PROGRAMAÇÃO DE DISPOSITIVOS MÓVEIS
PARTE-1 LINGUAGEM DART

Sumário

UNIDADE 1: AMBIENTE DE PROGRAMAÇÃO

1.1 INTRODUÇÃO

Dart é uma linguagem criada pelo Google em 2011. A linguagem DART vem sendo apoiada e mantida por uma grande comunidade nos últimos anos.

É uma linguagem relativamente nova, comparada a outras linguagens. Adotou princípios e conceitos incorporados em linguagens como Javascript, Java, C++ e C#. Tais características tornam a sua curva de aprendizado relativamente fácil para programadores com experiência prévia em linguagens semelhantes.

Dart é considerada uma linguagem multiplataforma. A partir dela e de seus frameworks é possível gerar aplicações nativas e de alta performance em back-end, front-end (transpilando para JavaScript), em desktop (Windows, Linux, Mac e Chromebook), em dispositivos mobile (Android, iOS).

1.2 CARACTERÍSTICAS DA LINGUAGEM DART

As seguintes características são inerentes à linguagem DART:

1. Tipada: Dart é uma linguagem fortemente tipada. Todas as variáveis e constantes precisam ter seu tipo especificado.

2. Orientada a objetos: Dart suporta o paradigma de orientação a objetos. Permite a criação de classes, objetos, herança, encapsulamento e polimorfismo.

3. Funcional: Dart também suporta o paradigma funcional. Permite a criação de closures, funções anônimas e métodos de alta ordem.

4. Baseada em classes: Dart é baseada em classes. Classes são o ponto central da estruturação de um aplicativo Dart.

5. Compilada: Dart é uma linguagem compilada. O código Dart é compilado para um código de máquina antes de ser executado.

6. Escalável: Dart é uma linguagem escalável. É possível criar aplicativos Dart de pequeno e grande porte, desde pequenos scripts até aplicativos complexos.

7. Moderna: Dart é uma linguagem moderna, com sintaxe clara, fácil de entender e com recursos avançados, como suporte a ***streams***, ***assincronismo***, ***inferência de tipo***, e outros.

8. Multiplataforma: Dart é uma linguagem multiplataforma. É possível criar aplicativos Dart para várias plataformas, como web, desktop e dispositivos móveis.

1.3 INSTALAÇÃO DO SDK DA LINGUAGEM DART

O Software Development Kit (SDK) é um conjunto de softwares composto por compilador, utilitários e bibliotecas da linguagem.

O SDK dá suporte ao processo de edição, compilação, depuração e manutenção do código de uma aplicação.

O SDK opera em conjunto com uma IDE (Integrated Development Environment).

Um IDE é um software para criar aplicações que combina ferramentas comuns de desenvolvedor em uma única interface de usuário gráfica (GUI).

O SDK está disponível para download no site https://dart.dev/get-dart/archive. Na seção ***stable channel*** é possível selecionar o número da versão estável e o sistema operacional associado.

Um arquivo zipado com a atual versão estável reúne todos os componentes do SDK.

Após fazer o download, descompacte o arquivo zipado em um diretório.

O Path (caminho relativo do diretório) onde reside o diretório ***bin*** deverá compor a lista de diretórios da ***variável de ambiente PATH*** do sistema operacional.

Figura-01 Site SDK do DART

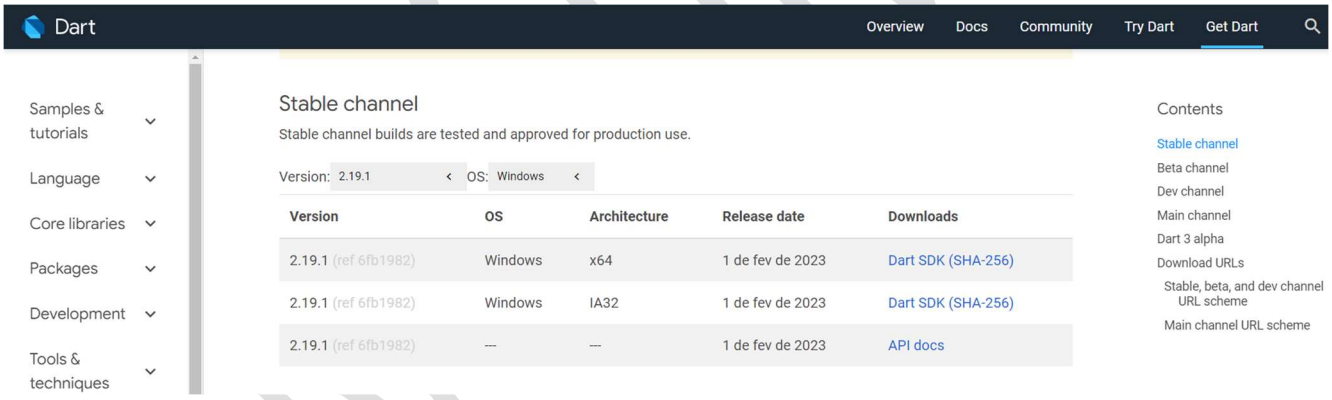


Figura-02 Diretório com o SDK descompactado

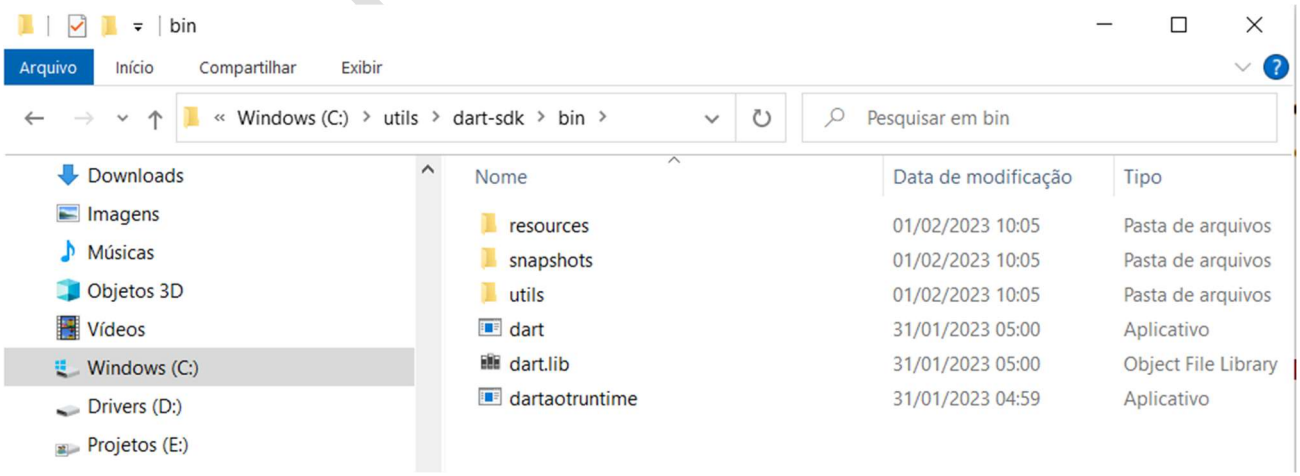


Figura-03 Acesso pelo Painel de Controle às Propriedades das Variáveis de Ambiente

