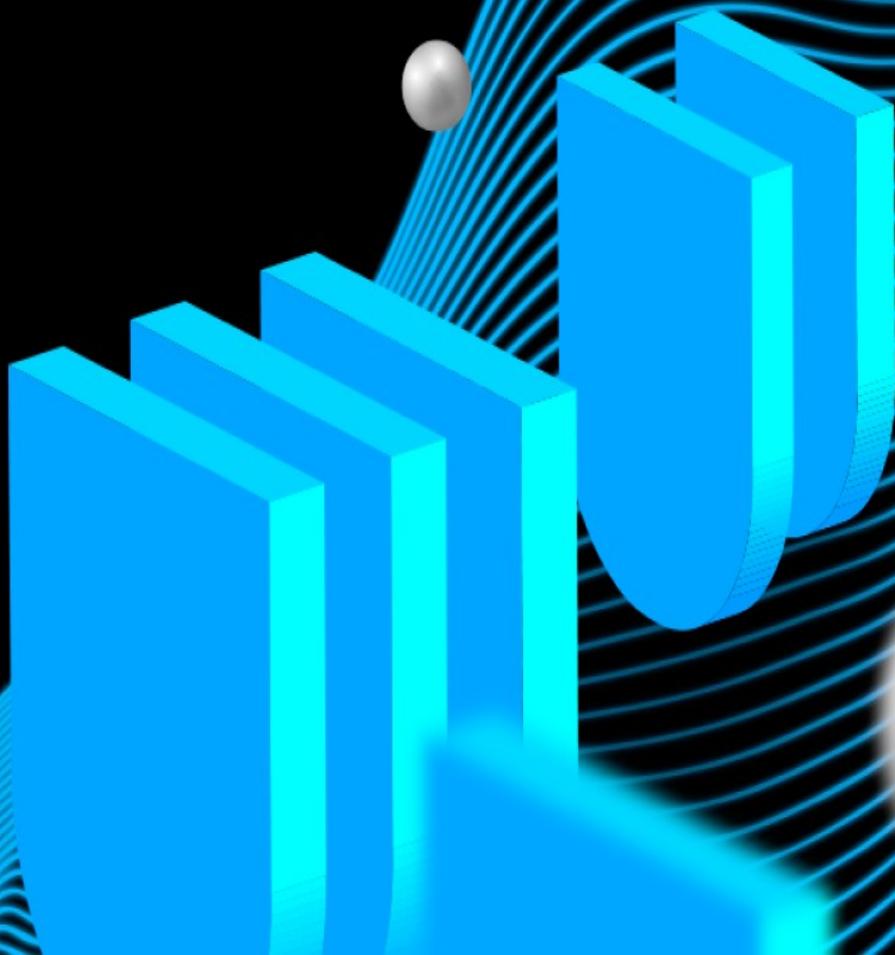




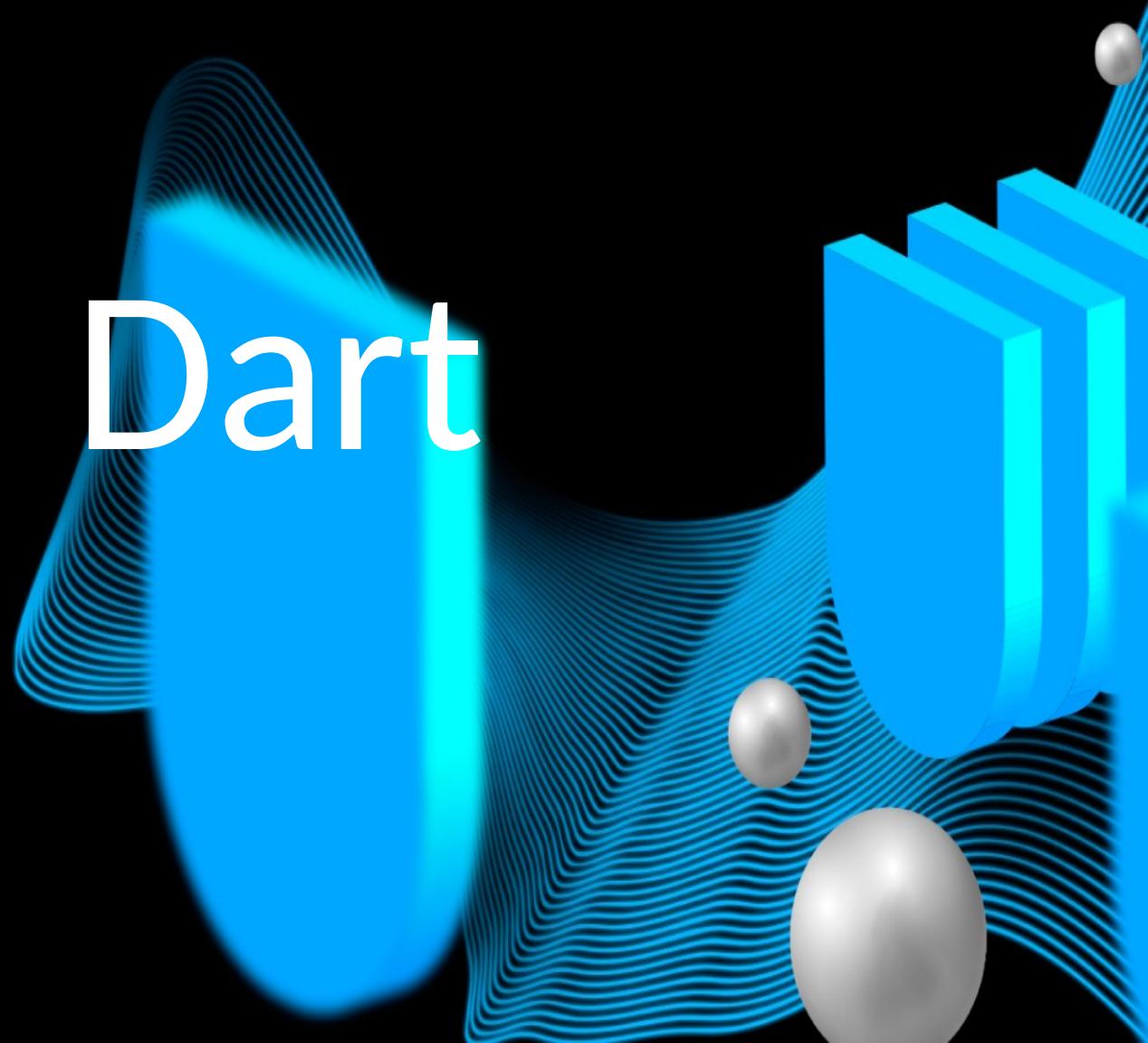
UNITAU

Universidade de Taubaté





Dart



Dart



Dart é uma linguagem de programação criada pelo Google em 2011.

O SDK da linguagem Dart tem ferramentas que permitem a execução do código Dart em diferentes plataformas. O SDK pode ser obtido em:

<https://dart.dev/tools/sdk>

A partir da versão 1.21 do Flutter, não é mais necessário instalar o SDK do Dart individualmente pois o arquivo de instalação do Flutter já contém o Dart.



Execução em Browsers

Para execução em browsers, o código Dart pode ser transformado em Javascript.

Para compilar o código para JavaScript temos o comando dart2js.

Para testar a execução de código Dart em um browser, podemos utilizar o site

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

Interpretador



O SDK também inclui uma máquina virtual para interpretar o código Dart.

helloworld.dart:

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

```
> dart helloworld.dart  
Hello World
```

Tipos de Dados



A linguagem Dart possui vários tipos de dados pré-definidos, os mais comuns são:

int	inteiro
double	número decimal
num	Inteiro ou número decimal
bool	booleano
String	string imutável
StringBuffer	string mutável
RegExp	expressão regular
List, Map, Set	coleções
DateTime	data e hora
Duration	intervalo de tempo
Uri	identificador de recurso uniformizado
Error	informações sobre erro

Declaração de Variáveis



Variáveis armazenam referências para os valores atribuídos.

Para permitir variáveis nulas, é obrigatório explicitar na declaração da variável a permissão para a variável ser nula colocando o operador ? em frente ao tipo da variável.

null.dart:

```
void main() {  
    String? s;  
    if ( s == null )  
        print("Nulo");  
    s = "Hello World";  
    if ( s is String )  
        print("String");  
}
```

```
> dart null.dart  
Nulo  
String
```

Tipos Estáticos



Variáveis podem ser declaradas com tipo de dados estático ou dinâmico.

