**;
- Para isso, vamos usar uma List de BoasVindas:
- Como sabemos, a priori que ambas as classes da lista também são instancias de Despedida, podemos iterar sobre essa lista chamando os dois métodos;
- Apenas precisaremos fazer um **casting**:

```
List<BoasVindas> boas = [  
    Portugues(),  
    Ingles()  
];  
  
boas.forEach((element) {  
    element.bemVindo();  
    (element as Despedida).seDespedir();  
});
```



## Mixins

- Mixins também é outro conceito que **não utilizaremos muito** no Flutter;
- Apenas iremos usá-los, porém, é necessário **entender** como funcionam;

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

**Exercícios**

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

## Exercícios

1. Crie uma classe `Quadrado` que possui um atributo `l` correspondente ao lado de um quadrado. Além de dois métodos **`calcularArea()`** e **`calcularPerimetro()`** que devem retornar o valor do lado ao quadrado e o valor do lado multiplicado por 4, respectivamente;
2. Instancie um objeto da classe `Quadrado`, defina algum valor para a medida de seu lado e calcule sua área e seu perímetro;
3. Crie uma classe `Retângulo` que estenda a classe `Quadrado` e possua um atributo `l2`. Sobreescreva o método **`calcularArea()`** para calcular de acordo com as fórmulas de um retângulo.

## ● Exercícios

4. Crie uma classe abstrata `Animal` que possui apenas um nome e cor e um método abstrato **`emitirSom()`**. Crie duas classes a sua escolha que estendam a classe `Animal`, implementando o método **`emitirSom()`** exibindo uma `String` no console. As classes também devem possuir algum atributo diferente de `Animal`;
5. Crie uma função **`main()`** e instancie um objeto de cada uma das classes criadas no item 1, adicionando-as a uma lista e iterando sobre essa lista invocando o método **`emitirSom()`** de cada objeto.
6. Crie um mixin `Aceleravel` que contenha um método **`acelerar()`**. Crie duas classes `Carro` e `Foguete` que implementem esse mixin. A classe `Carro` deve possuir um atributo `distancia`. Sua implementação do método **`acelerar()`** deve somar algum valor à distância. Já a classe `Foguete` deve possuir um atributo `altura` e sua implementação do método **`acelerar()`** deve somar algum valor à altura.

## ● Exercícios

7. Crie uma classe `Aluno` que contenha um nome, idade e uma lista de notas, além de um método **`calcularMedia()`** que retorna a média das notas. Crie uma outra classe `Turma` que contenha uma lista de alunos privada e um método **`adicionarAluno()`** que recebe um aluno e adiciona-o a lista. A classe `Turma` também deve possuir um método **`filtrarAprovados()`** que retorna uma lista de alunos que possuem a média maior ou igual a 7. Instancie um turma e dez alunos (procure como gerar números aleatórios se desejar) e filtre os aprovados da turma.
8. Procure ou implemente uma função em Dart para ordenar a lista de alunos pelo nome ou média antes de retornar os aprovados no método **`filtrarAprovados()`**;

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

**Sincronismo e Assincronismo**

Exceções

Exercícios

Singleton

## ● Sincronismo e Assincronismo

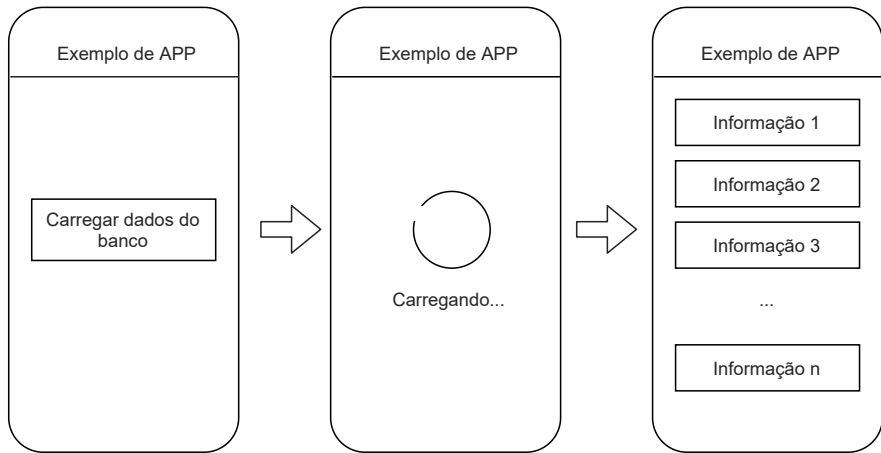
- Agora que vimos os principais tópicos de orientação a objetos implementados em Dart, vamos ver um pouco de **sincronismo e assincronismo**;
- Em Dart é muito comum acontecerem operações **fora da execução estruturada**;
- Muitas vezes é comum que uma operação leve mais tempo que o normal;
- Por exemplo, uma busca em um banco de dados;
- Não sabemos quanto tempo irá levar para ela ser executada;
- E **não podemos deixar o programa esperando** ela acontecer, existem outros processos que precisam ser realizados nesse meio tempo;
- Para isso, Dart trás implementado o conceito de **assincronismo**;

