PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS

PROF° DOUGLAS ROBERTO ROSA PEREIRA

OPERADOR *NON-NULL ASSERTION*

- Em alguns exercícios da aula passada foi necessário utilizar um ponto de exclamação ao final de alguns método de inserção de dados como aqui:

```
int duracaoEvento = int.parse(stdin.readLineSync()!);
```

- As exclamações ao final de uma variável em Dart representam o operador "*non-null assertion operator*"
- Isso significa que o desenvolvedor está garantindo que a variável não é nula, mesmo que o compilador não possa garantir isso.
- Esse operador só deve ser utilizado onde não existe a possibilidade real de que aquele valor possa ser nulo por mais que tudo indique o contrário.

LEITURA DA DOCUMENTAÇÃO

- Uma boa prática é sempre que surgirem dúvidas de como funciona alguma coisa na linguagem de programação que você está trabalhando, o programador consulte a documentação oficial da linguagem.
- No caso do Dart a documentação pode ser encontrada em: https://api.dart.dev

COLEÇÕES

- Coleções são grupos de objetos que representam um elemento específico. A biblioteca dart::collection é usada para implementar a coleção no Dart. Há uma variedade de coleções disponíveis em Dart.

LISTAS

- Uma lista é um grupo ordenado de objetos. Ela pode conter vários objetos do mesmo tipo como pode ter vários objetos de tipos diferentes:

```
1 void main() {
2   List nomes = ["Douglas", "Tião", "Pandora"];
3   List coisas = ["Banana", 24, nomes, 3.14];
4 }
5
```

- No caso da nossa segunda lista ela é uma lista do tipo *dynamic* pois contém vários tipos diferentes. Se você fizer como no exemplo e omitir o tipo ele assume que é *dynamic*.

LISTAS

- Agora definimos as mesmas listas com o seu tipo explícito:

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "Pandora"];
  List<dynamic> coisas = ["Banana", 24, nomes, 3.14];
}

```

LISTAS

- Pode-se verificar o tamanho de uma lista usando a função ***length***.

```
1  void main() {
2    List<String> nomes = ["Douglas", "Tião", "Pandora"];
3    List<dynamic> coisas = ["Banana", 24, nomes, 3.14];
4    print(nomes.length);
5    print(coisas.length);
6  }
7
```

Console

```
3
4
```

- As funções ***first*** e ***last*** retornam respectivamente o primeiro e o último elemento:

```
1  void main() {
2    List<String> nomes = ["Douglas", "Tião", "Pandora"];
3    List<dynamic> coisas = ["Banana", 24, nomes, 3.14];
4    print(nomes.first);
5    print(coisas.last);
6  }
7
```

Console

```
Douglas
3.14
```

LISTAS

- Os índices das listas em dart começam em 0. Com isso em mente podemos pegar um determinado item na lista podemos referencia-los por seu índice:

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "Pandora"];
  List<dynamic> coisas = ["Banana", 24, nomes, 3.14];
  print(nomes[0]);
  print(coisas[2]);
}

```

Console

```
Douglas
[Douglas, Tião, Pandora]
```

FUNÇÕES COM LISTAS

- **Sintaxe:** nome_lista.nome_função

✓ **add(item):** adiciona um item no fim da lista;

✓ **addAll(lista):** adiciona um conjunto itens de outra lista no final de sua lista. Pode-se também passar vários valores individuais no formato de lista. Ex: lista.addAll([1,2,3])

✓ **insert(posição, item):** insere um item na lista na posição especificada.

✓ **contains(item):** retorna *true* se o item existe na lista ou *false* se o item não existe na lista.

✓ **indexOf(item):** retorna o índice do item fornecido se ele existir, senão retorna -1.

FUNÇÕES COM LISTAS

- ✓ **remove(item):** remove o item da lista especificado. Se a remoção possível retorna *true*, se impossível retorna *false*.
- ✓ **removeAt(índice):** remove o item da lista com o índice especificado. Se a remoção for possível retorna *true*, se impossível retorna *false*.
- ✓ **shuffle():** embaralha o conteúdo da lista.
- ✓ **clear():** limpa a lista (apaga todos os itens).