Variáveis com tipo de dados estático são verificadas em tempo de compilação. Embora muitos desenvolvedores prefiram variáveis com tipo dinâmico ao invés de tipo estático por considerar que o desenvolvimento fica mais rápido, variáveis com tipo estático facilitam a criar um código mais robusto.

estatico.dart:

```
void main() {  
    String s = "Hello World";  
    print(s);  
}
```

Tipos Dinâmicos



Variáveis com tipo de dados dinâmico são verificadas em tempo de execução.

Uma variável com tipo dinâmico em Dart pode ser declarada com o tipo:

var - cria uma variável sem um tipo específico

dynamic - cria uma variável com o tipo “**dynamic**”

Variáveis do tipo **var e **dynamic** não são a mesma coisa e tem comportamento diferentes em algumas situações.**

Tipos Dinâmicos



Variáveis do tipo **dynamic** podem mudar o tipo de valor atribuído para a variável.

dynamic.dart:

```
void main() {  
    dynamic x = 'teste';  
    if ( x is String )  
        print("String");  
    x = 1;  
    if ( x is int )  
        print("Inteiro");  
}
```

```
> dart dynamic.dart  
String  
Inteiro
```

Tipos Dinâmicos



Variáveis do tipo **var** não podem mudar o tipo de valor atribuído para a variável.

var.dart:

```
void main() {  
    var x = 'teste';  
    if ( x is String )  
        print("String");  
    x = 1;  
    if ( x is int )  
        print("Inteiro");  
}
```

> dart var.dart

var.dart:5:7: Error: A value of type 'int' can't be assigned to a variable of type 'String'.
x = 1;



Tipos Dinâmicos

É permitido funções com retorno do tipo **dynamic**.

dynamicfunction.dart:

```
void main() {  
  print(multiplyMethod(2,4));  
}
```

```
dynamic multiplyMethod( int a, int b ) {  
  return( a * b );  
}
```

```
> dart dynamicfunction.dart
```

```
8
```

Tipos Dinâmicos



Não é permitido funções com retorno do tipo var.

varfunction.dart:

```
void main() {  
    print(multiplyMethod(2,4));  
}
```

```
var multiplyMethod( int a, int b ) {  
    return( a * b );  
}
```

> dart varfunction.dart

**varfunction.dart:5:1: Error: The return type can't be 'var'.
Try removing the keyword 'var', or replacing it with the
name of the return type.**

```
var multiplyMethod( int a, int b ) {  
    ^^^
```

Constantes



Variáveis declaradas como **final** ou **const** não podem ter seu valor alterado.

final.dart:

```
void main() {  
    final int x = 1;  
    print(x);  
    x = 2;  
    print(x);  
}
```

> dart final.dart

final.dart:4:3: Error: Can't assign to the final variable
'x'.

```
    x = 2;  
    ^
```

Constantes



final deve ser usado para variáveis com valor atribuído em tempo de execução.

const deve ser usado para variáveis com valor atribuído em tempo de compilação.

Strings



Strings podem ser declaradas com aspas simples ou aspas duplas.

quote.dart:

```
void main() {  
    print('Single quote');  
    print("Double quote");  
}
```

```
> dart quote.dart  
Single quote  
Double quote
```

Strings



Strings com múltiplas linhas podem ser declaradas com aspas triplas.

multiline.dart:

```
void main() {  
    String s = '''  
    Primeira linha  
    Segunda linha  
    Terceira Linha  
''';  
    print(s);  
}
```

```
> dart multiline.dart  
Primeira linha  
Segunda linha  
Terceira Linha
```

Strings



A concatenação de strings pode ser feita com o operador **+** ou apenas com a colocação das strings em sequência.

concatenacao.dart:

```
void main() {  
  print('Hello ' + 'world');  
  print('Hello ' 'world');  
}
```

> dart concatenacao.dart

```
Hello world  
Hello world
```

Strings



A interpolação de texto em uma string pode ser feira pelo operador \$ ou \${expressao}.

interpolacao.dart:

```
void main() {  
    String nome = "Ana";  
    int idade = 20;  
    print('Nome $nome, idade ${idade}anos');  
}
```

