

1 Introdução

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

Dart é uma linguagem compilada e interpretada.

Código Dart pode ser compilado para:

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

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

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

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

Seu compilador **ahead of time**:

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

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

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

2 Desenvolvimento

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

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

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

Dart > dart:core library

dart:core library

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

This library is automatically imported.

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

Numbers and booleans

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

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

Strings and regular expressions

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

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

`StringBuffer` provides a way to construct strings efficiently.

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

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

dart:core library

CLASSES

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

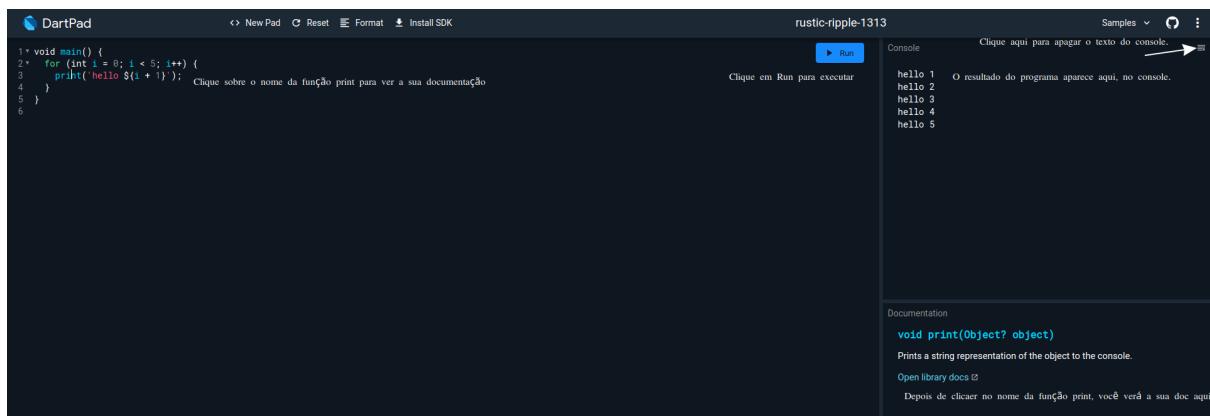
2.2 DartPad - Editor online O ambiente Dart inclui o DartPad, um editor on-line que viabiliza a execução de testes rápidos, particularmente útil para quem está começando. Ele pode ser visitado em

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

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

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

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



The screenshot shows the DartPad interface with the following components:

- Code Editor:** Displays the following Dart code:

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

A tooltip above the code says: "Clique sobre o nome da função print para ver a sua documentação".
- Console:** Shows the output of the code execution:

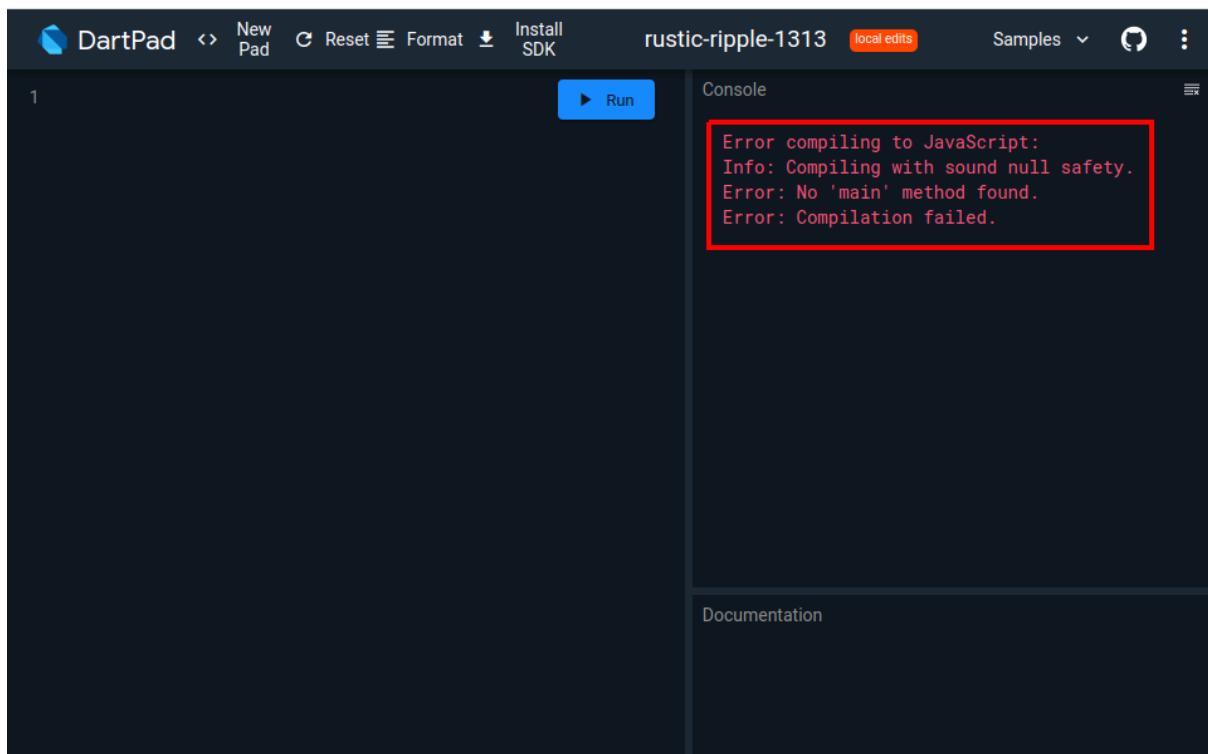
```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

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