## ● Sincronismo e Assincronismo

- Um exemplo disso, seria exibir uma barra de carregamento (animada) durante uma busca no banco de dados;
- Tal exemplo seria impossível sem o uso de assincronismo;
- Pois duas operações são necessárias ao mesmo tempo:
  - Exibir a barra de carregamento animada;
  - Realizar a busca no banco de dados;

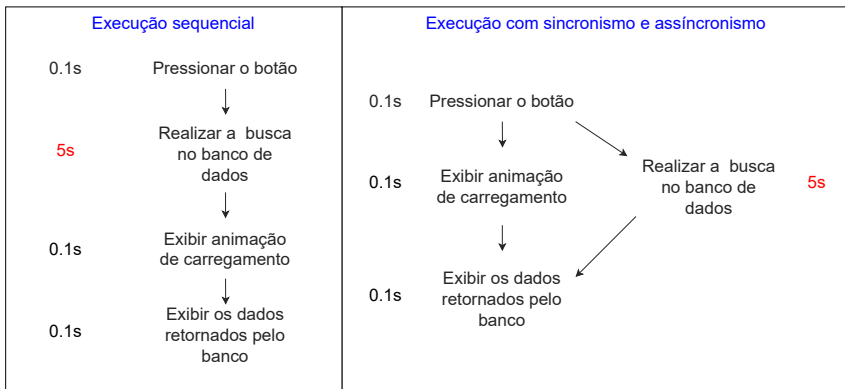


## ● Sincronismo e Assincronismo



**Figura 29:** Exemplo de carregamento com animação.

## ● Sincronismo e Assincronismo



**Figura 30:** Comparação entre processos sequenciais e síncronos/assíncronos.

## ● Sincronismo e Assincronismo

- Em Dart **métodos e funções** podem ser **assíncronas**;
- Para tal, existe uma palavra-reservada **async**;
- Toda vez que um método é **async**, seu retorno será um dado **futuro**;
- Pra isso Dart introduz um tipo de dado **Future<T>**;
- Como Dart possui tipagem opcional, um método não necessariamente precisa ser dito **async**, mas se for, seu tipo de retorno será **Future<T>**;
- A sintaxe é simples: após os parâmetros e antes das chaves, é necessário colocar a palavra **async**:

```
metodoAssincrono() async {  
  
}
```

## ● Sincronismo e Assincronismo

- Logo, como nosso método do exemplo não retornou nada, ele seria `void`;
- Porém, como é **async**, ele será `Future<void>`;
- Agora precisamos falar um pouco da classe **Future**;
- A classe possui algumas propriedades e métodos interessantes;
- Começaremos pelos seus **construtores**;

```
Future<void> metodoAssincrono() async {  
}
```

## ● Sincronismo e Assincronismo

- Citaremos os dois que mais utilizaremos em nosso curso:
  - **Future(FutureOr<T> computation())**: recebe uma “função” a ser executada paralelamente pelo dispositivo;
  - **Future.delayed(Duration duration, [FutureOr<T> computation()])**: recebe uma duração (um intervalo de tempo) e opcionalmente uma “função” a ser executada. Tal construtor irá aguardar o intervalo de tempo e se houver uma computação passada como parâmetro irá executá-la;
- Vamos ver um exemplo usando o **Future.delayed()** em seguida;
- Alguns parâmetros do construtor da classe **Duration**:
  - **int days**: número de dias;
  - **int minutes**: número de minutos;
  - **int seconds**: número de segundos;
  - **int milliseconds**: número de milissegundos ( $10^{-3}$ );
  - **int microseconds**: número de microssegundos ( $10^{-6}$ );