> dart interpolacao.dart

Nome Ana, idade 20anos

Strings



Strings são imutáveis.

identical.dart:

```
void main() {  
    String s1 = "Hello World";  
    String s2 = "Hello World";  
    if ( identical( s1, s2 ) )  
        print("As strings sao o mesmo objeto");  
    StringBuffer sb1 = StringBuffer("Hello World");  
    StringBuffer sb2 = StringBuffer("Hello World");  
    if ( identical( sb1, sb2 ) )  
        print("Os buffers sao o mesmo objeto");  
}
```

> dart identical.dart

As strings sao o mesmo objeto

Strings



Alterar o valor de um `String` implica na criação de um novo objeto. A performance da alteração de um `StringBuffer` é melhor do que a alteração de um `String`,

stringperformance.dart:

```
void main() {
  Stopwatch sw1 = Stopwatch();
  sw1.start();
  StringBuffer sb = StringBuffer();
  for (int i = 0; i < 10000; i++)
    sb.write("Teste");
  print("Com StringBuffer: "
+sw1.elapsedMilliseconds.toString());
  Stopwatch sw2 = Stopwatch();
  sw2.start();
  String s = '';
  for (int i = 0; i < 10000; i++)
    s += "Teste";
  print("Com String: " +sw2.elapsedMilliseconds.toString());
}
```

Strings



```
> dart stringperformance.dart  
Com StringBuffer: 4  
Com String: 46
```

Strings



String podem conter caracteres de escape, utilizando a contrabarra “\”, porém, se a string for prefixada com r, os caracteres de escape serão ignorados.

escape.dart:

```
void main() {  
    print("Uma\nvariavel\ntexto");  
    print("Uma\\nvariavel\\ntexto");  
    print(r"Uma\nvariavel\ntexto");  
}
```

> dart escape.dart

```
Uma  
variavel  
texto  
Uma\nvariavel\ntexto  
Uma\nvariavel\ntexto
```

Operadores



Além dos operadores mais comuns, a linguagem Dart possui o operador ternário <condicao>?<valor1>:<valor2>.

ternario.dart:

```
void main() {  
    print("4 e ${4%2==0?"par":"impar"}");  
    print("5 e ${5%2==0?"par":"impar"}");  
}
```

```
> dart ternario.dart  
4 e par  
5 e impar
```

Operadores



A linguagem Dart também possui um operador que atribui um valor apenas se a variável tem valor nulo ??=.

setnull.dart:

```
void main() {  
    int? x = 3;  
    int? y;  
    x ??= 4;  
    y ??= 4;  
    print("x = ${x}");  
    print("y = ${y}");  
}
```

> dart setnull.dart

setnull.dart:4:3: Warning: Operand of null-aware operation
'??=' has type 'int' which excludes null.

```
x ??= 4;
```

^

```
x = 3
```

```
y = 4
```

Bibliotecas



A linguagem Dart possui várias bibliotecas que fazem parte do pacote principal da linguagem. Essas bibliotecas fornecem objetos e funções para diversos fins como suporte a coleções (**dart:collection**), cálculos matemáticos (**dart:math**), suporte a operações de I/O (**dart:io**) e outras.

A biblioteca básica (**dart:core**) é automaticamente importada nos programas, não sendo necessário importar essa biblioteca. As demais bibliotecas devem ser importadas para utilizar a biblioteca.

Além das bibliotecas fornecidas com a linguagem, existem também pacotes criados pela comunidade e o desenvolvedor também pode criar seus próprios pacotes que devem ser importados para serem utilizados.

Bibliotecas



```
import.dart:  
import 'dart:math';  
  
void main() {  
  print("5^3 = ${pow(5,3)}");  
}  
  
> dart import.dart  
5^3 = 125
```

Funções



A linguagem Dart permite funções com parâmetros posicionais como a maioria das linguagens.

posicional.dart:

```
import 'dart:math';

int potencia( int base, int expoente ) {
  return( pow(base,expoente).toInt() );
}
```

```
void main() {
  print("5^3 = ${potencia(5,3)}");
}
```

```
> dart positional.dart
5^3 = 125
```

Funções



Também é possível definir parâmetros opcionais.