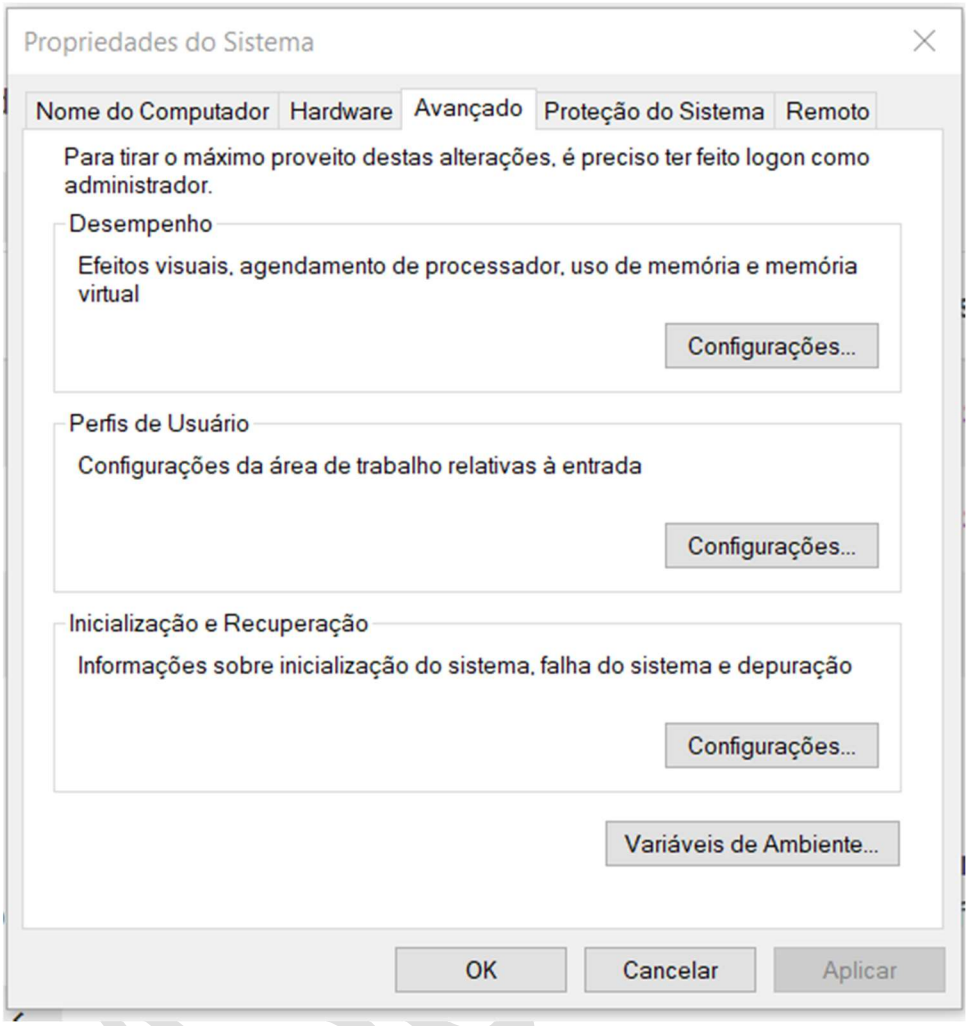


Figura-04 Variáveis de Ambiente do Windows

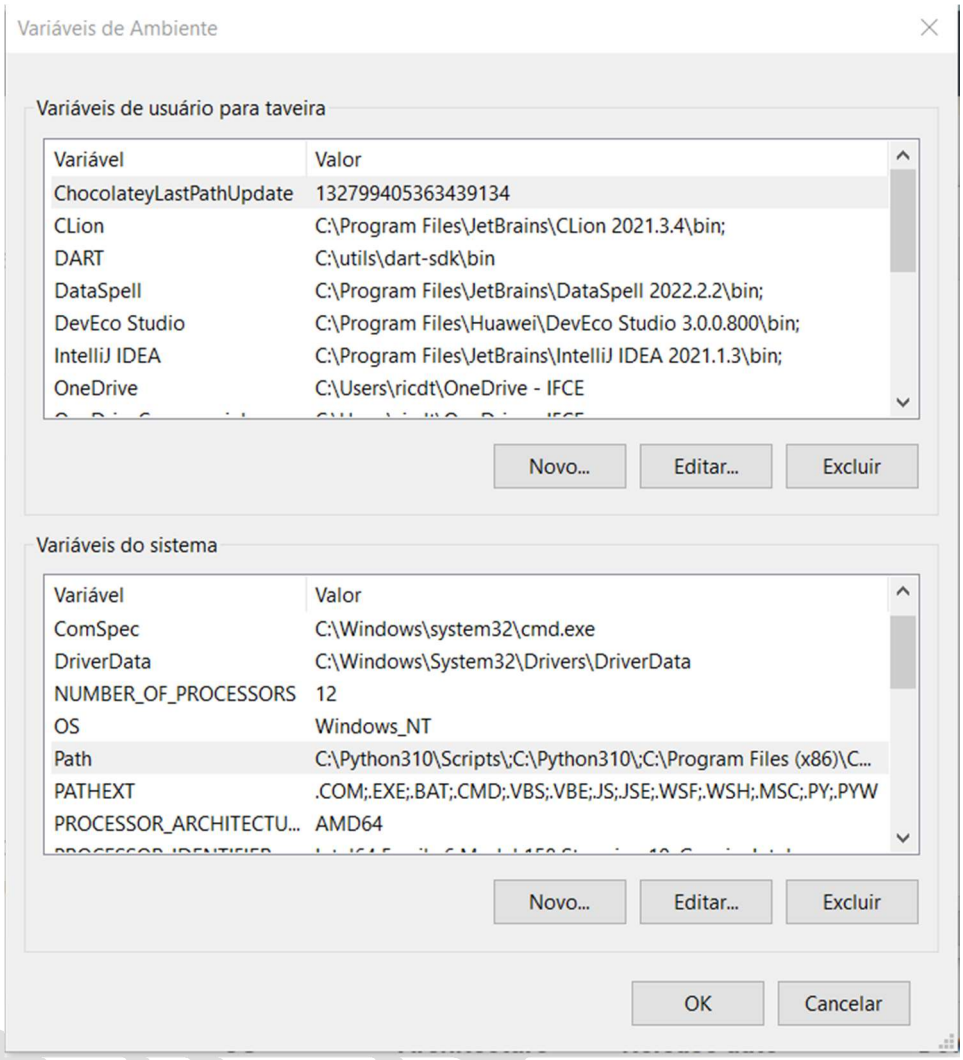


Figura-05 Editar Variável de Ambiente PATH

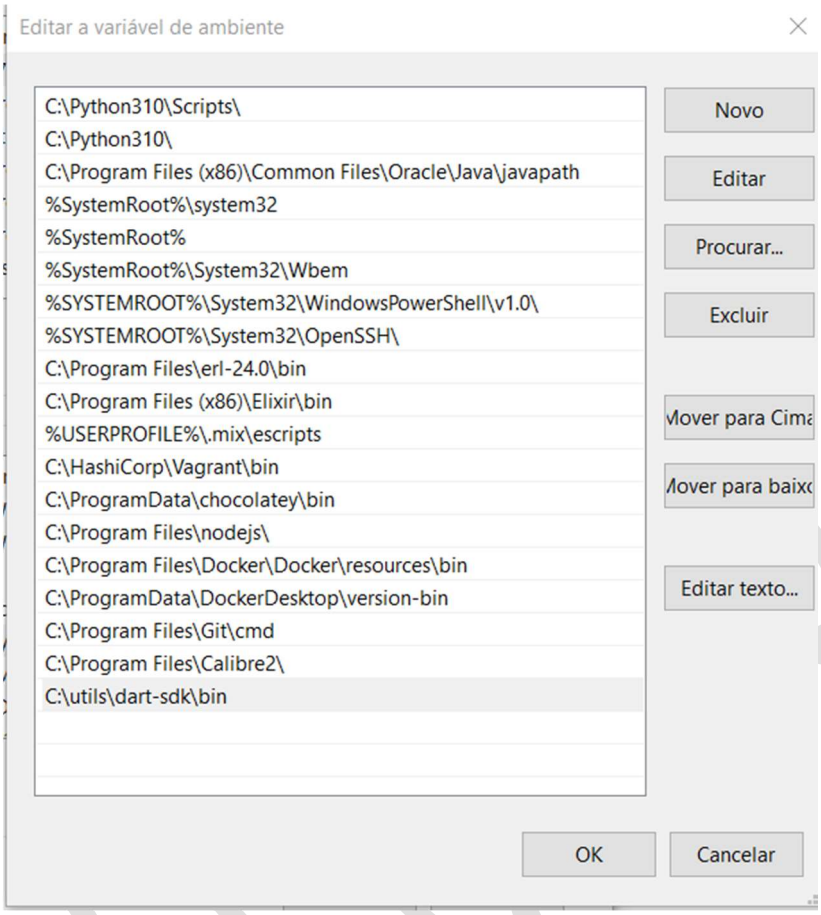
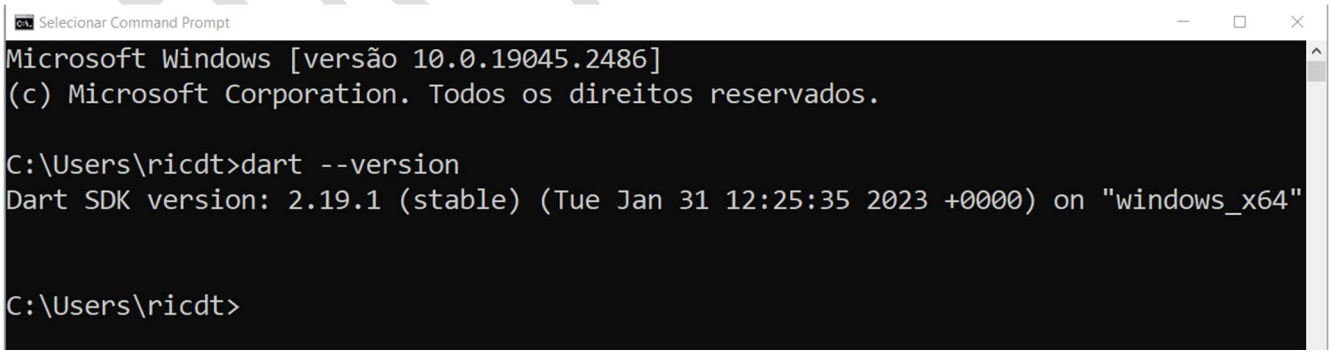


Figura-06 Teste de acesso ao SDK pela linha de comando



1.4 TESTE DO SDK

Um teste de uso do SDK do Dart pode ser feito da seguinte forma:

1. Criar um ***snippet*** (pequeno trecho de código) DART e salvá-lo em um diretório acessível;
2. Entrar nesse diretório com o comando ***cd <nome-do-diretório>;***
3. Executar o snippet DART com o comando ***dart run hello.dart***;

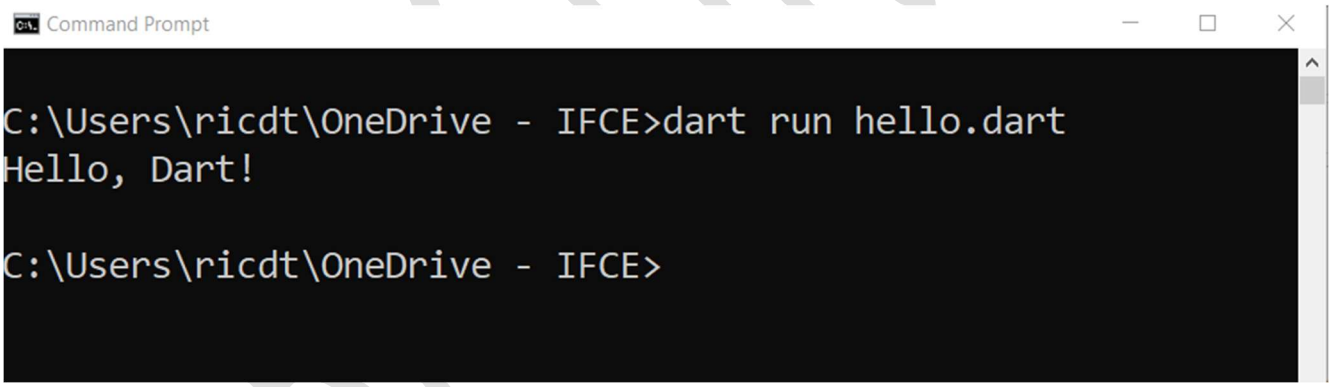
1.4.1 CÓDIGO DO SNIPPET

Criar um arquivo ***hello.dart*** em um diretório onde é permitido a gravação de arquivos.

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

1.4.2 COMPILAÇÃO E EXECUÇÃO DO SNIPPET

Figura-07 Execução do Snippet hello.dart



1.5 IDE VSCODE

O Visual Studio Code é uma IDE que se integra aos principais SDK´s das linguagens de programação.

Desenvolvido pela Microsoft é disponibilizado gratuitamente.