## ● Sincronismo e Assincronismo

- Antes precisamos ver alguns outros métodos da classe **Future**:
  - **then(FutureOr<T> onValue(T valor))**: recebe uma função de T que irá “interagir” com o resultado futuro;
  - **whenComplete(FutureOr<void> action())**: recebe uma função void a ser executada quando o processo finalizar;
- Veremos alguns exemplos agora;

```
then ( funcaoT );
```

```
void funcaoT (T resultado){  
    comandos...  
}
```

```
whenComplete ( funcaoVoid );
```

```
void funcaoVoid(){  
    comandos...  
}
```

**Figura 31:** Comparação entre os métodos **then()** e **whenComplete()**.

## ● Sincronismo e Assincronismo

- Vamos criar uma função que aguarda 3 segundos e após exibe uma mensagem no console;
- Utilizaremos o **Future.delayed()** e **whenComplete()**:
- A função basicamente irá aguardar três segundos e exibir uma mensagem no console;
- Não esqueça do **return**!

```
Future<void> espera3Segs() async {  
  return Future.delayed(Duration(seconds: 3)).  
    whenComplete(() {  
      print('A');  
    });  
}
```

## ● Sincronismo e Assincronismo

- Vamos executá-la em nossa **main()**;
- Para isso, vamos primeiro executá-la e depois exibir no console uma mensagem:
- Teoricamente o resultado esperado ao executar (sequencialmente) era:
- Porém, Dart executará as duas em paralelo, visto que a função é assíncrona, logo o resultado é o contrário:

```
main(List<String> args) {  
  espera3Segs();  
  print('B');  
}
```

A  
B

B  
A



## ● Sincronismo e Assincronismo

- É possível enxergar o que ocorre em função do tempo:

	Fluxo normal de execução	Fluxo 2
t = 0s	chama a função espera3Segs(); print( ' B ' );	função espera3Segs() é executada AGUARDANDO... (faltam 3s)
t = 1s		AGUARDANDO... (faltam 2s)
t = 2s		AGUARDANDO... (faltam 1s)
t = 3s		print( ' A ' ); voltando ao fluxo normal

**Figura 32:** Execução em função do tempo.

## ● Sincronismo e Assincronismo

- Mas e se quiséssemos forçar a execução do **espera3Segs()** antes do print?
- Existem duas formas de se fazer isso;
  - Usando o **whenComplete()** na chamada da função;
  - Forçar a espera usando o **await**;
- Ambas soluções são válidas, porém, o uso do **await** obriga a função a ser **async**;
- Sempre que houver **await** a função é **async**;
- No nosso caso, ambas as soluções são possível, mas existem lugares que se espera uma função síncrona, logo, apenas a solução do **whenComplete()** será válida;

## ● Sincronismo e Assincronismo

- Usando o **whenComplete()**:
- Notem que a nossa **main()** continuou síncrona;

```
main(List<String> args) {  
    espera3Segs().whenComplete(() {  
        print('B');  
    });  
}
```

## ● Sincronismo e Assincronismo

- Usando o **await**:
- Notem que a nossa **main()** se tornou assíncrona com o uso do **async**;

```
main(List<String> args) async {  
  await espera3Segs();  
  print('B');  
}
```

## ● Sincronismo e Assincronismo

- Ambos os casos nos trazem o resultado que queríamos:

A  
B

## ● Sincronismo e Assincronismo

- Por fim, veremos o uso do **then()**;
- Tal método permite que **interajamos com um resultado futuro**;
- Muito comum quando estamos fazendo **buscas em bancos de dados** ou **aguardando inputs** do usuário (o usuário pode *literalmente* ficar horas esperando para apertar um botão);
- Tendo em vista isso, vamos criar um método que aguarda 2 segundos e nos retorna uma lista contendo cinco inteiros:

```
Future<List<int>>> retornaLista() async {  
    await Future.delayed(Duration(seconds: 2));  
    return [1, 2, 3, 4, 5];  
}
```

## ● Sincronismo e Assincronismo

- A partir disso, vamos executar o método na **main()** e imprimir apenas os números pares dessa lista;
- Usaremos o método **then()**, como sabemos a priori que o método **retornaLista()** nos retorna um **List<int>**, precisamos de um método que recebe isso como parâmetro;
- Vamos primeiro declarar ele dentro do **then()** assim como fazemos com o **forEach()**:

```
main(List<String> args) {  
    retornaLista().then((value) {  
        for (int i = 0; i < value.length; i++) {  
            if (value[i].isOdd) {  
                print(value[i]);  
            }  
        }  
    });  
}
```