opcional.dart:

```
import 'dart:math';
```

```
int potencia( int base, [int? expoente] ) {  
  expoente ??= 2;  
  return( pow(base,expoente).toInt() );  
}
```

```
void main() {  
  print("5^3 = ${potencia(5,3)}");  
  print("5^2 = ${potencia(5)}");  
}
```

```
> dart opcional.dart
```

```
5^3 = 125  
5^2 = 25
```

Funções



Parâmetros podem ser nomeados, o que permite que o valor dos parâmetros sejam passados fora de ordem. Parâmetros nomeados são opcionais.

nomeado.dart:

```
import 'dart:math';

int potencia( {int? base, int? expoente} ) {
    base ??= 4;
    expoente ??= 2;
    return( pow(base,expoente).toInt() );
}

void main() {
    print("5^3 = ${potencia(expoente:3,base:5)}");
    print("5^2 = ${potencia(base:5)}");
    print("4^3 = ${potencia(expoente:3)}");
    print("4^2 = ${potencia()}");
}
```

Funções



```
> dart nomeado.dart
```

```
5^3 = 125
```

```
5^2 = 25
```

```
4^3 = 64
```

```
4^2 = 16
```



Funções

Pode ser especificado um valor default para parâmetros opcionais.

default.dart:

```
import 'dart:math';

int potencia( [int base=4, int expoente = 2] ) {
  return( pow(base,expoente).toInt() );
}

void main() {
  print("5^3  = ${potencia(5,3)}");
  print("5^2  = ${potencia(5)}");
  print("4^2  = ${potencia()}");
}
```

> dart default.dart

```
5^3  = 125
5^2  = 25
4^2  = 16
```

Funções Lambda



Funções com uma única linha de código podem ser escritas usando a notação lambda (arrow function).

lambda.dart:

```
import 'dart:math';

int pot( int b, int e ) => pow(b,e);
```

```
void main() {
  print("5^3 = ${pot(5,3)}");
}
```

```
> dart lambda.dart
5^3 = 125
```

Coleções



Coleções são objetos que permitem manipular um conjunto de elementos.

Na biblioteca básica da linguagem Dart (`dart:core`) são definidas as coleções `List`, `Map` e `Set`.

A biblioteca específica para coleções (`dart:collection`) define outros objetos para manipulação de coleções.

List



List é uma coleção indexada de elementos. A linguagem Dart não possui arrays, sendo usado **List** quando necessário.

list.dart:

```
void main() {  
    List contatos = ['Ana', 'Paula', 'Luis'];  
    contatos.forEach((nome) {  
        print(nome);  
    });  
    print("Elemento 2: ${contatos[2]}");  
}
```

> dart list.dart

```
Ana  
Paula  
Luis  
Elemento 2: Luis
```

Map



Map é uma coleção que armazena pares chave/valor e permitem recuperar o valor associado a uma chave.

map.dart:

```
void main() {  
  Map operadora = { 15: 'Vivo', 21: 'Claro', 41: 'Tim' };  
  print('Codigo 15 Operadora ${operadora[15]}');  
}
```

> dart map.dart

Codigo 15 Operadora Vivo

Set



Set é uma coleção de elementos que não permite repetição de elemento.

set.dart:

```
void main() {  
    Set contatos = Set();  
    contatos.add('Ana');  
    contatos.add('Luis');  
    contatos.add('Ana');  
    contatos.forEach((nome) {  
        print(nome);  
    });  
}
```

> dart set.dart

Ana
Luis

Estruturas de Controle



A linguagem Dart possui as estruturas de controle básicas comuns a maioria das linguagens estruturadas.

while



while.dart:

```
void main() {  
    int i = 1;  
    int fat = 1;  
    while(i<5) {  
        fat *= i;  
        print("${i}! = ${fat}");  
        i++;  
    }  
}
```

> dart while.dart

```
1! = 1  
2! = 2  
3! = 6  
4! = 24
```



do while

dowhile.dart:

```
void main() {  
    int i = 1;  
    int fat = 1;  
    do {  
        fat *= i;  
        print("${i}! = ${fat}");  
        i++;  
    } while(i<5);  
}
```