1.5.1 INSTALAÇÃO DA IDE VSCODE NO WINDOWS

Seguir os passos definidos abaixo para instalar o VSCode no sistema operacional Windows:

1. Baixe o instalador do Visual Studio Code na página oficial do software: https://code.visualstudio.com/

2. Execute o instalador baixado e siga as instruções na tela para instalar o software.

1.5.2 INSTALAÇÃO DA IDE VSCODE NO LINUX

Seguir os passos definidos abaixo para instalar o VSCode no sistema operacional Linux:

1. Abra o terminal e digite o seguinte comando para adicionar o repositório do Visual Studio Code ao seu sistema:

```
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg

curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg

sudo sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos/vscode stable main" > /etc/apt/sources.list.d/vscode.list'
```

2. Atualize sua lista de pacotes com o seguinte comando:

```
sudo apt-get update
```

3. Instale o Visual Studio Code com o seguinte comando:

```
sudo apt-get install code
```

1.6 ATIVIDADE-01

Executar as seguintes atividades:

1. Instalar o SDK do Dart;
2. Testar a execução de um snippet Dart;

1.7 ATIVIDADE-02

Executar as seguintes atividades:

1. Instalar o VsCode;
2. Testar a execução de um snippet Dart no ambiente do VsCode.

UNIDADE 2: EXPRESSÕES E OPERADORES

2.1 INTRODUÇÃO

Nessa unidade iniciamos a apresentação dos elementos que compõe a linguagem como comentários, expressões e operadores.

2.2 COMENTÁRIOS NO CÓDIGO

Os comentários são usados para adicionar descrições ou anotações ao código DART de forma a ajudar na compreensão do código que virá em seguida e na diferentes partes do código do aplicativo.

Os comentários podem aparecer no código Dart nas seguintes formas:

1. Comentários de linha única

```
// Este é um comentário de linha única
```

2. Comentários em múltiplas linhas

```
/*
Este é um comentário
de
múltiplas linhas.
*/
```

2.2 INTRODUÇÃO A OPERADORES E SENTENÇAS

A programação de computadores consiste em usar uma linguagem de programação de alto nível para definir uma série de comandos que serão traduzidos para uma linguagem binária. A linguagem binária gerada é reconhecida pela máquina e executa passo a passo as instruções.

O entendimento dessa etapa inicia o aprendizado de algoritmos. O papel de uma linguagem de programação de alto nível é abstrair esse código binário e permitir a comunicação entre humanos e as máquinas através de instruções lógicas que nós, humanos, entendemos.

Embora haja diferenças de funcionalidades e de sintaxe entre as várias linguagens de programação, elas possuem em comum dois conceitos que são a base para a construção de instruções: os operadores e as estruturas de controle.

Seguiremos apresentando os operadores disponíveis em Dart.

2.2.1 OPERADORES

Os operadores são usados na construção de expressões. Expressões são trechos de código que possuem, alteram ou executam alguma ação em um valor armazenado em memória em tempo de execução.

O código `x = y - z` é considerado uma expressão. Inicialmente acessa as variáveis **y** e **z**. Subtrai o valor da variável **z** ao valor da variável **y** usando o operador de subtração **-**.
Em seguida atribui o valor da diferença à variável **x** usando o operador de atribuição **=**.

A linguagem Dart possui um conjunto de operadores, que serão apresentados em categorias de acordo com as expressões em que são usados.

2.2.2 OPERADOR ARITMÉTICO

Operadores aritméticos são usados em expressões que realizam operações matemáticas.

Operador	Descrição	Exemplo
+	Adição / Concatenação	50 + 2 / '50' + '2'
-	Subtração	50 - 7
-expressão	Negação / Inversão do sinal da operação	-(-50)
*	Multiplicação	25 * 2
/	Divisão	25 / 2
~/	Divisão com retorno da parte inteira	25 ~ 2
%	Resto da Divisão	25 % 2

Exemplos de expressões aritméticas.

```
void main() {
  print(50 + 2); // > 52
  print('50' + '2'); // > 502
  print(50 - 7); // > 43
  print(-(-50)); // > 50
  print(25 * 2); // > 50
  print(25 / 2); // > 12.5
  print(25 ~/ 2); // > 12
  print(25 % 2); // > 1
}
```

2.3.2 OPERADOR RELACIONAL E DE IGUALDADE

Operadores relacionais e de equalidade realizam comparações entre os valores de diferentes objetos.

Operador	Descrição	Exemplo
==	Equalidade	50 == 50
!=	Subtração	50 != 50
>/>=	Negação / Inversão do sinal da operação	50 > 50 e 50 >= 50
</<=	Multiplicação	50 < 50 e 50 <= 50

Exemplos de expressões relacionais e de equalidade.

```
void main() {
  print(50 == 50); // > true
  print(50 != 50); // > false
  print(50 > 50); // > false
  print(50 >= 50); // > true
  print(50 < 50); // > false
  print(50 <= 50); // > true
}
```

2.3.3 OPERADORES LÓGICOS

Operadores para a construção de expressões booleanas.

Operador	Descrição	Exemplo
&&	AND	true && true
||	OR	true || true
!expressão	Negação / Inversão do sinal da operação	!false

Exemplos de expressões com operadores lógicos.

```
void main() {
  print(50 == 50 && 50 <= 10); // > false
  print((50 == 50 && 50 <= 10) || 52 != 52); // > false
  print(!(50 == 50 && 50 <= 10) || 52 != 20); // > true
}
```

2.3.4 OPERADORES DE MANIPULAÇÃO DE BITS

Operadores que realizam alteração em bits de valores inteiros.

Operador	Descrição	Exemplo
&	AND	`42 & 27`
|	OR	`42 | 27`
^	XOR	`42 ^ 27`
~	NOT	`~42`
<<	Deslocamento de bit para esquerda	`42<<1`
>>	Deslocamento de bit para direita	`42>>1`
>>>	Deslocamento de bit para direita sem sinal (unsigned)	`-42>>>1`

Exemplos de Operações usando álgebra booleana:

AND

O modificador & representa um operador **AND** em álgebra booleana. Por exemplo, resolvendo a ***expressão*** **`52 & 17`:**

```
52 em binário = 00110100
17 em binario = 00010001
52 and 17     = 00010000
52 and 17 na base 10 = 0016
```

```
void main() {
  var and = 52 & 17;
  print('52 em binário = ' + 52.toRadixString(2).padLeft(8, '0'));
  // > 00110100
  print('17 em binario = ' + 17.toRadixString(2).padLeft(8, '0'));
  // > 00010001
  print('52 and 17     = ' + and.toRadixString(2).padLeft(8, '0'));
  // > 00010000
  print('var and na base 10 = ' + and.toRadixString(10).padLeft(4,'0'));
  // > 00010000
}
```

No estudo das representações numéricas, aprendemos que um ***radix*** é a quantidade de dígitos distintos utilizados para representar os valores. O sistema decimal que utilizamos no dia a dia possui um ***radix*** de **10**, pois os valores são representados em dígitos de ***0 a 9***. Assim como o sistema hexadecimal possui ***radix*** de ***16***, consequentemente o sistema binário possui ***radix 2***, pois utiliza apenas zeros (***0***) e uns (***1***).

Em Dart tudo são objetos, até os números inteiros. ***toRadixString(2)*** é um método de ***int*** que transforma o inteiro em sua representação binária em ***String***, já o método ***padLeft(8, '0')*** de

String adiciona o caractere ***0*** na esquerda até que a ***String*** complete oito caracteres para padronizar a visualização do binário.

OR

O modificador **|** representa um operador **OR** em álgebra booleana, onde o resultado da operação bit a bit entre dois valores será **1** quando um dos dois valores tem o valor ***1***.

```
52 em binário = 00110100
17 em binario = 00010001
52 or 17      = 00110101
52 or 17 na base 10 = 0053
```

```
void main() {
  var or = 52 | 17;
  print('52 em binário = ' + 52.toRadixString(2).padLeft(8, '0'));
  // > 00110100
  print('17 em binario = ' + 17.toRadixString(2).padLeft(8, '0'));
  // > 00010001
  print('52 ord 17     = ' + or.toRadixString(2).padLeft(8, '0'));
  // > 00010000
  print('var or na base 10 = ' + and.toRadixString(10).padLeft(4,'0'));
  // > 00010000
}
```

XOR

O modificador **^** representa um operador **XOR** em álgebra booleana, onde o resultado da operação bit a bit entre dois valores diferente será **1**.

```
52 em binário = 00110100
17 em binario = 00010001
52 xor 17     = 00100101
52 xor 17 na base 10 = 0053
```

```
void main() {
  var xor = 52 ^ 17;
  print('52 em binário = ' + 52.toRadixString(2).padLeft(8, '0'));
  // > 00110100
  print('17 em binario = ' + 17.toRadixString(2).padLeft(8, '0'));
  // > 00010001
  print('52 xor 17     = ' + xor.toRadixString(2).padLeft(8, '0'));
  // > 00100101
  print('52 xor 17 na base 10 =' + xor.toRadixString(10).padLeft(4, '0'));
  // > 0037
}
```

NOT

O modificador **~** representa um operador **NOT** em álgebra booleana. A operação **NOT** inverte todos os bits. Os bits que possuem valor **1** ficarão com valor **0**. Os bits com valor **0** ficarão com valor **1**.

```
void main() {
  var not = ~52;
  print(52.toRadixString(2).padLeft(8, '0')); // > 00110100
  print(not); // > -53
  print(not.toRadixString(2)); // > -110101
}
```

SHIFT direita

O modificador **>>** faz o deslocamento de bits para a direita. A operação consiste em deslocar N bits para a direita. O resultado é equivalente a dividir o valor da variável envolvida na operação por N vezes por 2. Por exemplo: 52 >> 1 é igual a 52 / (1 * 2) = 26.

```
void main() {
  var shift = 52 >> 1;
  print(52.toRadixString(2).padLeft(8, '0')); // > 00110100
  print(shift); // > 26
  print(shift.toRadixString(2).padLeft(8, '0')); // > 00011010
}
```

SHIFT esquerda

O modificador **<<** faz o deslocamento de bits para a esquerda. A operação consiste em deslocar N bits para a esquerda. O resultado é equivalente a multiplicar o valor da variável envolvida na operação por N vezes por 2. Por exemplo: 52 << 1 é igual a 52 * 1 * 2 = 104.

```
void main() {
  var shift = 52 << 1;
  print(52.toRadixString(2).padLeft(8, '0')); // > 00110100
  print(shift); // > 104
  print(shift.toRadixString(2).padLeft(8, '0')); // > 01101000
}
```

SHIFT direita unsigned

O modificador **>>>** faz o deslocamento de bits para a direta. A operação consiste em deslocar N bits para a direita. A operação se comporta diferente para números negativos. O bit contendo o sinal é descolado para direita e é substituído por 0. .

```
void main() {
  var shift = -42 >>> 1;
  print((-42).toRadixString(2).padLeft(8, '0')); // > 0-101010
  print(shift); // > 9223372036854775787
  print(shift.toRadixString(2));
// > 1111111111111111111111111111111111111111111111111111111101011
}
```

2.3.5 OPERADORES DE ATRIBUIÇÃO

Os operadores de atribuição recebem valores de expressões que são avaliadas e o atribuem a uma variável a esquerda do operador de atribuição. Na tabela abaixo são apresentados os operadores de atribuição.