PERCORRENDO LISTAS

- 1° forma e menos eficiente: utilizando um **for** padrão.

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "José", "Maria", "Julia"];

  for (int i = 0; i < nomes.length; i++) {
    nomes[i] = nomes[i].toUpperCase();
    print(nomes[i]);
  }
}

```

PERCORRENDO LISTAS

- 2° maneira (mais eficiente e legível): Utilizando a estrutura **for in**

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "José", "Maria", "Julia"];

  for (String nome in nomes) {
    nome = nome.toUpperCase();
    print(nome);
  }
}
```

PERCORRENDO LISTAS

- É possível até mesmo percorrer a lista dessa outra maneira a partir de um determinado item utilizando a função ***sublist***:

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "José", "Maria", "Julia"];

  for (String nome in nomes.sublist(2)) {
    nome = nome.toUpperCase();
    print(nome);
  }
}
```

- A função ***sublist*** recebe como parâmetros o índice de início e o índice de fim da lista derivada que você pretende criar. Se o parâmetro de fim for omitido, entende-se que inicia-se no índice de início e termina no fim natural da lista.

PERCORRENDO LISTAS

- 3° maneira: Utilizando o **forEach**

```dart
void main() {
  List<String> nomes = ["Douglas", "Tião", "José", "Maria", "Julia"];

  nomes.forEach((nome) {
    nome = nome.toUpperCase();
    print(nome);
  });
}
```

- O forEach recebe como parâmetro uma função que irá ser repetida para cada item da coleção. No caso acima passamos uma função anônima que recebe a nossa lista nome e aplica o toUpperCase para todos os itens e em seguida mostra no console o valor.

PRINCIPAIS FUNÇÕES DE LISTAS

- Preenchendo uma lista com o mesmo item repetidas vezes:

```
1 void main() {
2   List<int> maluca = List.filled(100, 4);
3   print(maluca);
4 }
5
```

- O construtor **List.filled( )** recebe como primeiro parâmetro a quantidade de vezes que o item deve se repetir, o segundo parâmetro recebe a o item a ser repetido.

PRINCIPAIS FUNÇÕES DE LISTAS

- Criando uma lista que contém vários itens ou valores que são gerados por uma função:

```
void main() {
  List<int> doida = List.generate(10, funcao);
  print(doida);
}

int funcao(int pos) {
  return pos * 10;
}
```

- O construtor **List.generate( )** recebe dois parâmetros. O primeiro é o tamanho da lista a ser gerada. Como segundo parâmetro ele recebe uma função e passa como parâmetro dessa função o índice do item da lista.

PRINCIPAIS FUNÇÕES DE LISTAS

- Pode-se ainda mudar essa função para uma função anônima o lugar de uma função nomeada:

```
Run | Debug
1  void main() {
2    List<int> doida = List.generate(10, (i) => i * 10);
3    print(doida);
4  }
5
```

VERIFICANDO SE UMA LISTA ESTÁ VAZIA

- O método .**isEmpty** retorna *true* caso a lista esteja vazia e false caso ela não esteja:

```dart
void main() {
  List<int> lista = List.filled(10, 10);
  print(lista.isEmpty);
}
```

- Já o método .**isNotEmpty** retorna o contrário:

```dart
void main() {
  List<int> lista = List.filled(10, 10);
  print(lista.isNotEmpty);
}
```

VERIFICANDO O CONTEÚDO DE UMA LISTA

- O método **.any( )** recebe uma função e testa todos os elementos da lista para ver se algum destes itens atende ao requisito do teste passado pela função:

```
void main() {
  List<int> doida = List.generate(10, (i) => i * 10);
  doida.removeAt(0);
  print(doida);
  print(doida.any((i) => i % 10 == 0));
}
```

- Neste exemplo, estamos testando se cada item da lista é divisível por 10. Se algum deles for divisível, ela retorna *true*, caso contrário retorna *false*.

VERIFICANDO O CONTEÚDO DE UMA LISTA

- O método **.firstWhere( ),** também recebe uma função e retorna o primeiro item da lista que atenda a condição do teste:

