

DART

O DART é uma linguagem de programação orientada a objetos multiparadigma e sua primeira aparição foi na conferência GOTO na Dinamarca em Outubro de 2011 e tem como criadores o Lars Bak e Kasper Lund e veio com a premissa de substituir o JavaScript. Em fevereiro de 2018 a linguagem sofreu um reboot, onde era lançado o DART 2, otimizado para desenvolvimento no Client-Side, Web e Mobile.

Pode ser usada para desenvolvimento de:

- Aplicativos mobile;
- Desktop;
- Criação de scripts;
- Back-end.

Exemplos de uso da linguagem DART:

- Google AdSense (serviço de publicidade oferecido pelo Google);
- Google Ads (anteriormente chamada de AdWords e é o principal serviço de publicidade da Google e uma das principais fontes de receita desta empresa).

Link de referência (em inglês): <https://dart.dev/>

Documentação (em inglês): <https://dart.dev/guides>

INSTALAÇÃO DO DART

Atualmente, para instalar o DART no Windows 10, será necessário o uso do Chocolatey, um gerenciador de pacotes para Windows. Ele foi projetado para ser uma estrutura descentralizada para instalar pacotes (aplicativos/ferramentas) mais rapidamente. É muito semelhante ao que temos no mundo Linux, citando os mais famosos (apt e yum).

Instalação do Chocolatey

1º. Abra o PowerShell com direitos administrativos e insira a seguinte linha:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;  
[System.Net.ServicePointManager]::SecurityProtocol =  
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-  
Object  
System.Net.WebClient).DownloadString('https://community.chocolatey.or  
g/install.ps1'))
```

Referência: <https://chocolatey.org/install>

Para instalar o Dart SDK

Importante: Esses comandos requerem direitos de administrador.

Abra o Prompt de Comando com direitos administrativos.

```
C:\> choco install dart-sdk
```

Para atualizar o Dart SDK:

```
C:\> choco upgrade dart-sdk
```

OBS: Caso deseje instalar no ambiente Linux, visite o site abaixo Linux.

Referência: <https://dart.dev/get-dart>

Ambiente Online

Para quem utiliza o Windows 7 ou não possui permissões administrativas no computador, pode utilizar o compilador online desenvolvido pela equipe do DART. As sintaxes encontradas neste material funcionam perfeitamente nesta ferramenta online. Segue link abaixo:

Ambiente online: <https://dartpad.dev/>

SINTAXE E COMANDOS NO DART

Antes de começarmos a mostrar os comandos, vale a pena deixar algumas observações:

- A sintaxe é C-like, portanto, se você programa em Java, C#, PHP ou Javascript, você não terá dificuldades em aprender a linguagem;
- Segue o paradigma orientado a objetos;
- Todo o bloco de códigos termina com { };
- Indentação feita com o TAB;
- Toda linha de código termina com ponto-e-vírgula (;) como na maioria das linguagens;
- O Dart é *case sensitive*;
- Para fazer comentários usa-se // (barra dupla) para comentar uma linha e/ou /* */ (barra-asterisco-asterisco-barra) para blocos de códigos

Função Principal (Main)

A função **main()** serve como o ponto de partida para a execução do programa. Em geral, ela controla a execução direcionando as chamadas para outras funções no programa. Normalmente, um programa para de ser executado no final de **main**, embora possa terminar em outros pontos no programa por diversos motivos .

Sintaxe:

```
main( ){
```

```
Bloco de códigos do programa
```

```
}
```

Saída (Output):

É todo o resultado gerado pelo programa impresso na tela do computador.

Variáveis

São declaradas com o comando **var** seguido pelo nome desta. Caso deseje atribuir um valor inicial digite-o após o sinal de =.

Sintaxe:

var <nome_variavel>

var <nome_variavel> = valor atribuído

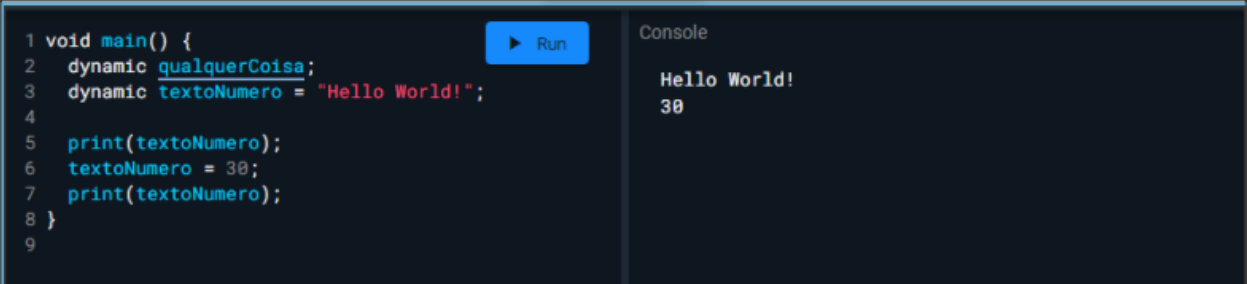
```
1 void main() {  
2   var nome;  
3   var sobrenome = "Silva";  
4   var idade = 20;  
5   var condicao = true;  
6 }  
7
```

Se você declarar uma variável com o comando **var** e atribuir um valor a ela, automaticamente o DART não permitirá que outro tipo de dado seja inserido posteriormente. Por exemplo, se for criada uma variável **var total** que receberá um valor do tipo inteiro, ela não poderá receber valores que não sejam mais do tipo inteiro após sua primeira atribuição (como *string* ou *bool*).

Para que isso seja possível, podemos usar o tipo **Dynamic** que permite que uma variável receba qualquer tipo de dados mesmo após sua primeira atribuição.

Sintaxe:

dynamic <nome_variavel>



The screenshot shows a code editor with the following Dart code:

```
1 void main() {  
2   dynamic qualquerCoisa;  
3   dynamic textoNumero = "Hello World!";  
4  
5   print(textoNumero);  
6   textoNumero = 30;  
7   print(textoNumero);  
8 }  
9
```

Next to the code is a blue "Run" button. To the right is a "Console" window showing the output:

```
Hello World!  
30
```

Notem que iniciei a variável **textoNumero** com uma **string** e, durante o desenvolvimento do código atribui um valor *inteiro*, e, na execução do código, ele não deu erro.

Tipagem

Serve para declarar o tipo de variável que trabalharemos, para facilitar o desenvolvimento do programa.

Tipos: **String**, **int**, **double**, **bool**, entre outros.

Exemplos de Sintaxe:

String nome = "Loja do Salim";

int funcionarios = 20;

double preco_produto = 2.45;

bool ainda_tem = true // true - verdadeiro / false – falso

```
1 void main() {  
2   String nome = "Loja do Salim";  
3   int funcionarios = 20;  
4   double preco_farinha = 2.45;  
5   bool ainda_tem = true  
6 }  
7
```

Um exemplo do uso do bool é a seguinte forma, a variável **info** recebe o valor de **aprovado** e verifica qual o valor recebido pelo atributo **?** e resulta o valor dentro dos argumentos dentro das aspas.

Se o resultado for **true** ele escreve **"Aprovado!!!"** e se for **false** ele escreve **"Reprovado!!!"**

bool aprovado = true;

String info;

info = aprovado ? "Aprovado!!!" : "Reprovado!";

Os tipos **List** e **Map** serão abordados futuramente em tópicos dedicados.

Operador ?? (interrogação dupla)

Serve para tratar se o valor de uma variável é nula ou se possui valor. Por exemplo, se a variável **nome** tiver um valor, será atribuído para a variável **info2**, mas se não tiver algum valor será impresso a frase **"Não informado"**.

Sintaxe:

```
String nome;  
String info 2 = nome ?? "Não informado";  
print(info2);
```

Concatenação

Serve para concatenar duas ou mais variáveis de texto, usa-se o sinal de **+** (mais).

Exemplo:

```
String nome = "Carla";  
print("O nome é: " + nome);
```

OBS: Pode-se usar o **\$** (cifrão) também para referenciar alguma variável sem a necessidade do operador **+**, como por exemplo:

```
print("O nome é: $nome");
```

Operadores

Permite realizar operações aritméticas e lógicas entre as variáveis.

- **Aritméticos:** Utilizados para realizar cálculos entre as variáveis. Os operadores mais comuns são:

+	Soma
-	Subtração
-expr.	Inversão (Reverte os sinais de uma expressão)
*	Multiplicação
/	Divisão
~/	Divisão que retorna apenas a parte inteira do resultado
%	Módulo (resto de divisão)

- **<variavel> += 15** - soma o valor da variável já atribuído anteriormente com o valor de 15. Podemos substituir o + pelos outros operadores aritméticos.
 - **<variavel> ++** - soma uma unidade ao valor final.
 - **<variavel> --** - subtrai uma unidade ao valor final.
- **Lógicos:** Permite comparar uma ou mais condições entre as variáveis.

```
// Comparadores
// -----
// >   maior
// >=  maior ou igual
// <   menor
// <=  menor ou igual
// ==  igual
// !=  diferente
```

- **Operador OR:** usa-se o símbolo || para comparar duas ou mais expressões.

Sintaxe: **bool testeOr = (false || false);**

```
// Operador OR
// -----
// true   true   -> true
// true   false  -> true
// false  true   -> true
// false  false  -> false
```

- **Operador AND:** usa-se o símbolo && para comparar duas ou mais expressões.

Sintaxe: **bool testeAnd = (false && true);**

```
// Operador AND
// -----
// true   true   -> true
// true   false  -> false
// false  true   -> false
// false  false  -> false
```

- **Operador NOT:** usa-se o símbolo ! para comparar a expressão.
Sintaxe: ***bool testeNOT = !true;***

Estruturas Condicionais

Possibilita a escolha de um grupo de ações e estruturas a serem executadas quando determinadas condições são ou não satisfeitas. Podem ser **Simples** ou **Composta**.

Condição IF

Utilizada em estruturas condicionais simples ou composta ela executa um comando ou vários comandos se a condição for verdadeira.

Na **Estrutura Condicional Simples** se a condição for falsa, a estrutura é finalizada sem executar os comandos.

A **Estrutura Condicional Composta** segue o mesmo princípio da Estrutura Condicional Simples, com a diferença de que quando a condição não é satisfeita, será executado o outro comando.

Sintaxe Simples:

```
if (condição){  
bloco de códigos se verdadeiro  
} else {  
bloco de códigos se falso  
}
```

Sintaxe Composta:

```
if (condição){  
bloco de códigos  
} else if{  
bloco de códigos  
} else if{  
bloco de códigos  
} else {  
bloco de códigos  
}
```


Condição Switch

É muito utilizada, principalmente para uso em estruturas de menu.

O conteúdo de uma variável é comparado com um valor constante, e caso a comparação seja verdadeira, um determinado comando é executado.

Sintaxe:

```
String linguagem = "Dart";  
switch(linguagem) {  
case "Dart":  
print("Dart!!!");  
break;  
case "Java":  
print("Java!!!");  
break;  
case "Swift":  
print("Swif!!!");  
break;  
default:  
print("Outra linguagem");  
}
```

Entrada (Input)

Para receber dados digitados via console pelo usuário, é necessário importar uma biblioteca já desenvolvida pelos desenvolvedores do DART. Para isso é necessário importar acima da função **main ()** a seguinte biblioteca **import 'dart:io';**

Assim sendo, temos que utilizar o comando **stdin.readLineSync()** para capturar o que o usuário deseja atribuir a determinada variável.

Exemplo: **var nome = stdin.readLineSync();**

Para converter em **int** ou **double**, por exemplo, temos que usar o comando **parse**.

Exemplo: **var idade = int.parse(stdin.readLineSync());**