Operador	Descrição	Exemplo
=	Atribuição	a = 4;
+=	Adição e atribuição	a += 5;
-=	Subtração e atribuição	a -= 4;
*=	Multiplicação e atribuição	a *= 11;
/=	Divisão e atribuição	a /= 5;
~=	Divisão com retorno da parte inteira e atribuição	a ~= 2;
%=	Resto da Divisão e atribuição	a % = 4;
&=	AND e atribuição	b &= 70;
|=	OR e atribuição	b |= 34;
^=	XOR e atribuição	b ^= 34;
<<=	Subtração e atribuição	b <<= 4;
>>=	Subtração e atribuição	B
>>>=	Subtração e atribuição	

2.3.6 OPERADORES DE INCREMENTO E DECREMENTO

Operadores que adicionam ou diminuem o valor de uma variável numérica. Normalmente são usados como variáveis que controlam a contagem na execução de um loop (estrutura de repetição).

Operador	Descrição	Exemplo
++var	Incrementa 1 antes de utilizar a variável	var a = ++b;
var++	Incrementa 1 após utilizar a variável	var a = b++;
--var	Decrementa 1 antes de utilizar a variável	var a = --b;
var --	Decrementa 1 após utilizar a variável	var a = b--;

```
void main() {

  var a = 0;
  var b = 1 + ++a; // 1 + 1
  print(a); // > 1
  print(b); // > 2

  var c = 0;
  var d = 1 + --c; // 1 + -1
  print(c); // > -1
  print(d); // > 0
}
```

2.4 OPERADORES DE VALIDAÇÃO DE TIPOS

Operadores são usados para verificar se uma determinada variável é de

Operador	Descrição	Exemplo
as	Conversão de tipo	27 as num;
is	Validação de tipo, ***true*** se for do tipo testado	27 is int
is!	Validação de tipo, ***true*** se não for do tipo testado	27 is! int

```
// Converte o tipo num para int
// em seguida acessa a propriedade bitlenght

void main() {
  num a = 42;
  print((a as int).bitLength); // > 6
}
```

```
// Testa o tipo da variável num

void main() {
  num a = 42;
  print(a is int); // > true
  print(a is! String); // > true
```

2.5 OPERADORES GERAIS

Operador	Descrição	Exemplo
.	Acesso a membros	27.bitLength
()	Chamada em função	print(27)
..	Operações em cascata	StringBuffer..write('Hello')
...	Operador spread	var alfabeto = [... vogais, ...consoantes]
expressaoA ? expressaoB : expressaoC	Ternário	var valor = b % 2 == 0 ? 'par' : 'impar'
[ ]	Acesso em listas e mapas	Letras[1]

2.7 CASCADE: ..

O operador cascade .. facilita a construção de um código fluído. Permite a criação de chamadas em forma de cascata para uma mesma referência de objeto.

```
final frase = StringBuffer();
frase.write('Operação ');
frase.write('em ');
frase.write('cascade.');
```

```
void main() {
 final frase = StringBuffer()
   ..write('Operação ')
   ..write('em ')
   ..write('cascade.');
 print(frase); // > Operação em cascade.
}
```

2.8 SPREAD

O operador spread fornece uma sintaxe açúcar (syntax sugar) de forma a facilitar a leitura e a construção de código. Nesse caso do Dart permite a inserção de uma lista em outra lista.

```
void main() {
 final frase = StringBuffer()
   ..write('Operação ')
   ..write('em ')
   ..write('cascade.');
 print(frase); // > Operação em cascade.
}
```

2.7 ATRIBUIÇÃO NULO

2.9 TERNÁRIO

O operador ternário executa uma operação booleana condicional com três operandos. Em uma expressão como ***a ? b : c.*** Onde ***a*** é uma expressão booleana. Se ***a*** for ***true***, o resultado será ***b***, mas se ***a*** for false, o resultado será ***c***.

```
void main() {
  int x = 22;
  print(x % 2 == 0 ? 'par' : 'ímpar'); // > par
}
```

2.10 ACESSO A ITENS: []

O operador [ ] é usado em expressões onde é necessário acessar os valores de uma lista por meio do seu índice ou dos valores de um mapa usando a chave do elemento correspondente.

```
void main() {
  final map = {
    'vogais': 'a,e,i,o,u',
    'consoantes': 'b,c,d,...',
  };
  final vogais = ['a', 'e', 'i', 'o', 'u'];
  print(vogais[0]); // > a
  print(vogais[4]); // > u
  print(map['vogais']); // > a,e,i,o,u
```

2.11 OPERADORES DE NULIDADE

Operadores de nulidade são usados para evitar o lançamento de uma exceção devido ao uso de variáveis com uma referência nula.

A linguagem Dart introduziu a funcionalidade Null-safety .

A necessidade de usar esses operadores diminuiu, já que por padrão as variáveis não aceitam mais referências nulas, mas ainda assim são úteis em alguns casos.

Operador	Descrição	Exemplo
??	Ternário para valores nulos	var x = z ?? 20;
??=	Atribuição caso nulo	valor ??= 50;
?.	Acesso a atributos caso não nulo	notafiscal?.cpf;
?..	Operação em cascata caso não nulo	itemNotaFiscal?..calcularValor()
?[]	Acesso a índices caso não nulo	itemNotaFiscal?[10]
Expressão!	Garantia de aviso ao compilador para objetos que podem ser nulos.	itemNotaFiscal!.imprimirMensagem()

2.11.1 TERNÁRIO NULO: ??

Funciona de forma similar a um operador ternário, pois permite atribuir um valor a uma determinada variável se tal condição for nula.

Dada a expressão ***a = b ?? c***, a variável ***a*** receberá o valor de ***b*** apenas se *a* não for nula, caso contrário, receberá o valor de ***c***. Sendo assim, escrever ***a = b ?? c***, é o equivalente a ***a = b != null ? b : c***.

2.11.2 ATRIBUIÇÃO NULO: ??=

Operador de atribuição que de forma similar ao operador anterior. Funciona como atalho caso o valor seja atribuído na própria variável cuja nulidade deve ser verificada. Dada a expressão: a ??= b, a variável ***a*** receberá o valor de ***b*** apenas se ela for nula. Escrever ***a ??= b*** é o equivalente a **a = a ?? b**.

```
void main() {
  //admite que a variável quantidade pode ser nula
  int? quantidade = null;
  quantidade ??= 10; // se quantidade for nula, recebe o valor
  print(quantidade); // > 10
}
```

2.11.3 ACESSO NULO: ?.

O Operador Acesso Nulo permite acesso a propriedades e métodos de um objeto de forma segura contra referências nulas. Pode facilmente ser encadeado em uma sequência de chamadas **a?.b?.c**. Ao acessar uma referência nula em tempo de execução, a expressão resulta em Null, não lançando um erro.

```
void main() {
  int? resposta = null;
  resposta ??= 42;
  print(resposta); // > 42
}
```

2.11.4 CASCADE NULO: ?..

Assim como o operador de acesso **.** possui o seu equivalente **?.** para validação de nulos, o operador em cascade **..** também possui o seu equivalente ***?..*** .

```
void main() {
  StringBuffer? frase = null;
  frase?..write('Operação ')
    ..write('em ')
    ..write('cascade.');
  print(frase); // > null
}
```

Caso o objeto de origem seja nulo, todas as chamadas em cascata retornarão null, evitando uma exceção.

2.11.5 ACESSO A ITENS NULOS: ?:[]

Valida o acesso a valores de uma lista ou map quando a variável pode conter um valor nulo.

```
void main() {
  List<String>? vogais;
  print(vogais?[1]); // > null
}
```

O operador ***?[ ]*** usado em ***vogais?[1]*** retorna null porém não levanta uma exceção.

2.11.6 EXPRESSÃO!

O operador ***!*** no fim de uma expressão é utilizado para converter uma variável possivelmente nula para o seu valor original, forçando o compilador a ignorar qualquer aviso de inconsistência.

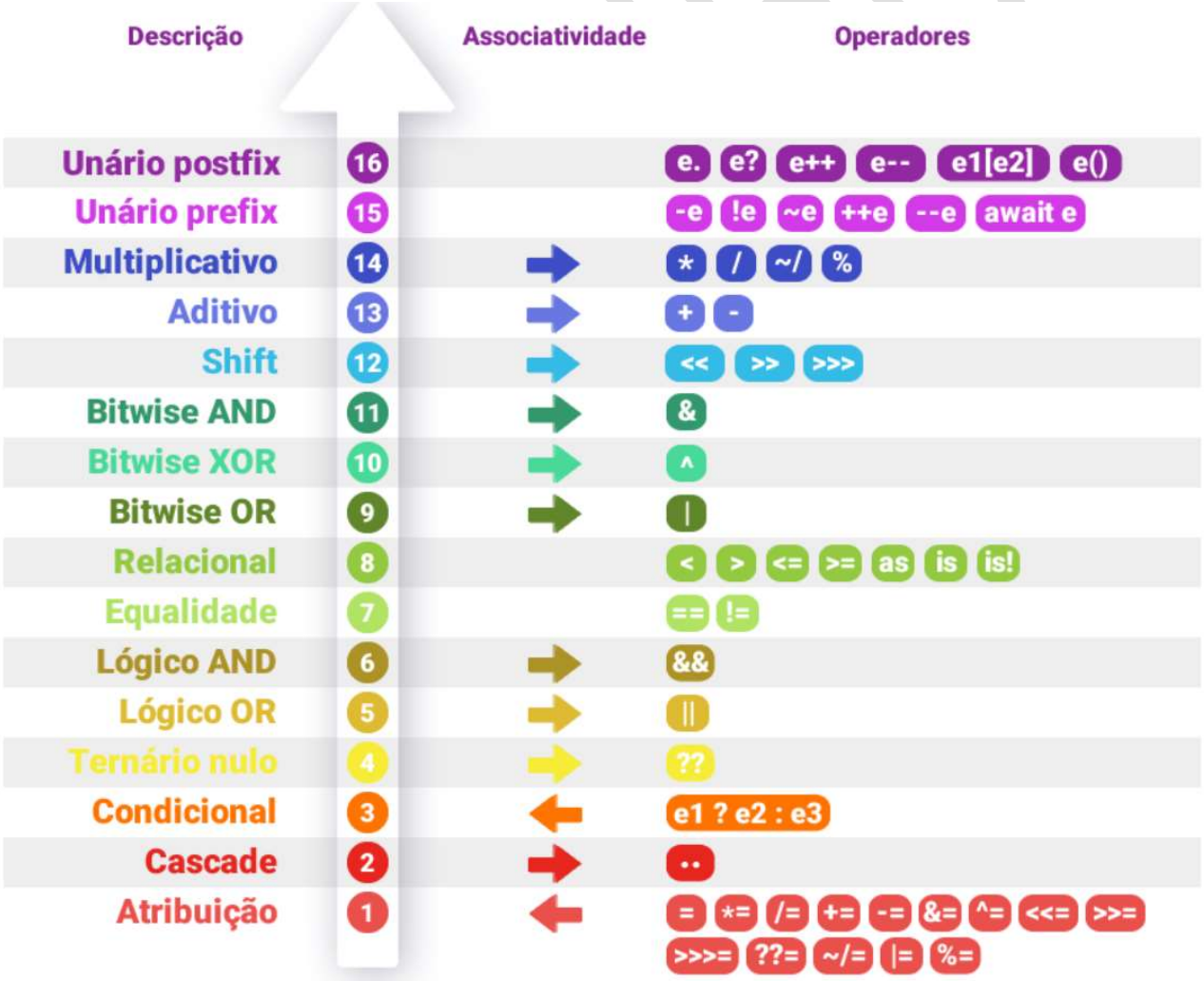
```
void main() {
  List<String>? vogais;
  print(vogais?[1]); // > null
}
```