```dart
void main() {
  List<int> doida = List.generate(10, (i) => i * 10);
  doida.removeAt(0);
  print(doida);
  print(doida.any((i) => i % 10 == 0));
  print(doida.firstWhere((i) => i % 40 == 0));
}
```

VERIFICANDO O CONTEÚDO DE UMA LISTA

- Em contraste o método **.lastWhere( )**, também recebe uma função e retorna o último item da lista que atenda a condição do teste:

```
void main() {
  List<int> doida = List.generate(10, (i) => i * 10);
  doida.removeAt(0);
  print(doida);
  print(doida.any((i) => i % 10 == 0));
  print(doida.firstWhere((i) => i % 40 == 0));
  print(doida.lastWhere((i) => i % 40 == 0));
}
```

VERIFICANDO O CONTEÚDO DE UMA LISTA

- Já o método **.where( )** retorna todos os elementos da lista que satisfazem a condição. Veja:

```
void main() {
  List<int> doida = List.generate(10, (i) => i * 10);
  doida.removeAt(0);
  print(doida);
  print(doida.any((i) => i % 10 == 0));
  print(doida.firstWhere((i) => i % 40 == 0));
  print(doida.lastWhere((i) => i % 40 == 0));
  print(doida.where((i) => i % 40 == 0));
}
```

GERANDO NOVAS LISTAS A PARTIR DE OUTRA LISTA

- O método **.map( )** recebe uma função e nos retorna uma nova lista:

```
void main() {
  List<int> doida = List.generate(10, (i) => i * 10);
  doida.removeAt(0);
  print(doida);
  print(doida.any((i) => i % 10 == 0));
  print(doida.map((i) => i + 1));
}
```

LISTAS E *NULL SAFETY*

- Caso você deseje inserir valores nulos em sua lista é necessário mudar seu tipo para *nullable* assim como já fizemos anteriormente em outras aulas.

```
void main() {
  List<String> lista1 = [];
  List<String?> lista2 = [];
  List<String>? lista3;
}
```

- A primeira lista é uma lista de *strings* vazia. (não é *null*)
- A segunda lista aceita valores *null* devido ao seu tipo.
- A terceira lista pode ser *null* (a própria lista é *null*)

MAPS

- Um *map* é uma coleção de objetos que possui um objeto simples baseado em pares de chave / valor. A chave e o valor de um mapa podem ser de qualquer tipo.
- Veja a sintaxe:

```
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};

  print(ddds[11]);
}
```

- Caso seja especificado uma chave inexistente o Map retorna o tipo de elemento que ele retorna normalmente só que *nullable* (no nosso exemplo retornaria uma String?)

MAPS

- Para saber o tamanho de um *map* é só utilizar o método **.lenght**.
- Para adicionar um novo elemento no *map* é só passar a chave entre colchetes e atribuir o novo item:

```
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};
  print(ddds[11]);
  print(ddds.length);
  ddds[61] = "Brasilia";
  print(ddds);
}
```

MAPS

- Se você tentar adicionar um novo item ao *map* com a chave coincidido com uma chave já existente, o item original será sobrescrito.
- Para remover um item do **map**, basta o usar o método .**remove( )** e passar a chave do item:

```
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};
  print(ddds[11]);
  print(ddds.length);
  ddds[61] = "Brasilia";
  ddds[0] = "Sei lá";
  print(ddds);
  ddds.remove(0);
  print(ddds);
}
```

MAPS

- Maps não podem ter duas chaves iguais, mas valores iguais são permitidos.
- O método .**containsKey( )** retorna *true* ou false e verifica se o *map* contem aquela chave. Você deve passar a chave a ser consultada por parâmetro.
- O método .**containsValue( )** retorna *true* ou false e verifica se o *map* contem aquele valor. Você deve passar o valor a ser consultada por parâmetro.
- O método .**isEmpty** retorna *true* se o *map* está vazio e *false* se ele não está vazio.
- O método **.clear( )** limpa todos os itens do *map*.

MAPS

- Parecido com as listas, para percorrer o mapa e realizar alguma ação utilizamos o **forEach( )**.Veja:

```
Run | Debug
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};
  ddds[61] = "Brasilia";
  ddds.forEach((key, value) {
    print("Chave: $key Valor: $value");
  });
}
```

MAPS

- O método .addAll( ) recebe por parâmetro outro *map* e adiciona esse *map* dentro do nosso *map* já existente:

```
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};
  ddds[61] = "Brasilia";
  ddds.addAll({98: "Cidade 1", 99: "Cidade 2"});
  print(ddds);
}
```

MAPS

- É possível remover todos os itens de um *map* que atendam uma certa condição com o método .**removeWhere(**). Este método recebe uma função e aplica ela para cada item. Caso o resultado seja *true* ele remove o item. Veja o exemplo:

```
void main() {
  Map<int, String> ddds = {11: "São Paulo", 19: "Campinhas", 41: "Curitiba"};
  ddds[61] = "Brasilia";
  ddds.removeWhere((key, value) => key > 20);
  print(ddds);
}
```

MAPS

- Sobre o operador ?, o comportamento é o mesmo de que com as listas.
- **IMPORTANTE: Embora uma valor num *map* possa ser nulo, uma chave jamais de ser nula.**

EXERCÍCIOS

- 1 - Crie uma função que recebe uma lista de números inteiros como parâmetro e retorna a soma de todos os elementos da lista.
- 2 - Crie uma função que recebe uma lista de números inteiros como parâmetro e retorna o maior valor presente na lista.
- 3 - Crie uma função que recebe uma lista de números inteiros como parâmetro e retorna uma nova lista contendo apenas os números pares.
- 4 - Crie uma função que recebe uma lista de números inteiros como parâmetro e retorna a média dos valores presentes na lista.

EXERCÍCIOS

- 5 - Desenvolva uma função que recebe uma string como argumento e conta quantas palavras únicas (sem repetição) estão presentes nessa string. Considere que as palavras são separadas por espaços.
- 6 - Crie uma função que recebe uma lista de números inteiros e uma função de filtro como parâmetros. A função de filtro deve ser aplicada a cada elemento da lista, e a função principal deve retornar uma nova lista contendo apenas os elementos que passaram no filtro.

EXERCÍCIOS

- 7 - Crie uma função que recebe uma lista de palavras como parâmetro e retorna um mapa onde as chaves são as palavras únicas da lista e os valores são a contagem de vezes que cada palavra aparece.
- 8 - Crie um mapa que armazene palavras em inglês como chaves e suas traduções em português como valores. Crie uma função que recebe uma palavra em inglês como parâmetro e retorna a tradução em português, caso exista, caso contrário retorne uma mensagem "Tradução não encontrada". Para testes utilize o seguinte mapa: (no próximo slide)

EXERCÍCIOS

- Map<String, String> dicionario = {
- "apple": "maçã",
- "banana": "banana",
- "orange": "laranja",
- "avocado": "abacate",
- "strawberry": "morango",
- "pineaple": "abacaxi"
- };

EXERCÍCIOS

- 9 - Crie um mapa onde as chaves são os nomes dos alunos e os valores são listas de notas. Crie uma função que recebe o mapa como parâmetro e retorna um novo mapa com as médias de notas de cada aluno.

REFERÊNCIAS

- CIOLFI, Daniel; DUTRA, Ewerton; STARTTO.DEV. Curso de Criação de Apps Android/iOS/Web com Flutter - 5 cursos em 1. Disponível na plataforma Udemy.
- LIMA. **Dart - Coleções**. 2022. Disponível em: https://acervolima.com/dart-colecoes/. Acesso em: 28 ago. 2023.