```
void print(Object? object)
Prints a string representation of the object to the console.
```

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

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



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

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

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

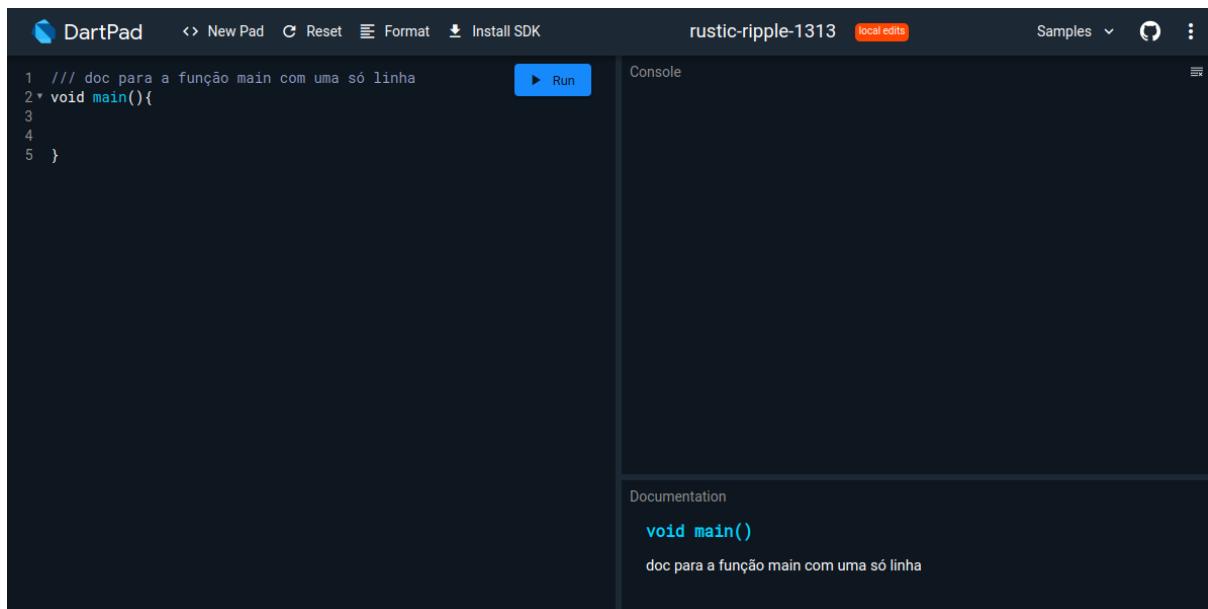
```
void main() {  
  print('Hello, World');  
}
```

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

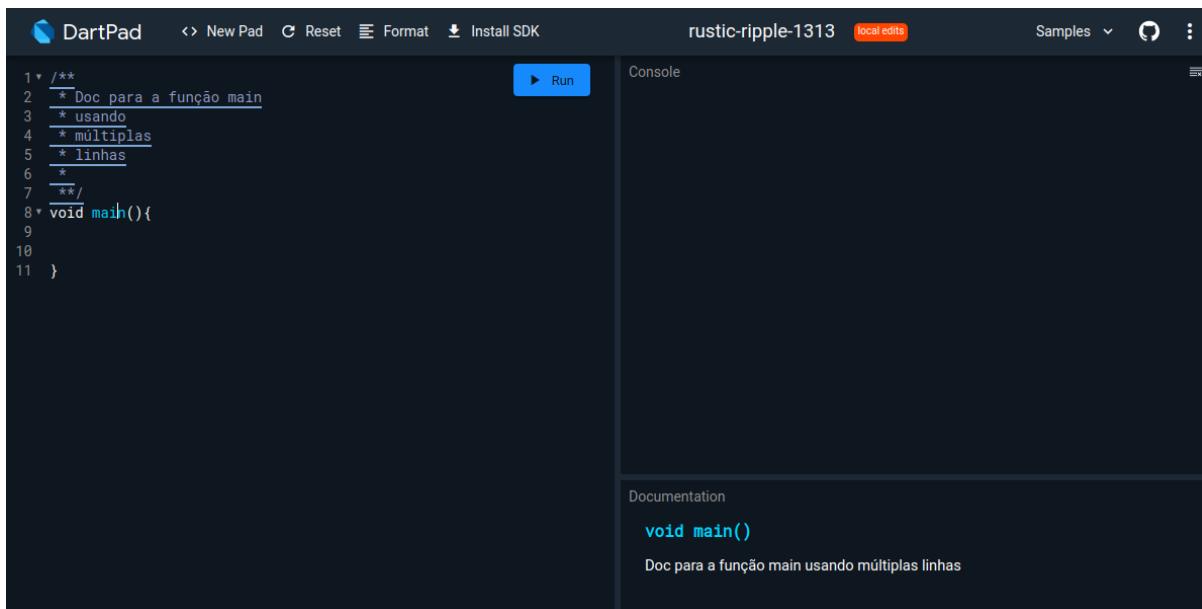
```
void main() {
  //sou um comentário de uma única linha