## ● Sincronismo e Assincronismo

- Note que a variável **value** é o nosso resultado futuro, uma lista de inteiros;
- Como é uma função que recebe isso como parâmetro que precisamos passar para o **then()**, podemos organizar nosso código melhor e passar essa função como parâmetro:

```
main(List<String> args) {  
    retornaLista().then(imprimePares);  
}  
  
void imprimePares(List<int> valores) {  
    for (int i = 0; i < valores.length; i++) {  
        if (valores[i].isOdd) {  
            print(valores[i]);  
        }  
    }  
}
```



## ● Sincronismo e Assincronismo

- **Utilizaremos muito** nesse curso **métodos assíncronos**;
- Algumas funções do Flutter as utilizam, seu uso é muito comum também em **pacotes da comunidade**;
- Por isso é muito importante pegar bem esses conceitos, pois as vezes é muito fácil se perder ao tentar “sincronizar” seu código;
- **Erros de sincronia são difíceis de se perceber** a e o compilador não irá detectá-los visto que **são erros de lógica**.

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

**Exceções**

Exercícios

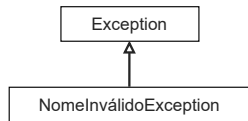
Singleton

## Exceções

- Durante a execução de um código, **erros** podem vir a acontecer;
- Por exemplo, **acessar uma posição que não existe** de uma lista;
- Não queremos que esses erros **parem a execução da aplicação**;
- Para isso, **iremos tratar erros e exceções** que podem vir a acontecer durante a execução;
- Também veremos como **criar nossas próprias** classes que representam erros;

## ● Exceções

- Dart provê duas classes para representar erros e exceções;
- Todas as outras estendem/implementam tais classes;
- São elas:
  - **Error;**
  - **Exception;**
- Utilizaremos **preferencialmente a classe Exception;**
- Através de herança, vamos estender a classe Exception para criar nossos erros;



**Figura 33:** Exemplo de classe que estende a classe Exception.

## ● Exceções

- Veremos quatro cláusulas/comandos:
  - **try**: tenta executar uma função que pode vir a gerar um erro, caso aconteça um erro, executará os comandos do catch;
  - **catch**: série de comandos a serem executados para um erro capturado;
  - **throw**: lança um erro que para a execução do código se não for capturada por uma cláusula **catch**;
- Existe também uma cláusula **finally**, porém não utilizaremos ela muito;
- Basicamente tal cláusula apenas executa independente do **catch** ser executado ou não;

```
try {  
    comandos que lançam erros  
} on ClasseFilhaException catch ( objeto ) {  
    faz algo com objeto  
} on Exception catch ( objeto2 ) {  
    faz algo com objeto2  
} on Error catch ( objeto3 ) {  
    faz algo com objeto3  
} finally {  
    comandos...  
}  
  
funcaoLancaErro() {  
    throw new Exception("mensagem");  
}
```

Figura 34: Cláusulas/Comandos.

## ● Exceções

- Começaremos vendo um exemplo de um erro, depois veremos como tratá-los:

```
main(List<String> args) {  
  List<int> valores = [];  
  print(valores[3]);  
}
```

- Ao executarmos o código, temos o seguinte resultado:

```
Unhandled exception:  
RangeError (index): Invalid value: Valid value range is empty: 3  
#0    List.[] (dart:core-patch/growable_array.dart:254:60)  
  
#1    main  
main.dart:10  
#2    _delayEntrypointInvocation.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:281:32)  
#3    _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:184:12)  
  
Exited (255)
```

## ● Exceções

- Note que a execução do código foi encerrada (Exited 255);
- Para contornar isso, colocaremos o trecho de código que causou a exceção na **cláusula try**:

```
List<int> valores = [];  
  
try{  
    print(valores[3]);  
}
```

## ● Exceções

- Para capturar uma exceção/erro, vamos utilizar o comando **catch**, dando um nome para o objeto;
- Precisamos determinar o tipo de exceção ou erro que vamos capturar, em primeiro momento vamos utilizar a classe Error, visto que ao executar o código tivemos um RangeError;
- A sintaxe do comando é **on TIPO catch (nomeObjeto)**:

```
List<int> valores = [];  
  
try{  
    print(valores[3]);  
}on Error catch(e){  
    print(e);  
}
```



## ● Exceções

- Dessa forma, a hora que o erro ocorrer, **a execução do código não será finalizada** imediatamente, logo, seu houvesse um terceiro comando, ele seria executado:

```
List<int> valores = [];  
  
try{  
    print(valores[3]);  
}on Error catch(e){  
    print(e);  
}  
print('comando executado ao final');
```