2.12 PRECEDÊNCIA DE OPERAÇÕES

Aprendemos a avaliar o valor de uma expressão aritmética como 3 + 5 * 2. O resultado será 13. Nos foi ensinado que as operações de multiplicação e a divisão possuem prioridade, em relação a adição e subtração. Da mesma forma, nas expressões os operadores da linguagem Dart, obedecem uma convenção de prioridade na execução. Isso influencia diretamente o resultado das ***expressões***.

Na figura abaixo os operadores Multiplicativo (valor 14) tem uma ordem de precedência maior que os operadores Aditivo (valor 13). Como citado na expressão 3 + 5 * 2, primeiro é feita a multiplicação, em seguida a adição é realizada com o resultado da multiplicação.

Figura-xx



UNIDADE 3: VARIAVÉIS E TIPOS

3.1 INTRODUÇÃO

Na linguagem Dart a variável é um espaço na memória do computador que é usado para armazenar um valor. É um nome que se refere a um valor em tempo de execução e pode ser alterado ao longo do tempo.

Associado a uma variável há um tipo de variável que é uma forma de definir as operações e a representação binária com a qual a variável poderá armazenar. Dessa forma a definição de um tipo ajuda o compilador a garantir que o código esteja correto em tempo de compilação e também ajuda os desenvolvedores a entenderem melhor o que está acontecendo no código no decorrer de sua execução.

Dessa forma, se você declarar uma variável do tipo ***int***, ela só pode armazenar valores inteiros. Da mesma forma, se você declarar uma variável do tipo ***String***, ela só pode armazenar valores de texto.

3.2 VARIAVEIS NUMÉRICAS

Variáveis numéricas são variáveis que armazenam valores numéricos. Existem três tipos de variáveis numéricas em Dart: ***int*** (inteiro), ***double*** (número de ponto flutuante) e ***num*** (número genérico).

```
int idade = 25;
double salário = 5000.00;
num desconto = 5.5;
```

3.3 VARIAVEIS DE TEXTO

Variáveis de texto são variáveis que armazenam valores de texto. Em Dart, essas variáveis são do tipo ***String*** e ***StringBuffer***.

Variáveis do tipo ***String*** são imutáveis. Isto é, a sequência de caracteres não pode ser alterada.

```
String nome = "João";
```

Já uma variável do tipo **StringBuffer** é uma classe que representa uma sequência modificável de caracteres.

Para usar uma variável do tipo **StringBuffer**, você primeiro precisa criar uma nova instância da classe usando o construtor padrão, em seguida os vários métodos como ***writeAll()***, ***writeCharCode()***, ***clear()***, ***remove()***, ***replace()***, ***substring()***, entre outros. O Método ***toString()*** é usado para converter a sequência de caracteres em uma String para ser impresso como sáida.

```
void main() {
  var buffer = new StringBuffer();

  // adiciona várias cadeias de caracteres
  buffer.write('Olá');
  buffer.write(' ');
  buffer.write('Mundo!');

  print(buffer.toString()); // saida: Olá Mundo!
}
```

3.4 VARIAVEIS BOOLEANAS

Variáveis booleanas são variáveis que armazenam valores ***true*** (verdadeiro) ou ***false*** (falso). Em Dart, essas variáveis são do tipo ***bool***.

```
bool valido = true;
```

3.5 VARIAVEIS DINAMICAS

Variáveis dinâmicas são variáveis que podem armazenar qualquer tipo de valor. Em Dart, essas variáveis são do tipo ***dynamic***.

```
dynamic valor = "um texto";

valor = 30.5;
```

3.4 CONSTANTES

Constantes, que são variáveis cujo valor não pode ser alterado após a sua definição. Em Dart, as constantes são definidas usando a palavra-chave ***const***.

```
const pi = 3.14;
```

3.5 INFERÊNCIA DE TIPOS

A inferência de tipo em Dart é uma funcionalidade que permite que o compilador determine automaticamente o tipo de uma variável com base no valor atribuído a ela. Em outras palavras, você não precisa especificar o tipo de uma variável ao declará-la se o compilador puder determinar o tipo com base no contexto.

Por exemplo, em vez de declarar uma dessa forma:

```
const pi = 3.14;
```

Poderá fazê-lo dessa forma:

```
var nome = "João";
```

Dessa forma , o compilador irá inferir automaticamente que a variável nome é do tipo ***String***, com base no valor atribuído a ela.

3.6 GENERICS

Generics em Dart é uma forma de escrever código que permite que tipos sejam parametrizados.

Suponha que você precise desenvolver uma classe que represente um par de valores com o mesmo tipo, por exemplo, um par de inteiros ou um par de strings.

Para fazer isso, você precisa definir diferentes classes, uma para um par de inteiros e outra para um par de strings. Por exemplo:

```
class ParInt {
  int x;
  int y;
  ParInt(this.x, this.y);
}

class ParString {
  String x;
  String y;
  ParString(this.x, this.y);
}
```

A solução acima não é escalável quando se pretende ter pares de vários tipos. Isso é resolvido usando o conceito de genéricos.

Generics permite parametrizar os tipos de funções, classes e métodos.

Por exemplo, o código a seguir define uma classe que representa um par de valores com o mesmo tipo.

```
class Par<T> {
  T x;
  T y;
  Par(this.x, this.y);
}
```

A letra **T** dentro dos colchetes **<>** é o tipo. Por convenção, as variáveis de tipo têm nomes com uma única letra, como **T**, **E**, **S**, **K** e **V**.

No código abaixo é criado um par de inteiros. Você especifica **int** como o tipo **T** ao criar um novo objeto **Par.**

```
var parStr = Par<String>('José', 'Maria');

print('x=${paiStr.x}, y=${parStr.y}');
```

O código abaixo apresenta um exemplo de como usar Generics em Dart. Está disponível em **https://github.com/ricdtaveira/mobdev-parte-01/blob/master/07-variaveis-tipos/07-generics-01.dart.**

UNIDADE 4: COLEÇÕES

4.1 INTRODUÇÃO

Coleções são estruturas de dados que permitem armazenar, organizar e manipular um conjunto de valores relacionados a variáveis de um determinado tipo (int, double, String, bool) ou objetos (instancia de uma determinada classe). Existem três tipos principais de coleções em Dart: Listas, Conjuntos e Mapas.

4.2 LISTAS

Uma Lista é uma coleção indexável de objetos com um comprimento. Uma lista pode conter elementos duplicados e **null**. Dart usa a **List<E>** classe para gerenciar listas.

4.2.1 CRIANDO UMA LISTA

A seguinte expressão cria uma lista vazia de números inteiros.

```
int scores = [];
```

Para criar uma lista e inicializar seus elementos, coloque uma lista de elementos separados por vírgulas dentro dos colchetes ([]). Neste exemplo, o Dart infere a lista como List<int>.

Por exemplo, o seguinte cria uma lista de números inteiros que possuem 4 números:

4.2.2 EXIBINDO UMA LISTA

```
void main() {
  var scores = [1, 3, 4, 2];
  print(scores); // [1, 3, 4, 2];
}
```

4.2.3 ACESSANDO ELEMENTOS

4.2.4 ATRIBUINDO VALORES AOS ELEMENTOS

4.2.5 ADICIONADO ELEMENTOS A UMA LISTA

4.2.6 REMOVENDO ELEMENTOS DE UMA LISTA

4.2.6 LISTA IMUTÁVEL

4.2.7 LISTAR PROPRIEDADES

4.2.8 ITERAR SOBRE OS ELEMENTOS DA LISTA

4.2.9 ESPALHANDO ELEMENTOS DA LISTA

4.2.10 COLEÇÃO SE

4.2.11 COLEÇÃO PARA

4.3 CONJUNTOS

Listas

4.4 MAPAS

Listas

UNIDADE 5: FLUXO DE CONTROLE

5.1 INTRODUÇÃO

5.2 COMPARAÇÕES

5.3 VALORES BOOLEANOS

5.4 OPERADORES BOOLEANOS

5.5 LÓGICA BOOLEANA

5.6 SENTENÇA IF

5.7 ESCOPO DE VARIAVEL

5.8 SENTENÇA SWITCH

5.8.1 SWITCH COM NUMÉRICOS

5.8.2 SWITCH COM STRINGS

5.8.3 TIPO ENUMERADO

5.8.4 SWITCH COM ENUMS

5.9 LOOPS

5.9.1 WHILE

5.9.2 DO-WHILE

5.9.3 FOR-IN

5.9.4 FOR-EACH

5.10 FUNÇÕES

5.10.1ANATOMIA DE UMA FUNÇÃO

UNIDADE 6: PROGRAMAÇÃO ORIENTADA A OBJETO

6.1 INTRODUÇÃO

A Orientação a Objeto é uma pratica de Engenharia de Software que envolve os processos de analise, projeto e programação. A Programação Orientada a Objeto é um paradigma de programação que em conjunto aos processos de análise e projeto ajudam a construir software por meio de composição e interação de unidades de software chamados de objeto.