> dart dowhile.dart

```
1! = 1  
2! = 2  
3! = 6  
4! = 24
```



for

```
for.dart:  
void main() {  
    List<String> contatos = ['Ana', 'Luis', 'Paula'];  
    for( int i=0; i<contatos.length; i++ ) {  
        print(contatos[i]);  
    };  
}  
}
```

```
> dart for.dart
```

```
Ana  
Luis  
Paula
```

for in



O laço **for** tem uma variação para percorrer coleções.

forin.dart:

```
void main() {  
    List<String> contatos = ['Ana', 'Luis', 'Paula'];  
    for( String nome in contatos ) {  
        print(nome);  
    };  
}
```

> dart for.dart

```
Ana  
Luis  
Paula
```



Orientação a Objetos

A linguagem Dart é orientada a objetos e todas as classes descendem da classe **Object**.

É permitido declarar várias classes dentro do mesmo arquivo.

Construtores



Se não for definido nenhum construtor para uma classe, será criado um construtor default sem parâmetros. Para atribuir o valor de uma propriedade do objeto no construtor pode ser usado **this.<nome da variável>** como parâmetro do construtor.

construtor.dart:

```
class Contato {  
    String nome;  
    String sobrenome;  
    Contato(this.nome, this.sobrenome);  
}
```

```
void main() {  
    Contato c = Contato("Ana", "Martins");  
    print("Contato ${c.nome} ${c.sobrenome}");  
}
```

> dart construtor.dart
Contato Ana Martins



Construtores Nomeados

É possível criar construtores nomeados para uma classe.

construtor_nomeado.dart:

```
class Cor {  
    String valor = "";  
  
    Cor.Vermelho() { valor="vermelho"; }  
    Cor.Verde() { valor="verde"; }  
    Cor.Azul() { valor="azul"; }  
}  
  
void main() {  
    print("Cor ${Cor.Vermelho().valor}");  
    print("Cor ${Cor.Verde().valor}");  
    print("Cor ${Cor.Azul().valor}");  
}  
  
> dart construtornomeado.dart  
Cor vermelho  
Cor verde  
Cor azul
```

Getters / Setters



Getters e Setters são métodos para atribuir e obter valores das propriedades do objeto.

getset.dart:

```
class Cor {  
    String valor = "";  
  
    void set cor(String v) { this.valor = v; }  
    String get cor { return(valor); }  
}  
  
void main() {  
    Cor c = Cor();  
    c.cor = "Azul";  
    print("Cor ${c.cor}");  
}
```

```
> dart getset.dart  
Cor Azul
```

Membros Públicos/Privados



Por default, todos os métodos e propriedades são públicos. Para que um método ou propriedade seja privado, o nome deve começar com _.

Membros Púlicos



```
cor_publico.dart:  
class Cor {  
  String valor = "";  
}
```

```
publico.dart:  
import 'cor_publico.dart';
```

```
void main() {  
  Cor c = Cor();  
  c.valor = "Azul";  
  print("Cor ${c.valor}");  
}
```

```
> dart publico.dart  
Cor Azul
```

Membros Privados



cor_privado.dart:

```
class Cor {  
    String _valor = "";  
  
    void set cor(String v) { this._valor = v; }  
    String get cor { return(_valor); }  
}
```

privado_valido.dart:

```
import 'cor_privado.dart';  
  
void main() {  
    Cor c = Cor();  
    c.cor = "Azul";  
    print("Cor ${c.cor}");  
}
```

> dart privado_valido.dart

Cor Azul

Membros Privados



cor_privado.dart:

```
class Cor {  
    String _valor = "";  
  
    void set cor(String v) { this._valor = v; }  
    String get cor { return(_valor); }  
}
```

privado_invalido.dart:

```
import 'cor_privado.dart';  
  
void main() {  
    Cor c = Cor();  
    c._valor = "Azul";  
    print("Cor ${c._valor}");  
}
```



Membros Privados

```
> dart privado_invalido.dart
```

```
privado_invalido.dart:5:5: Error: The setter '_valor' isn't  
defined for the class 'Cor'.
```

```
- 'Cor' is from 'cor_privado.dart'.
```

```
Try correcting the name to the name of an existing setter,  
or defining a setter or field named '_valor'.
```

```
c._valor = "Azul";  
~~~~~
```

```
privado_invalido.dart:6:18: Error: The getter '_valor' isn't  
defined for the class 'Cor'.
```