## ● Exceções

- Vamos agora **criar uma classe para representar um erro** em nosso programa;
- Imagine que temos uma função que **soma os inteiros** de uma lista, **apenas se forem positivos**;
- Se a **soma passar de 1000**, também **será considerado um erro** do programa;
- Criaremos uma classe **ValorNegativoException** para usarmos quando um valor negativo for achado na lista;
- Para isso vamos implementar a classe Exception, usando a palavra **implements**:

```
class ValorNegativoException implements Exception {  
  
}
```

## ● Exceções

- A principio vamos apenas sobrescrever o método **toString()**, retornando uma mensagem que indique o erro:
- Agora, criaremos a função que irá “lançar” essa exceção;
- Para lançar exceções, usaremos a palavra **throw**;

```
class ValorNegativoException implements  
    Exception {  
    @Override  
    String toString() {  
        return 'Valor negativo nao sera somado!';  
    }  
}
```

- Basicamente a nossa função **soma()** irá iterar sobre a lista e lançar uma exceção se encontrar um valor negativo:

```
int soma(List<int> valores) {  
    int soma = 0;  
    valores.forEach((element) {  
        if (element.isNegative){  
            throw ValorNegativoException();  
        }  
        soma += element;  
    });  
    return soma;  
}
```

## ● Exceções

- Se o valor da soma chegar a ser maior que 1000, vamos lançar uma Exception contendo uma mensagem informando que o valor estourou:

```
int soma(List<int> valores) {  
    int soma = 0;  
    valores.forEach((element) {  
        if (element.isNegative){  
            throw ValorNegativoException();  
        }  
        soma += element;  
        if(soma>1000){  
            throw Exception('Valor estourou');  
        }  
    });  
    return soma;  
}
```

## ● Exceções

- Agora para tratar a exceção ao ser chamado o método, vamos utilizar a classe `ValorNegativoException`:

```
main(List<String> args) {  
    List<int> valores = [1, 2, 3, 4, 5];  
    try {  
        print(soma(valores));  
    } on ValorNegativoException catch (e) {  
        print(e);  
    }  
}
```

## ● Exceções

- Em seguida, iremos tratar a Exception abaixo da ValorNegativoException:
- Note que se a Exception fosse tratada antes, o trecho de código da ValorNegativoException não seria executado, visto que através do polimorfismo toda exceção será capturada ao capturar Exception;

```
main(List<String> args) {  
    List<int> valores = [1, 2, 3, 4, 5];  
    try {  
        print(soma(valores));  
    } on ValorNegativoException catch (e) {  
        print(e);  
    } on Exception catch (e) {  
        print(e);  
    }  
}
```

- Por fim, para finalizar, é possível também tratar exceções ao utilizar **métodos assíncronos**;
- Para isso, são definidos dois métodos na **classe Future**:
  - **Future<void> catchError(Function onError)**: função de tratamento de erro semelhante a utilizar o bloco **catch**, basta passar uma função que recebe uma exceção como parâmetro;
  - **Future<void> onError(FutureOr<void> Function(E, StackTrace))**: função a ser executada semelhante ao bloco **catch**, porém a função recebida como parâmetro recebe além da exceção um objeto StackTrace onde é possível ver todas as informações relacionadas ao erro;



- Veja um **exemplo de uso** para os dois métodos usando a função **espera3Segs()** apresentado nos slides anteriores:

```
espera3Segs().catchError((e) {  
    print(e);  
});
```

```
espera3Segs().onError((error, stackTrace) {  
    print(error);  
});
```

## Exceções

- Nem sempre iremos exibir os erros no console usando o **print()**;
- Nas aulas seguintes veremos meios de “apresentar” o erro para o usuário sem utilizar o **print()**;

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

**Exercícios**

Singleton

## Exercícios

1. Crie uma função que dados dois inteiro a e b, retorne o maior deles. Se os valores forem iguais a função deve lançar uma exceção contendo uma mensagem apropriada. Utilize o bloco try/catch em uma função main para testar a execução do método.
2. Crie uma classe ValorInvalidoException que implementa a classe Exception e altere a função do item anterior para que ela lance uma exceção dessa classe caso o valor de n seja negativo ou maior que 20.

3. Crie uma função assíncrona que receba um inteiro  $n$  e retorne uma lista contendo os  $n$  primeiros números pares;
4. Crie uma função assíncrona que receba dois inteiros  $n$  e  $m$  e chame a função do item anterior passando  $n$  como parâmetro e a partir do resultado, subtraia de cada valor da lista o valor de  $m$  e exiba no console o resultado.