6.2 PARADIGMA ORIENTADA A OBJETO

Um paradigma é uma abordagem ou modelo utilizado para resolver problemas dentro de uma determinada área de conhecimento. O paradigma fornece uma forma de pensar e trabalhar como um conjunto de regras, práticas e técnicas que orientam a solução de problemas dentro dessa área de conhecimento.

A Programação Orientada a Objeto (POO) é um paradigma de programação que se concentra na criação de classes implementadas por meio de objetos que têm propriedades (atributos) e comportamentos (métodos) e que interagem entre si para resolver problemas.

6.3 CONCEITOS BÁSICOS DA POO

Os seguintes conceitos caracterizam a POO:

1. Classe: Uma classe é um modelo ou uma descrição de um objeto. Ela define as propriedades e comportamentos do objeto.
2. Objeto: Um objeto é uma implementação da descrição de uma classe na memória. É uma instância de uma classe. Na criação de um objeto ou no decorrer de sua existência são atribuídos valores às suas propriedades e os métodos (funções) definidas na classe são executados pelo objeto.
3. Encapsulamento: O encapsulamento define como os dados e comportamentos de um objeto devem estar protegidos dentro do objeto e não acessíveis externamente.
4. Herança: A herança permite que uma classe herde as propriedades e comportamentos de outra classe. A classe que herda é chamada de subclasse e a classe que é herdada é chamada de superclasse.
5. Polimorfismo: O polimorfismo permite que um objeto possa assumir várias formas ou comportamentos. Isso significa que um objeto pode ser tratado como um objeto de sua própria classe ou como um objeto de uma de suas subclasses.
6. Abstração: A abstração implica que na definição de classes haja uma restrição àquilo que é relevante para o seu uso no problema em questão. Isso significa que apenas as características e comportamentos importantes são incluídos em uma classe.

Esses conceitos são a base da POO e ajudam os programadores a desenvolverem programas modularizados, reutilizáveis e fáceis de manter. Adiante iremos exemplificar esses conceitos nos próximos tópicos.

6.4 CLASSES E OBJETOS

A Programação Orientada a Objetos consiste em conceber um código onde a solução de um determinado problema identifica os seguintes elementos:

1. Um escopo do problema que define a abrangência e os limites da atuação da solução para resolver o problema;
2. As entidades ou coisas que pertencem ao escopo do problema;
3. As classes que são as definições codificadas das entidades que implementam a solução do problema;
4. As relações comportamentais e estruturais entre as classes que pertencem ao escopo do problema;

6.4.1 CLASSE

Uma classe é uma estrutura codificada em uma linguagem de Programação Orientada a Objeto que define o comportamento e as propriedades de uma entidade. A classe é como um modelo que define um tipo que possui dados e funcionalidades. .

6.4.2 OBJETO

Um objeto é uma instância de uma classe. Uma instância é como uma variável de programa que é alocada na memória, possui uma referência e que contém dados e comportamentos específicos definidos pela classe à qual pertence.

6.4.3 MEMBROS

Um membro de uma classe em POO é um elemento que pertence a essa classe. Essa definição inclui os atributos e os métodos da classe.

Os atributos também definidos como variáveis de instância são membros que representam as propriedades do objeto e armazenam informações específicas para cada instância da classe. Por exemplo, se a classe é "***`Aluno`***", seus membros de variáveis de instância podem ser "***`matricula`***", "***`nome`***", "***`cpf`***", "***`data_de_nascimento`***", etc.

Os métodos são membros que definem o comportamento do objeto e realizam funcionalidades específicas. Por exemplo, se a classe é "***`NotaFiscal`***", seus membros de método podem ser "***`imprimirNotaFiscal`***", "***`inserirItemdeNotaFiscal`***", "***`calcularValorNotaFiscal`***", etc.

Existem membros de classe, que são compartilhados por todas as instâncias da classe. Esses membros podem ser atributos de classe e métodos de classe, que são definidos usando a palavra-chave "***static***".

Os membros de uma classe podem ser acessados por outras partes do programa usando o nome da classe seguido de um ponto (.) e o nome do membro. Por exemplo, se a classe é "***Calculadora***" e tem um método chamado "***somar***", podemos acessá-lo chamando "***Calculadora.somar()***".

```
class Calculadora {
  static int somar(int a, int b) {
    return a + b;
  }
}
```

No exemplo acima, o método "somar" pertence à classe "Calculadora" e é estático, o que significa que podemos chamá-lo diretamente da classe, sem criar uma instância da classe Calculadora.

```
int resultado = Calculadora.somar(10, 15); // resultado = 25
```

O método ***somar*** é estático. Na sua chamada é usado o nome da classe "Calculadora" para acessá-lo, em vez de criar um objeto dessa classe. Essa característica torna os métodos estáticos úteis para realizar operações que não dependem do estado de um objeto que é a instância específica da classe.

6.4.4 VISIBILIDADE DE ATRIBUTOS E MÉTODOS

Os membros de uma classe (atributos e métodos) possuem visibilidade pública ou privada reconhecida pelo nome do membro. Se o nome do membro começa pelo símbolo _ (sublinhado/underline) o membro terá visibilidade privada ***(private)***, se não começar com _ terá visibilidade pública.

A visibilidade privada implementa o ***encapsulamento*** quando estabelece essa visibilidade para os membros da classe (atributos e métodos).

O acesso a atributos públicos de um objeto pode ser feito usando a notação ***<nome_do_objeto>.<atributo>*** para acessar um atributo do objeto ou ***<nome_do_objeto>.<metodo>*** para acessar um método.

O acesso a atributos e métodos privados só pode ser feito por meio de métodos públicos definidos na classe. É a única forma de acesso a membros privados de objetos.

Os métodos assessores e modificadores que veremos a seguir são métodos públicos que são capazes de acessar membros privados.

6.4.4 O MODIFICADOR LATE

O modificador ***late*** em Dart é utilizado para atrasar a inicialização de variáveis, permitindo que elas sejam inicializadas em um momento posterior ao da declaração.

Normalmente, quando uma variável é declarada, ela precisa ser inicializada imediatamente com um valor ou pode ser declarada sem valor e, em seguida, receber um valor em um momento posterior. No entanto, se uma variável declarada sem valor não for inicializada antes de ser usada, isso resultará em um erro em tempo de execução.

O modificador ***late*** permite que uma variável seja declarada sem valor e inicializada posteriormente antes de ser usada, sem gerar um erro em tempo de execução. A variável é marcada como ***late***, o que significa que ela é declarada sem valor, mas será inicializada antes da primeira leitura.

O código abaixo apresenta um exemplo de como usar o modificador late em Dart. Está disponível em https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-late.dart.

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

  Pessoa() {
    // Inicializando as variáveis após a criação do objeto
    nome = "João";
    idade = 30;
  }

  void imprimirDados() {
    print("Nome: $nome, idade: $idade");
  }
}

void main() {
  Pessoa pessoa = Pessoa();
  pessoa.imprimirDados();
}
```

6.4.5 O MODIFICADOR STATIC

O modificador ***static*** é usado na Linguagem Dart para definir membros de classe que pertencem à classe em si, em vez de pertencer a um objeto. Isso significa que uma única cópia do membro está disponível para todas as instâncias da classe.

O código abaixo exemplifica o uso do modificador static. Está disponível em ***https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-static.dart*.**

```
class Exemplo {
  static int contador = 0;

  Exemplo() {
    contador++;
  }

  static void mostrarContador () {
    print('O contador é $contador');
  }
}

void main() {
  Exemplo objeto1 = Exemplo();
  Exemplo objeto2 = Exemplo();
  Exemplo.mostrarContador(); // Imprime "O contador é 2"
}
```

6.4.7 MÉTODOS ASSESSORES E MODIFICADORES

Métodos assessores e modificadores (conhecidos como ***getters*** e ***setters***) são métodos definidos que permitem acessar e modificar os atributos de um objeto. Eles são usados para garantir o encapsulamento e a proteção dos dados de um objeto.

Os métodos assessores são responsáveis por retornar o valor de um atributo privado de um objeto, enquanto os métodos modificadores são responsáveis por definir o valor de um atributo privado de um objeto.

Em Dart, a sintaxe para métodos assessores é definida usando a palavra-chave ***get*** seguida do nome do método, enquanto a sintaxe para métodos modificadores é definida usando a palavra-chave ***set*** seguida do nome do método.

O código abaixo exemplifica o uso de getters e setters. Está disponível em ***https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-get_set.dart***.

```
class Pessoa {
  String _nome; // atributo privado
  int _idade;     // atributo privado

  String get nome => _nome; // Método assessor

  set nome(String nome) => _nome = nome;  // Método Modificador

  int get idade => _idade; // Método assessor

  set idade(int idade) => _idade = idade;
}

void main() {
  Pessoa pessoa = Pessoa();

  pessoa.nome = "João"; // chama o método modificador "set nome"
  pessoa.idade = 30; // chama o método modificador "set idade"