```
- 'Cor' is from 'cor_privado.dart'.
```

```
Try correcting the name to the name of an existing getter,  
or defining a getter or field named '_valor'.
```

```
print("Cor ${c._valor}");  
~~~~~
```

Membros Privados



Funções e classes definidas no mesmo arquivo podem acessar membros privados da classe.

mesmoarquivo.dart:

```
class Cor {  
    String _valor = "";  
  
    void set cor(String v) { this._valor = v; }  
    String get cor { return(_valor); }  
}  
  
void main() {  
    Cor c = Cor();  
    c._valor = "Azul";  
    print("Cor ${c.cor}");  
}  
  
> dart mesmo_arquivo.dart  
Cor Azul
```



Operador cascata

O operador `..` permite acessar propriedades e/ou executar métodos de um objeto em cascata.

cascata.dart:

```
class Contato {  
    String nome = "";  
    String sobrenome = "";  
    void Show() {  
        print("Contato ${this.nome} ${this.sobrenome}");  
    }  
}  
  
void main() {  
    Contato c = Contato();  
    c..nome = "Ana"  
    ..sobrenome = "Martins"  
    ..Show();  
}
```

```
> dart cascata.dart  
Contato Ana Martins
```

Herança



A linguagem Dart utiliza a palavra **extends** para a herança entre as classes. Não é permitido herança múltipla, sendo necessário usar interfaces como em Java.

Os construtores não são herdados e para acionar o construtor da superclasse é necessário usar a chamada **super()** no construtor da subclasse.

Herança



heranca.dart:

```
class Contato {  
    String nome;  
    String sobrenome;  
  
    Contato(this.nome, this.sobrenome);  
}  
  
class Email extends Contato {  
    String email = "";  
  
    Email(String n, String s, String v) : super(n, s) {  
        this.email = v;  
    }  
}  
  
void main() {  
    Email e = Email("Ana", "Martins", "ana@email.com");  
    print("Contato ${e.nome} ${e.sobrenome}, ${e.email}");  
}
```

Herança



> **dart heranca.dart**

Contato Ana Martins, ana@email.com

Interface



A linguagem Dart não tem a declaração explícita de interface. Não se declara uma classe usando a palavra **interface**. Ao se declarar uma classe, automaticamente é criada uma interface dessa classe, portanto, qualquer classe pode ser usada como interface. Porém é obrigatório sobrescrever todos os métodos e propriedades na classe que implementar essa interface.

Interface



```
interface.dart:  
class Mamifero {  
    int vidas = 1;  
    String especie = "mamifero";  
  
    Mamifero([String? especie]) {  
        if ( especie != null )  
            this.especie = especie;  
    }  
  
    void quantasVidas() {  
        print("Um ${especie} tem ${vidas} vidas");  
    }  
    void falar() {  
        print("Nao fala");  
    }  
}
```

Interface



```
class Gato implements Mamifero {  
    int vidas = 7;  
    String especie ="";  
  
    void quantasVidas() {  
        print("Um gato tem ${vidas} vidas");  
    }  
    void falar() {  
        print("Ronronando");  
    }  
}  
  
class Cao extends Mamifero {  
    Cao() : super("cao") {}  
  
    void falar() {  
        print("Latindo");  
    }  
}
```

Interface



```
main() {  
    Mamifero m = Mamifero();  
    Gato g = Gato();  
    Cao c = Cao();  
    m.quantasVidas();  
    m.falar();  
    g.quantasVidas();  
    g.falar();  
    c.quantasVidas();  
    c.falar();  
}
```

```
> dart interface.dart  
Um mamifero tem 1 vidas  
Nao fala  
Um gato tem 7 vidas  
Ronronando  
Um cao tem 1 vidas  
Latindo
```

Classes Abstratas



Classes abstratas não podem ser instanciadas. Podem ser usadas para derivar uma subclasse ou implementar como interface.

Se usadas como interface, todos métodos e propriedades devem obrigatoriamente ser sobrescritos.

Se usadas para derivar uma subclasse, só é obrigatório sobrescrever os métodos abstratos.

