

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);  
}
```

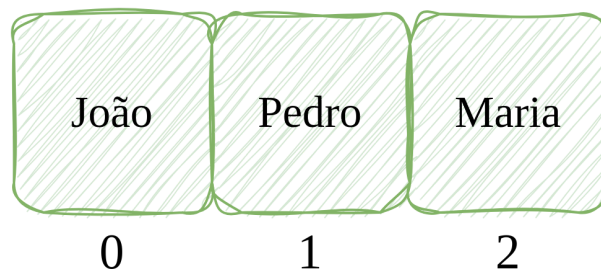
Podemos acessar os elementos de uma lista usando o operador [] também. Cada um tem a sua posição, começando a contagem do zero.

```
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 qualquerCoisa = [1, true, 'Ana'];
    //<List<String> com type annotation
    var somenteStrings = <String> ['Ana', 'Pedro'];
    print(qualquerCoisa.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>? podeSerNullEH = null;
  //Null
  print(podeSerNullEH.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 paises = {'Brasil', 'Brasil'};  
    print(paises);  
}
```

Se construímos um objeto usando {} e ele possui pelo menos um elemento “comum”, ele é um conjunto. Se construímos 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 portugueses = {'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']);
}

```

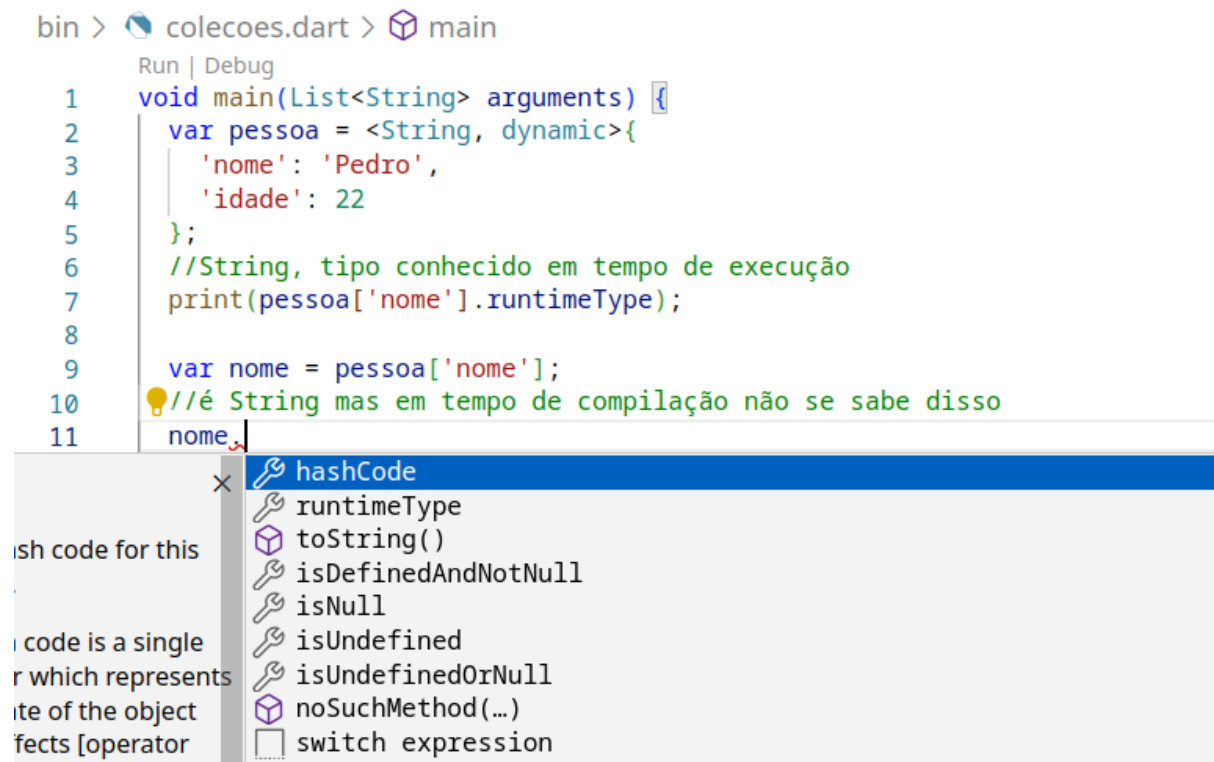
Suponha que tenhamos um mapa cujos valores são do tipo “dynamic”. Qual o resultado do seguinte programa?

```

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

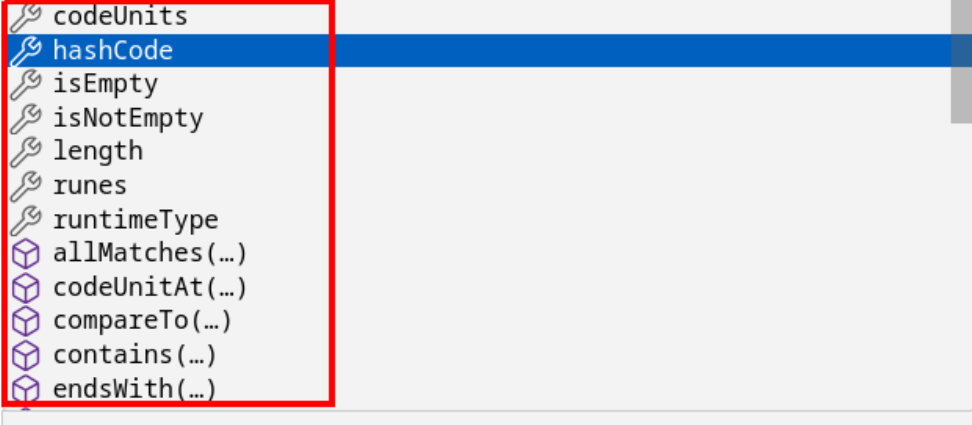
```

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.



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("Título?");
  String? titulo = stdin.readLineSync();
  print("Gênero?");
  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.