  print(pessoa.nome); // chama o método assessor "get nome"
  print(pessoa.idade); // chama o método assessor "get idade"
}
```

6.5 RELACIONAMENTO ENTRE CLASSES

Na Programação Orientada a Objetos a implementação de uma aplicação se dá pelo estabelecimento de relações entre os objetos. Essas relações podem ser comportamentais e estruturais. As relações comportamentais são explicitadas na implementação dos métodos

6.5.1 DIAGRAMA DE CLASSES

O Diagrama de Classes é um dos diagramas da UML (Unified Modeling Language), que é uma linguagem visual usada para modelar sistemas orientados a objetos.

O Diagrama de Classes é usado para representar as classes e seus relacionamentos em um domínio (escopo) de um sistema. Ele fornece uma visão geral da estrutura do sistema, mostrando as classes que o compõem, seus atributos, métodos e relacionamentos.

No Diagrama de Classes, cada classe é representada por um retângulo com três compartimentos, contendo o nome da classe, seus atributos e seus métodos. Os

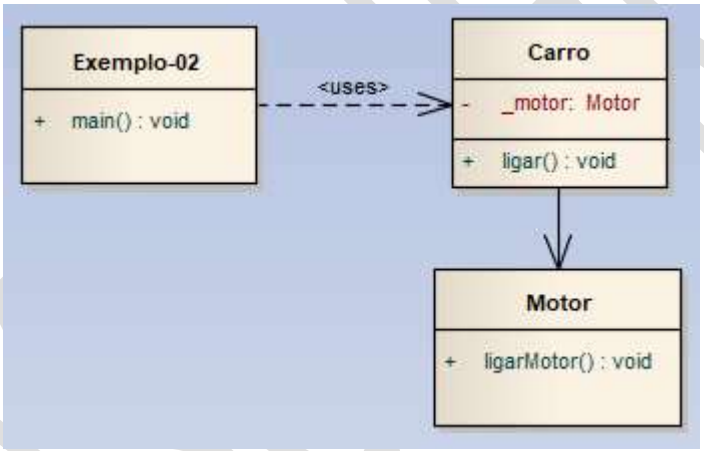
relacionamentos entre as classes são representados por linhas que conectam os retângulos, e podem incluir associações, agregações, composições, generalizações, dependências e interfaces.

O Diagrama de Classes pode incluir outras informações, como visibilidade dos membros, tipos de dados, multiplicações e cardinalidades, estereótipos, etc.

O Diagrama de Classes da UML é uma excelente ferramenta para modelar sistemas orientados a objetos. Fornece uma visão clara e abrangente da estrutura do sistema e suas interações. É um diagrama usado na análise, design e documentação de sistemas de software.

6.5.2 DEPENDÊNCIA

A Dependência é um relacionamento comportamental onde uma classe se apropria da funcionalidade implementada por outra classe. Também é um exemplo de reutilização de código.

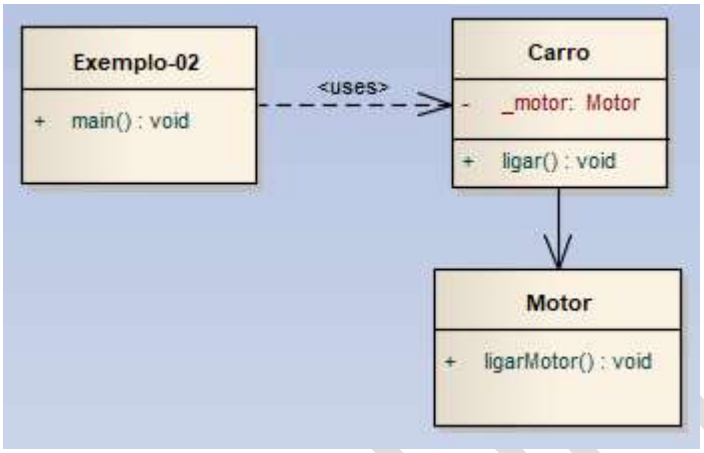


Em um Diagrama de Classes, o Relacionamento de Dependência é representado por uma seta pontilhada direcionada para a classe que é usada. No exemplo acima a classe ***`Exemplo-02`*** faz menção a classe ***`Carro.`*** A menção à classe usada ocorre no bloco de código de um método da classe ou função, ou na lista de parâmetros de um método ou de uma função.

Ver um código exemplo do programa em ***https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-dependencia-00.dart.***

6.5.3 ASSOCIAÇÃO

A Associação é um relacionamento estrutural onde uma classe inclui como seu atributo uma instancia de outra classe. Também é um exemplo de reutilização de código.



No Diagrama de Classes acima, o Relacionamento de Associação é representado por uma seta continua. No exemplo abaixo a classe ***Carro*** tem um atributo ***_motor*** que é uma instancia da ***Motor*** A menção à classe associada ocorre na definição dos atributos na Classe Carro.

Ver o código exemplo do programa ***14-dependencia-00.dart*** no GitHub

6.5.4 HERANÇA E CLASSE ABSTRATA

A Herança é um relacionamento comportamental e estrutural onde uma classe se apropria dos atributos e funcionalidades implementadas por outra classe na forma de uma herança.

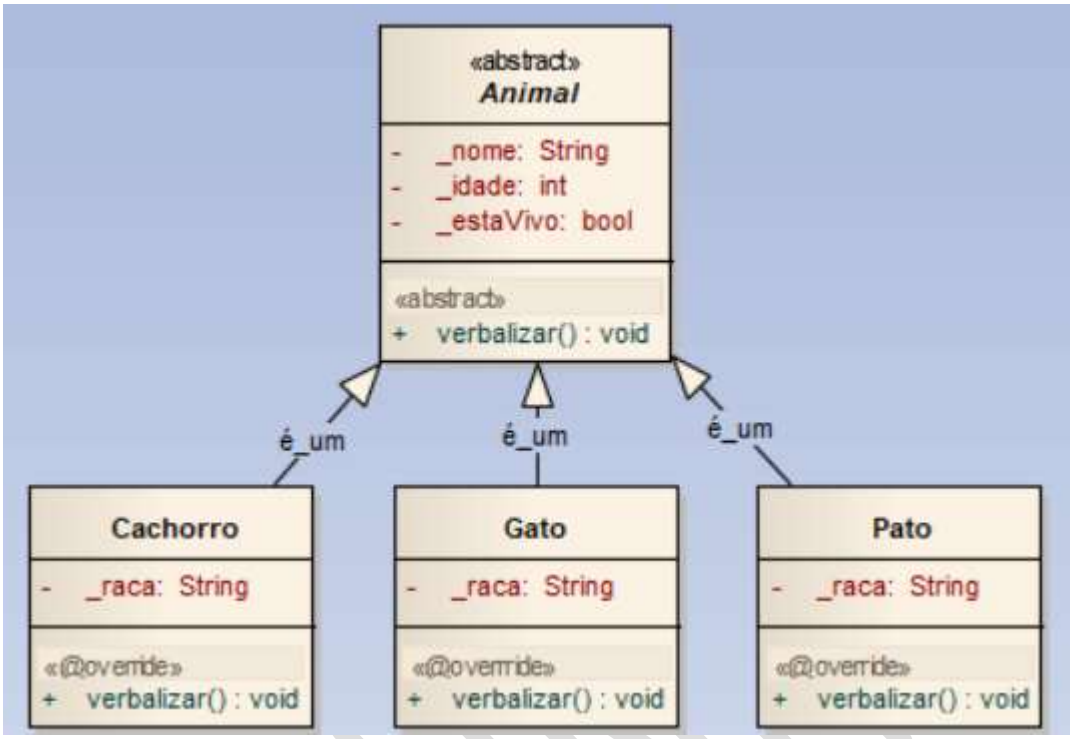
É um relacionamento entre classes do tipo "is a" (É UM). Também é um exemplo de reutilização de código. O significado de uma Herança que envolve duas classes onde a expressão ***é_um*** deve ser coerente no contexto semântico. Por exemplo: A expressão ***Gato é_um Animal*** é correta no contexto semântico. Porém a expressão ***Carro é_um Animal*** é incorreta no seu contexto semântico.

Uma hierarquia de Herança formada de forma incorreta quanto ao contexto semântico incorre em problemas de ambiguidade que por sua vez levam uma aplicação a produzir soluções incorretas.

A criação de uma hierarquia de herança pressupõe a existência de uma classe mãe (superclasse) cujos atributos e métodos serão reutilizados pelas classes filhas (subclasses). O uso de uma classe abstrata inicia uma hierarquia de herança como uma superclasse no qual seus atributos e métodos serão reutilizados pelas classes filhas (subclasse).

Classe Abstrata é uma classe que não é usada para instanciar objetos. Apenas é usada para iniciar uma hierarquia de Herança.

O Diagrama de Classe abaixo apresenta uma hierarquia de herança evidenciada por uma seta contínua apontada das subclasses (Cachorro, Gato, Pato) para a superclasse (Animal).



O Diagrama Classe acima apresenta uma série de características que devem ser observadas na tradução para um código **Dart** que o implementa. São as seguintes características:

1. Estereótipo **<<*abstract*>>**: Apesar da notação de classe abstrata ser evidenciada com o nome da classe usando um tipo de letra em ***itálico***, foi explicitado ainda com o estereótipo **<<*abstract*>>**.
2. Visibilidade dos atributos: Na classe abstrata Animal os atributos estão iniciados com o caractere _ . Essa é uma característica da linguagem DART para tornar o atributo ***private*** e convergir para implementar o conceito de ***encapsulamento***.
3. Método abstrato: Um método abstrato em uma superclasse será sobrescrito por uma implementação específica em cada subclasse. O método sobrescrito (ou "override") é um método que substitui um método existente na classe pai (superclasse). Isso permite que uma classe filha (subclasse) forneça uma implementação diferente para um método que já existe na classe pai. Essa funcionalidade caracteriza um ***polimorfismo de herança***.