Classes Abstratas



abstrata.dart:

```
abstract class Mamifero {  
    int vidas = 1;  
    String especie = "mamifero";  
  
    Mamifero([String? especie]) {  
        if ( especie != null )  
            this.especie = especie;  
    }  
  
    void quantasVidas() {  
        print("Um ${especie} tem ${vidas} vidas");  
    }  
    void falar();  
}
```



Classes Abstratas

```
class Gato implements Mamifero {  
    int vidas = 7;  
    String especie ="";  
  
    void quantasVidas() {  
        print("Um gato tem ${vidas} vidas");  
    }  
    void falar() {  
        print("Ronronando");  
    }  
}  
  
class Cao extends Mamifero {  
    Cao() : super("cao") {}  
  
    void falar() {  
        print("Latindo");  
    }  
}
```

Classes Abstratas



```
main() {  
    Gato g = Gato();  
    Cao c = Cao();  
    g.quantasVidas();  
    g.falar();  
    c.quantasVidas();  
    c.falar();  
}  
  
> dart abstrata.dart  
Um gato tem 7 vidas  
Ronronando  
Um mamifero tem 1 vidas  
Latindo
```

Mixins



Para permitir a criação de interfaces que permitem utilizar os métodos definidos na interface, a linguagem Dart tem os **mixins**.

Qualquer classe pode ser usada como um mixin. Porém classes usadas como mixins não podem ter construtor.

Para que uma classe implemente um mixin deve ser usada a palavra **with** no lugar de **implements**.

Se um mixin for declarado com a palavra **Mixin**, não poderá ser instanciado como classe.

Uma classe pode implementar várias interfaces ou mixins para obter o efeito da herança múltipla.

Mixins



mixin.dart:

```
mixin Mamifero {  
    String especie = "mamifero";  
    int vidas = 1;  
  
    void Make(String e) {  
        especie = e;  
    }  
  
    void setVidas(int v) {  
        vidas = v;  
    }  
  
    void quantasVidas() {  
        print("Um ${especie} tem ${vidas} vidas");  
    }  
}
```

Mixins



```
mixin Som {  
    String fala = "nao fala";  
  
    void Make(String s) {  
        fala = s;  
    }  
  
    void falar() {  
        print("${fala}");  
    }  
}
```

Mixins



```
class Gato with Som, Mamifero {  
    Gato() {  
        Make("Gato");  
        setVidas(7);  
    }  
}
```

```
class Cao with Mamifero, Som {  
    Cao() {  
        Make("latindo");  
    }  
}
```

Mixins



```
main() {  
  Gato g = Gato();  
  g.quantasVidas();  
  g.falar();  
  Cao c = Cao();  
  c.quantasVidas();  
  c.falar();  
}  
  
> dart mixin.dart  
Um Gato tem 7 vidas  
nao fala  
Um mamifero tem 1 vidas  
latindo
```

Singlets



Um **Singleton** é uma classe que tem uma única instância em toda o programa. Todos os objetos dessa classe fazem referência a mesma instância, permitindo que essa instância seja compartilhada por todo o programa.

No Dart o construtor **factory** permite retornar uma instância já existente de uma classe ao invés de criar um novo objeto.

Para criar singlets em Dart é recomendável o uso do construtor **factory**.

Singletons



```
comun.dart:  
class Comun {  
    int count = 0;  
  
    void inc() => count = count + 1;  
}  
  
void main() {  
    Comun a = Comun();  
    Comun b = Comun();  
    print("a = ${a.count}");  
    print("b = ${b.count}");  
    a.inc();  
    print("a = ${a.count}");  
    print("b = ${b.count}");  
}
```

Singletons



```
> dart comun.dart
a = 0
b = 0
a = 1
b = 0
```

Singletons



```
singleton.dart:  
class MySingleton {  
    static final MySingleton _singleton =  
MySingleton._internal();  
    int count = 0;  
  
    factory MySingleton() => _singleton;  
  
    MySingleton._internal() {  
        count = 0;  
    }  
  
    void inc() => count = count + 1;  
}
```

Singletons



```
void main() {  
    MySingleton a = MySingleton();  
    MySingleton b = MySingleton();  
    print("a = ${a.count}");  
    print("b = ${b.count}");  
    a.inc();  
    print("a = ${a.count}");  
    print("b = ${b.count}");  
}
```

```
> dart singleton.dart
```

```
a = 0  
b = 0  
a = 1  
b = 1
```