## Exercícios

5. A partir da função acima, crie uma função que aguarde entre 1 e 5 segundos e retorne um inteiro aleatório entre 0 e 100. Crie uma função **main()** e gere dois valores aleatórios, somando-os ao final. A função **main()** deve ser síncrona!

## ● Seções

Introdução

Tipos de Dados

Estruturas da Linguagem

Null-Safety

Exercícios

Classes, Objetos e Enumeráveis

Exercícios

Herança, Classes Abstratas e Mixins

Exercícios

Sincronismo e Assincronismo

Exceções

Exercícios

Singleton

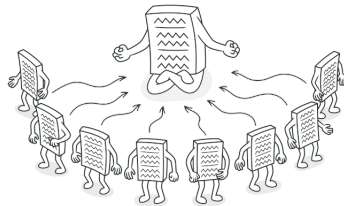
## ● Singleton

- Agora que vimos todos os conceitos de Dart (ou pelo menos os principais), veremos um **padrão de projetos** que pode ajudar muito na hora que estamos no Flutter;
- Como não vimos exemplos complexos de classes ainda, iremos apenas apresentar o padrão, veremos seu uso prático na hora do Flutter;
- Os padrões de projetos surgiram para **solucionar problemas comuns** na hora do desenvolvimento de software;



## ● Singleton

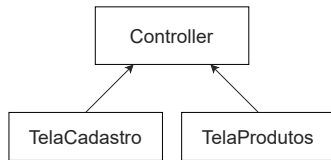
- Veremos o **Singleton**, cuja ideia é tornar uma classe global e única em todo o código;
- Logo, não será possível construir novos objetos dessa classe;
- Isso é bem útil quando estamos lidando com bancos de dados ou sensores;
- Ao invés de criarmos um novo objeto toda hora, vamos sempre ter um e utilizar esse;



**Figura 35:** Singleton. Fonte: singleton.

## ● Singleton

- Muitas vezes precisamos manipular o mesmo objeto;
- Principalmente no Flutter, quando entrarmos na ideia de Controller;
- Porém, nem sempre vamos conseguir **passar como referência** o objeto que precisamos em duas classes;



```
Controller c = Controller();
```

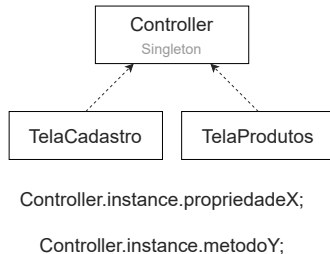
```
TelaCadastro cadastro = TelaCadastro();  
cadastro.controller = c;
```

```
TelaProdutos produtos = TelaProdutos();  
produtos.controller = c;
```

**Figura 36:** Motivação do uso do Singleton.

## ● Singleton

- Com o Singleton, podemos tornar uma classe “**global**” e acessá-la em qualquer ponto da aplicação;



**Figura 37:** Motivação do uso do Singleton.

## ● Singleton

- A idéia central do Singleton é a **instancia única** do objeto;
- Para implementar o Singleton é necessário que a classe tenha:
  - Um atributo **static** do mesmo tipo da classe;
  - Um **construtor privado** e nenhum construtor público;
  - Um método **static** para obter a instancia única e caso ela não tenha sido criada ainda, criá-la;

# ● Singleton

- Singleton:

```
class Singleton {  
    static Singleton? _instance;  
  
    Singleton.__privado();  
  
    factory Singleton() {  
        return _instance ??= Singleton.__privado();  
    }  
}
```

## Singleton


- Basicamente, basta copiar, colar e alterar o nome da sua classe para usar;

## Singleton


- Para utilizar o Singleton, basta invocar o construtor como se estivesse criando um novo objeto normalmente:

```
Singleton x = Singleton();  
Singleton y = Singleton();
```

## ● Referencias

 CHARANRAJ. **Beginner's guide - Object Oriented Programming**. 2018. Disponível em: <<https://dev.to/charanrajgolla/beginners-guide---object-oriented-programming>>.

 DART. **Dart Docs**. 2021. Disponível em: <<https://dart.dev/guides>>.

 TECHBLOG. **Polimorfismo**. 2011. Disponível em: <<http://techblog.desenvolvedores.net/tag/objetos/>>.



Obrigado :)  
Vinicius Takeo Friedrich Kuwaki  
vtkwki@gmail.com  
github.com/takeofriedrich

**NEMOBIS**