Ver um código exemplo do programa em ***https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-heranca-00.dart.***

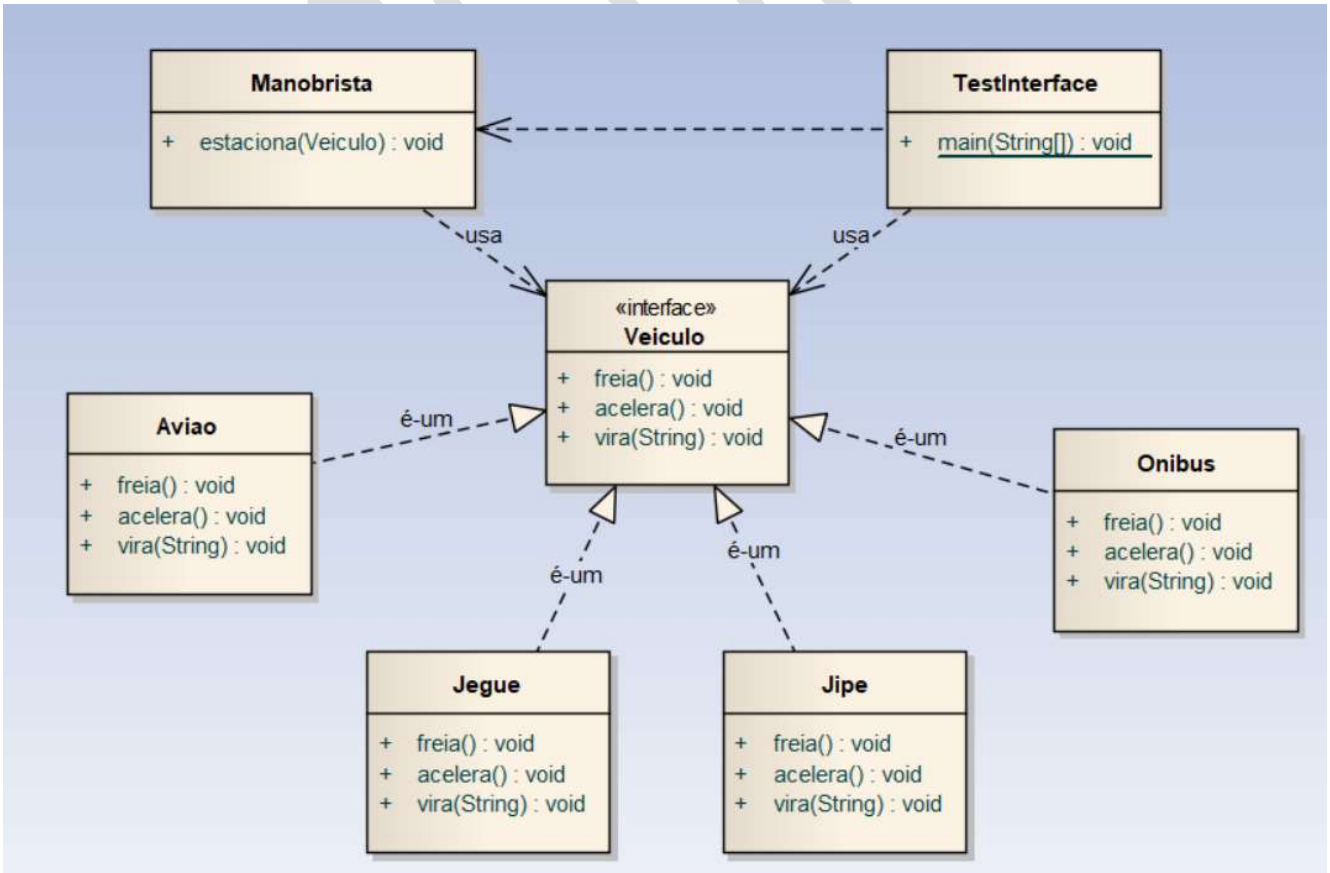
6.5.5 INTERFACE

Uma interface é um contrato entre classes. Ao contrário de outras linguagens, a linguagem Dart não possui a palavra-chave ***interface***.

A implementação de uma interface em Dart é feita definindo uma classe abstrata que possui métodos que definem o contrato que será implementado por classes concretas.

O Diagrama de Classe abaixo apresenta uma interface ***Veiculo*** que define os métodos ***freia()***, ***acelera()***, ***vira()***. A interface é evidenciada por uma seta pontilhada apontada das classes concretas que implementam o contrato da interface (***Aviao***, ***Jegue***, ***Jipe***, ***Onibus***) para a interface ***Veiculo***.

A implementação dos métodos do contrato definidos na interface ***Veiculo*** pelas classes concretas (***Aviao, Jegue, Jipe, Onibus***) evidenciam o ***polimorfismo de interface***. Outra forma de polimorfismo é evidenciada pelo método ***estaciona*** que recebe em tempo de execução um objeto que implementa a interface e executa o contrato de forma peculiar a cada objeto recebido como parâmetro.



Ver um código exemplo do programa em ***https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-interface.dart*.**

6.5.8 AGREGAÇÃO E COMPOSIÇÃO

Agregação e Composição são duas formas de associação entre uma ***Classe Todo*** e uma coleção da ***Classe Parte***. A classe que é Parte é uma ***coleção de objetos*** que pertence aos atributos da ***Classe Todo***.

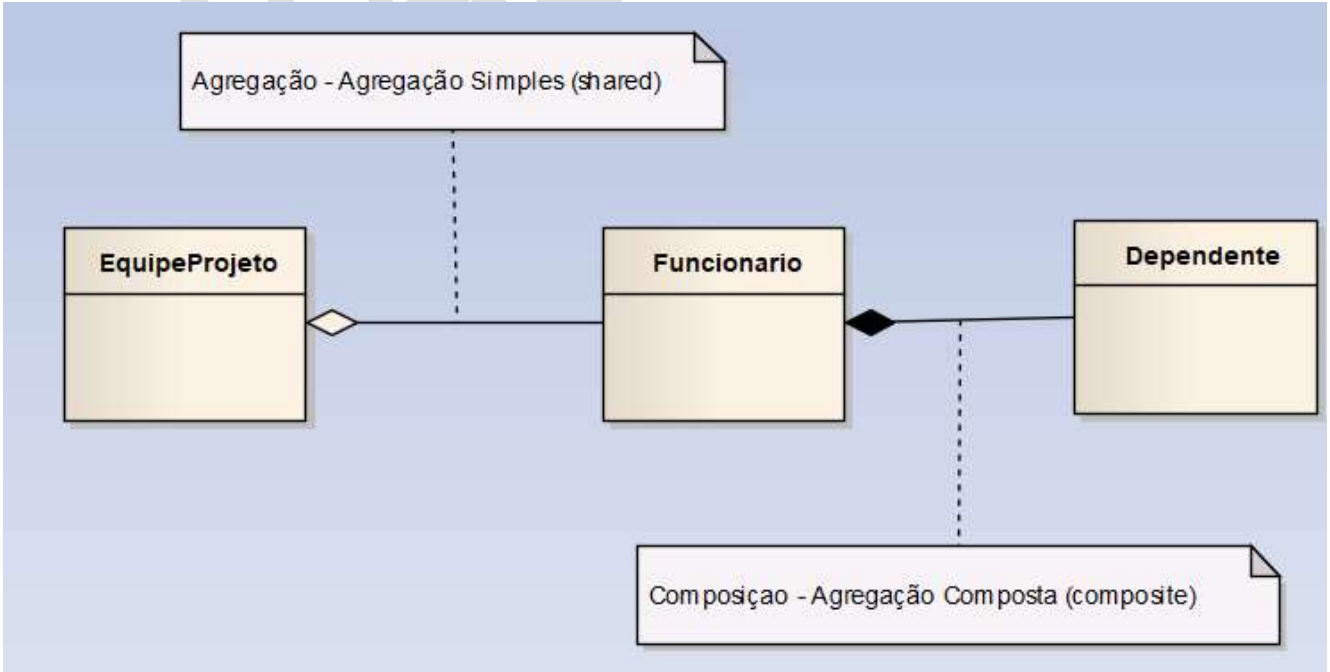
A diferença entre uma Agregação e uma Composição ocorre no ***ciclo de vida*** inerente ao ***objeto Todo*** e os ***objetos Parte***.

Na Agregação quando um objeto Todo é excluído da memória os objetos Parte não serão excluídos da memória.

Na Composição quando um objeto Todo é excluído da memória os objetos Parte também serão excluídos.

O Diagrama de Classes abaixo observa-se o seguinte:

1) A classe EquipeProjeto (Todo) tem uma agregação (losango amarelo) com a classe Funcionario (Parte). Quando um objeto da classe EquipeProjeto é excluído os objetos Funcionário continuarão a existir;
2) Uma coleção de objetos da classe Funcionario será atributo de um objeto da classe EquipeProjeto;
3) A classe Funcionário (Todo) tem uma composição (losango preto) com a classe Dependente (Parte). Quando um objeto da classe Funcionario é excluído os objetos Dependente também serão excluídos.



O Código da implementação do diagrama acima está acessível em: https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-agregacao.dart.

6.5.9 MIXIN

Um mixin é uma classe especial que fornece funcionalidades para outras classes sem herdar delas. Um mixin pode ser usado para compartilhar código entre várias classes sem criar uma hierarquia de herança.

Em outras palavras, os mixins permitem que você adicione comportamentos adicionais a uma classe, sem precisar herdar de outra classe.

Para definir um mixin em Dart, você pode usar a palavra-chave "mixin" em vez de "class".

O código da implementação de um Mixin está em:

https://github.com/ricdtaveira/mobdev-parte-01/blob/master/14-poo/14-mixin.dart.

UNIDADE 7: TRATAMENTO DE EXCEÇÕES

7.1 INTRODUÇÃO

7.2 ERROR VS EXCEPTION

7.3 CONSTRUINDO EXCEÇÕES

7.4 ASSERT

UNIDADE 8: BIBLIOTECAS

8.1 INTRODUÇÃO

8.2 BIBLIOTECAS

8.3 PACOTES

UNIDADE 9: PROGRAMAÇÃO ASSÍNCRONA

9.1 INTRODUÇÃO

9.2 PARALELISMO

9.3 CONCORRENCIA

9.4 PARALELISMO VS CONCORRENCIA

9.5 ISOLATES

9.6 EVENT LOOP

UNIDADE 10: ACESSO A REDE

10.1 INTRODUÇÃO

10.2 TCP/IP

10.3 ACESSO A DISPOSITIVOS DE REDE VIA SOCKET

REFERÊNCIAS

BIBLIOGRAFIA BÁSICA

ARAÚJO. Everton C. **Aprofundando em Flutter. Desenvolva aplicações Dart com Widgets.** Casa do Código, 2022.

BITENCOURT, Julio. **O guia de Dart. Fundamentos, prática, conceitos avançados e tudo mais**. Casa do Código, 2022.

BITENCOURT, Julio. **O guia de Dart. Fundamentos, prática, conceitos avançados e tudo mais**. Casa do Código, 2022.

BIBLIOGRAFIA COMPLEMENTAR

ALESSANDRIA, Simone; KAYFITZ, Brian. **Flutter Cookbook: Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart**. Packt Publishing Ltd, 2021.

BRACHA, Gilad. **The Dart programming language**. Addison-Wesley Professional, 2015.

BELCHIN, Moises; JUBERIAS, Patricia. **Web Programming with Dart**. Apress, 2015.

BIESSEK, Alessandro. **Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2**. Packt Publishing Ltd, 2019.

MEILLER, Dieter. **Modern App Development with Dart and Flutter 2: A Comprehensive Introduction to Flutter**. Walter de Gruyter GmbH & Co KG, 2021.

NAPOLI, Marco L. **Beginning flutter: a hands on guide to app development**. John Wiley & Sons, 2019.

PAYNE, Rap. **Beginning App Development with Flutter: Create Cross-Platform Mobile Apps**. Apress, 2019.

SINHA, Sanjib. **Quick Start Guide to Dart Programming: Create High-Performance Applications for the Web and Mobile**. Apress, 2019.

WINDMILL, Eric. **Flutter in action**. Simon and Schuster, 2020.

ZAMMETTI, Frank; ZAMMETTI, Frank. Flutter: A Gentle Introduction. **Practical Flutter: Improve your Mobile Development with Google's Latest Open-Source SDK**, p. 1-36, 2019.