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

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



Agora um exemplo de comentário de documentação de múltiplas linhas.



The screenshot shows the DartPad interface with the following code in the editor:

```

1 /**
2  * Doc para a função main
3  * usando
4  * múltiplas
5  * linhas
6  */
7 **/
8 void main(){
9
10
11 }

```

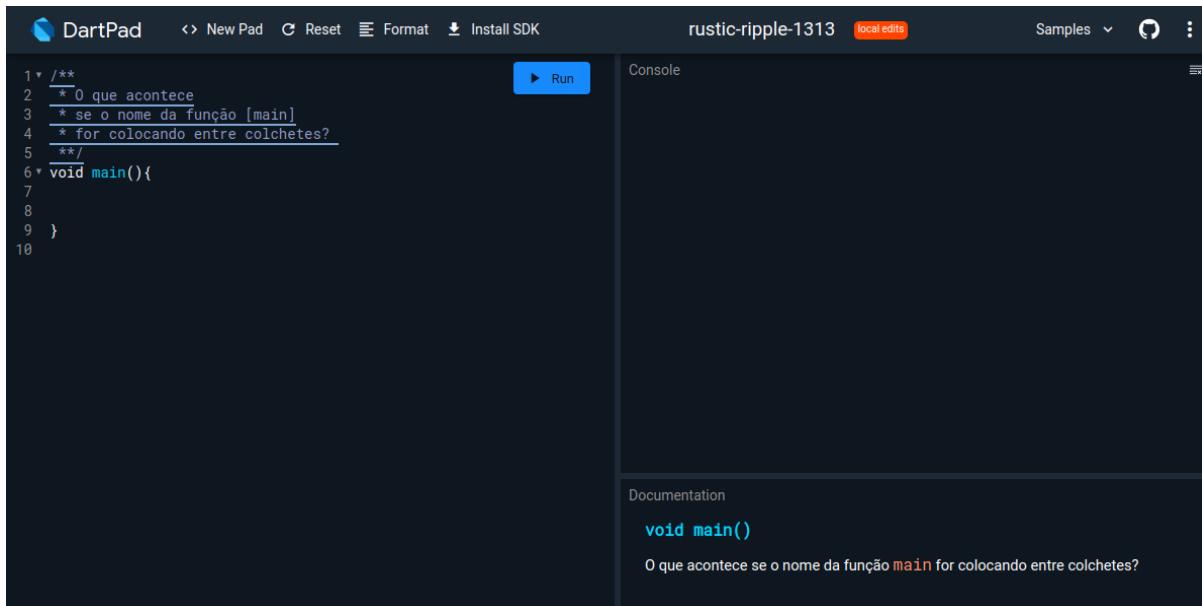
A 'Run' button is visible above the code. The 'Documentation' panel on the right shows:

```

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

```

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



The screenshot shows the DartPad interface with the following code in the editor:

```

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

```

A 'Run' button is visible above the code. The 'Documentation' panel on the right shows:

```

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

```

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

2.6 Tipos de dados, variáveis e inicialização

Os tipos de dados em Dart são os seguintes.

Em Dart

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

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

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

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

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

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

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

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

```

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

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

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

```

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

```

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

```

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

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

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

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

```

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

```

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

```

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

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

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

```

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

2.9 Operadores aritméticos

Vejamos os operadores aritméticos da linguagem Dart.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ou lógico: ||

E lógico: &&

Negação: !

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

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

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

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

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

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

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

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

Veja os exemplos a seguir.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

```

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

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

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

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

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

Veja algumas variações da estrutura if/else.

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

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

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

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

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

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

```

    }
}
```

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

```

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

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

```

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

Também é possível manipular valores reais.

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

E até listas.

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

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

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

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

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

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

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

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

Exercícios

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Referências

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