```
var altura = int.double(stdin.readLineSync());
```

Funções

Servem para facilitar a programação através de uma programação modular, onde se cria funções para determinadas atividades dentro de um código.

Existem basicamente 3 tipos:

- sem retorno e sem parâmetros:

Sintaxe: **nomeFuncao(){**

bloco de códigos

}

- sem retorno e com parâmetros

Sintaxe: **nomeFuncao(parametro){**

bloco de códigos

}

- com retorno e com parâmetros

Sintaxe: **tipo_retorno nomeFuncao(tipo_variavel <nome_variavel>){**

bloco de códigos

}

Outra forma de representar uma função com retorno e com parâmetros é a seguinte:

<tipo_retorno> nomeFuncao(tipo_parametro <nome_parametro>) => linha de código;

onde o **arrow (=>)** serve como um return para essa função, auxiliando na construção de uma função mais simples e curta.

- **Funções com Parâmetros Opcionais:** podemos utilizar parâmetros opcionais nos nossos códigos, para isso inserimos chaves entre os parâmetros que desejamos ter essa configuração.

Exemplo:

```
<tipo_retorno>    nomeFuncao(<tipo_par><nome_par>,    {<tipo_par>  
<nome_par>, <tipo_par> <nome_par> }) {
```

Bloco de Códigos

```
}
```

OBS: Se eu desejar passar parâmetros padrões para as variáveis da função, eu uso o ?? na frente de uma declaração. por exemplo:

```
void criarBotao (String texto, {String cor, double largura}) {
```

```
print(texto);
```

```
print(cor ?? "Preto")
```

```
print(largura :: 10.0);
```

```
}
```

Loops

Permitem repetir um bloco de códigos até que sua condição seja atendida.

Temos, basicamente, dois tipos:

- **For:** repete uma condição até que a variável de controle (conhecida como ponteiro) atinja o valor que valide a condição para encerrar o loop.

Sintaxe:

```
for (int <var_controle>; <condição>; <var_controle> ++){
```

```
bloco de códigos
```

```
}
```

- **While:** repete uma condição até que ela seja atendida. Não necessita de um incremento para limitar o loop.

Sintaxe:

```
while (condição) {
```

```
Bloco de códigos
```

```
Condição = false
```

```
}
```

- **Do..While:** repete uma condição até que ela seja atendida. Não necessita de um incremento para limitar o loop.

Sintaxe:

```
do {  
Bloco de códigos  
} while (condição)
```

Listas (Lists)

Cria uma lista de dados de mesmo tipo de variável. Também conhecido como **vetor** e **array** em outras linguagens.

Sintaxe:

```
List<String> nome = [ ];  
List<int> idade = [ ];
```

Alguns comandos existentes na lista:

nome.add() = adiciona um dado no final da lista.

nome.length = mostra a quantidade (tamanho) da lista.

nome.remove("valor_remove") = remove o dado escolhido.

nome.removeAt(indice) = remove o valor do índice informado.

nome.first() = retorna o primeiro elemento da lista.

nome.last() = retorna o último elemento da lista.

nome.isEmpty() = retorna *true* se a lista estiver vazia; caso contrário, retorna *false*.

nome.contains("valor") = busca se existe o valor dentro da lista e, caso positivo retorna *true*, senão retorna *false*.

OBS: Lembrando que o índice de uma lista inicia com 0. Uma lista com 3 elementos terá índice de 0 a 2.

Mapas (Maps)

São coleções de dados organizados em um formato chave-valor. Cada elemento inserido em um mapa no DART possui uma chave a ele relacionado. Os mapas são estruturas muito úteis quando precisamos relacionar cada elemento com um identificador único.

Sintaxe: **Map<String, dynamic>cadastros = {**
"chave" : "valor",
"chave2" : "valor2",
"chave3": valor3,
"chaveN" : valorN,
};

Exemplo:

```
main() {  
    Map<String, dynamic> dados = {  
        "Nome": "Pedro Gabriel",  
        "idade": 45,  
        "Cidade": "Araçatuba",  
        "Estado": "São Paulo",  
        "Peso": 78.5,  
    };  
}
```

Alguns comandos existentes em mapas:

print(dados) = exibe todos os dados do mapa.

print(dados['idade']) = retorna o valor associado a chave 'idade'.

dados.addAll(novosDados) = adiciona os valores que a variável novosDados possui (var novosDados = {'nome' : 'João', 'sobrenome' : 'Silva'};)

dados.remove('Estado') = remove o valor referente a chave informada entre parenteses.

dados.clear() = limpa todos os dados do mapa.

Comando para limpar a tela

Um comando interessante para realizar a limpeza da tela, que pode ser usado em momentos em que se repetira uma ação, como um novo cadastro e necessita limpar a tela do console.

Sintaxe: **print("\x1B[2J\x1B[0;0H").**

PROGRAMAÇÃO ORIENTADA A OBJETOS

Classe (Class)

Em orientação a objetos, uma **classe** é uma descrição que abstrai um conjunto de objetos com características similares. Mais formalmente, é um conceito que encapsula abstrações de dados e procedimentos que descrevem o conteúdo e o comportamento de entidades do mundo real, representadas por objetos. É composta de atributos (variáveis) e métodos (funções).

Sintaxe:

```
Class <NomeClasse> {  
  
    Tipo <atributo A>;          //atributo  
    Tipo <atributo B>;          //atributo  
  
    void <nome_metodoA>( ){      //método  
        comandos;  
    }  
    void <nome_metodoB>( ){      //método  
        comandos;  
    }  
}
```

Para instanciar um objeto da classe podemos fazer o seguinte:

Pessoa pessoa1 = new Pessoa (); //a palavra **new** é opcional.

Pessoa pessoa1 = Pessoa ();

Exemplo:

```
1 class Pessoa {
2     //Atributos da Classe
3     String nome;
4     int idade;
5     double altura;
6     //Métodos da Classe
7     void dormir(){
8         print("0 $nome está dormindo");
9     }
10
11     void trabalhar(){
12         print("0 $nome está trabalhando");
13     }
14 }
15
16 void main() {
17     //Instanciando um objeto
18     Pessoa pessoa1 = Pessoa();
19     //Atribuindo valores
20     pessoa1.nome = "João";
21     pessoa1.idade = 20;
22     pessoa1.altura = 1.73;
23     //Imprimindo na tela os valores
24     print(pessoa1.nome);
25     print(pessoa1.altura);
26     //Chamando os métodos
27     pessoa1.dormir();
28     pessoa1.trabalhar();
29 }
30
```

Run

Console

```
João
1.73
0 João está dormindo
0 João está trabalhando
```

Documentation

Construtores

São utilizados basicamente para inicializar as variáveis de uma classe e é obrigatório para toda a classe. Ou seja, sem eles seria obrigatório passar valores para os atributos de uma classe ao inicializa-la (cria-la). No DART ele deve levar o mesmo nome da classe. Quando não é criado um construtor o DART cria implicitamente um construtor vazio. Ou seja, na hora que vamos instanciar um objeto através de **Pessoa pessoa1 = Pessoa();** a função destacada em vermelho já é o construtor implícito.

Há duas formas de criar um construtor, conforme exemplos abaixo:

Exemplo 01:

```
Pessoa(String nome, int idade, double altura) {  
    this.nome = nome;  
    this.idade = idade;  
    this.altura = altura;  
}
```

Exemplo 02:

```
Pessoa(this.nome, this.idade, this.altura);
```

```
1  class Pessoa {  
2      //Atributos da Classe  
3      String nome;  
4      int idade;  
5      double altura;  
6      //Métodos da Classe  
7      void dormir(){  
8          print("O $nome está dormindo");  
9      }  
10  
11     void trabalhar(){  
12         print("O $nome está trabalhando");  
13     }  
14     //Construtores  
15     Pessoa(this.nome, this.idade, this.altura);  
16 }  
17  
18 void main() {  
19     //Instanciando um objeto através do Construtor  
20     Pessoa pessoa1 = Pessoa("João", 20, 1.73);  
21     //Imprimindo na tela os valores  
22     print(pessoa1.nome);  
23     print(pessoa1.altura);  
24     //Chamando os métodos  
25     pessoa1.dormir();  
26     pessoa1.trabalhar();  
27 }  
28
```


Variáveis locais (ou privadas)

Para proteger algumas variáveis de sofrerem alterações fora da classe, temos que determinar que sejam variáveis locais, ou privadas. Para fazer essa alteração, é só inserir o "**underline**" () na frente da variável criada.

Por exemplo:

```
class Pessoa {  
    String nome; -> Variável global, pode ser acessada pela classe MAIN.  
    _int idade; -> Variável local, só pode ser processada pela Classe.  
    double altura; -> Variável global, pode ser acessada pela classe MAIN.  
}
```

Final

Essa propriedade determina que o valor da variável não pode ser alterado depois de atribuída uma única vez.

Sintaxe:

```
final <tipo> <variavel> = valor;
```

Herança

Herança é um princípio da Orientação a Objetos que permite que características comuns a diversas classes sejam herdadas de uma **classe base**, ou **superclasse**. Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos. Por exemplo, uma classe **Trabalhador () { }** pode herdar atributos de uma classe **Pessoa () { }**, pois um trabalhador é uma pessoa.

Sintaxe:

```
class <Nome> extends <SuperClasse> {  
    <atributos>  
    <métodos>  
    <construtor>  
}
```

Exemplo:

```
1 class Pessoa {
2   //Atributos da Classe Pessoa
3   String nome;
4   int idade;
5   double altura;
6   //Construtor
7   Pessoa(this.nome, this.idade, this.altura);
8 }
9
10 class Trabalhador extends Pessoa {
11   //Atributos da Classe Trabalhador
12   String cargo;
13   int matricula;
14   //Construtor
15   Trabalhador(String nome, int idade, double altura, this.cargo, this.matricula)
16     : super(nome, idade, altura);
17 }
18
19 void main() {
20   Trabalhador funcionario = Trabalhador("João", 33, 1.68, "Engenheiro", 2319);
21   print(funcionario.nome);
22   print(funcionario.idade);
23   print(funcionario.cargo);
24   print(funcionario.matricula);
25 }
```

Note que a classe **Trabalhador** está herdando os atributos da classe **Pessoa** (*nome, idade e altura*) e possui seus próprios atributos (*cargo e matrícula*). Quando criamos o construtor na classe **Trabalhador**, usamos o comando *:super(nome, idade, altura)* para indicar que os primeiros atributos (*String nome, int idade e double altura*) são herdados da classe Pessoa.

Além dos atributos, todos os métodos dentro da Superclasse serão herdados.

NULL SAFETY

Ferramenta inclusa no update do DART 2.0 que ajuda a prever alguns erros em variáveis null. Com o Null Safety, o DART é capaz de te ajudar a identificar o que pode ser nulo ou não, o que é necessário ter um valor ou não, e com isso evitar crashes da sua aplicação, além de evitar a verificação manual se a variável é nula ou não.

Essa ferramenta utiliza várias formas de auxiliar no tratamento de erros quando um valor não é especificado em uma variável declarada. Podemos usar o `?` na frente da tipagem, como por exemplo ***String? nome***, onde indica-se que essa variável pode ser nula. Também podemos usar o atributo ***late*** que indica que, mais tarde ou a qualquer momento, essa variável receberá um valor ao ser inicializada. Por exemplo: ***late String nome***.

- Erros em tempo de desenvolvimento;
- Facilita o tempo de desenvolvimento;
- Determinar para o compilador se a variável pode ser nula;
- A interrogação que se coloca depois da variável dá a possibilidade dela ser nula. Ou seja, essa variável pode receber um valor nulo.
- Existe alguns casos que você quer assumir o risco, então se usa o `!` (exclamação) para garantir que você está assumindo que aquele valor não será nulo.
- `Late` serve para criar uma variável sem valor algum, mas que você tem certeza que será inicializada antes de ser utilizada. Nunca poderá atribuir o valor nulo para o `late`.

Referências: ateliware.com/blog/null-safety-em-dart

ateliware.com/blog/flutter-2-0

<https://www.youtube.com/watch?v=COHLvdZT0cw&t=27s>