

## ***Instância da classe***

---

Agora que já aprendemos um pouco mais sobre o uso de classes e sua estrutura, vamos aprender a criar instâncias dessas classes, também conhecidas pelo nome de “objetos”.

Um objeto nada mais é que uma representação concreta de uma classe. De uma forma bastante simples e intuitiva, a classe é o molde e o objeto é o produto criado a partir deste molde.

Agora que já entendemos um pouco mais sobre o que são objetos, vamos aprender a criar nossos próprios objetos e como utilizá-los em nossas aplicações.

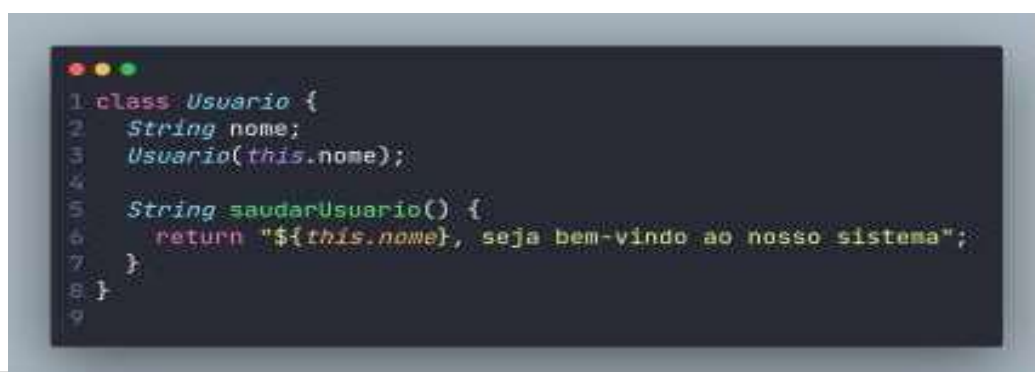
A screenshot of a code editor with a dark background and light-colored text. The code is in Dart and demonstrates creating an instance of a class named 'Medico'. It starts with an import statement for 'medico.dart', followed by a 'main' function. Inside 'main', a variable 'carlos' is assigned a new 'Medico' object with the name 'Carlos Alberto' and ID '123456'. Then, the 'toString' method is called on the 'carlos' object and printed to the console.

```
1 import 'medico.dart';
2
3 void main() {
4   Medico carlos = Medico("Carlos Alberto", 123456);
5
6   // Em ambos os casos o método toString será chamado
7   print(carlos);
8   print(carlos.toString());
9 }
10
```

## ***Referência “this”***

This (esta ou este) é uma referência à própria classe. Ou seja, quando desejamos informar ao Dart que desejamos pegar algo que está na mesma classe em que estamos declarando nosso método, colocamos essa palavra-chave para identificar que esse elemento é parte da classe. Vamos a um exemplo prático, tenho a certeza que ele lhe ajudará a entender melhor esse tipo de conceito da Orientação a Objetos.

### **Classe Usuario**

A screenshot of a code editor with a dark background and light-colored text. The code defines a class named 'Usuario'. It has a private field 'nome' of type 'String'. The constructor 'Usuario' takes 'this.nome' as an argument. There is a method 'saudarUsuario' that returns a string greeting using 'this.nome' in a template string.

```
1 class Usuario {
2   String nome;
3   Usuario(this.nome);
4
5   String saudarUsuario() {
6     return "${this.nome}, seja bem-vindo ao nosso sistema";
7   }
8 }
9
```

## Arquivo main



```
1 import 'usuario.dart';
2
3 void main() {
4   Usuario andre = Usuario("André Silva");
5
6   // Em ambos os casos o método toString será chamado
7   print(andre.saudarUsuario());
8 }
9
```

## Herança

---

A herança é um dos pilares fundamentais da Orientação a Objetos, possibilitando que uma classe seja criada com base em uma outra já existente, refinando seu comportamento.

Isso traz muitas vantagens em relação a um uso estrutural, que nos exigiria uma grande réplica de blocos de código com o intuito de alterar pequenos trechos dentro desses blocos.

Com o uso da herança, uma classe pode herdar as características e comportamentos de outra, refinando-os a ponto de melhorá-los ou mesmo modificá-los para atender melhor às suas necessidades específicas.

O uso da herança em Dart não se difere muito de outras linguagens que implementam o paradigma Orientação a Objetos. Basta adicionarmos a palavra chave **“extends”** ao final do nome da classe e definir de qual classe a mesma descende. Vejamos um exemplo prático na figura abaixo.

```
1 class Pessoa {
2   String? _email;
3   String? _telefone;
4 }
5
6 class PessoaFisica extends Pessoa {
7   String? _nome;
8   String? _cpf;
9 }
10
11 class PessoaJuridica extends Pessoa {
12   String? _razaoSocial;
13   String? _cnpj;
14 }
15
```

É importante ressaltar que assim como outras linguagens Orientadas a Objetos, o Dart não permite que uma classe herde mais de uma classe, o que é conhecido no mundo do desenvolvimento como herança múltipla. No entanto, se precisarmos utilizar múltipla herança, no Dart pode ser utilizada uma técnica conhecida como Mixin. O Mixin é um recurso utilizado na linguagem para adicionar características a uma classe sem precisar utilizar herança.

### ***Abstração de Classes***

---

Outro aspecto importante da Orientação a Objetos é a possibilidade de definir que uma classe é abstrata, isso permite que criemos classes genéricas, as quais só existem para servirem de base para outras, e desta forma declaramos que tais classes não podem ser utilizadas para prover objetos em nosso sistema.

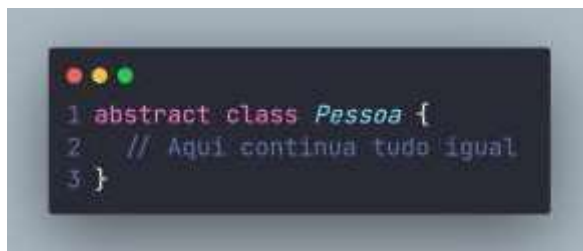
Se você ainda não entendeu o que isso quer dizer, vamos a um exemplo prático:

Imagine que estejamos criando nosso sistema bancário e para tal criamos a classe Conta. Essa por sua vez, é utilizada como base para a criação de outras classes, por exemplo, a classe conta corrente. Até aí tudo certo não é?

Mas tem algo que talvez não tenha ficado claro. E se eu quiser criar um objeto a partir da classe Conta? Faz sentido?

A resposta é bem simples: **Não**. A classe Conta só existe para que possamos definir características e ações comuns entre os tipos distintos de contas. Mas a conta em si "É abstrata" demais para existir. Afinal, como é uma conta? Tem limite, assim como a conta corrente. Tem rendimento como uma poupança ou conta de investimentos?

Para resolver esse problema, podemos definir que nossa classe Conta é abstrata e não poderá ser instanciada. Vejamos na figura abaixo como ficaria nosso código atualizado para que a classe conta seja abstrata.

A screenshot of a code editor with a dark background and light-colored text. The code is written in a syntax-highlighted language, likely Java. It shows the declaration of an abstract class named 'Pessoa'. The code is as follows:

```
1 abstract class Pessoa {  
2     // Aqui continua tudo igual  
3 }
```

Perceba que não houveram grandes mudanças na classe. Apenas foi adicionada a notação "abstract" antes da declaração da classe. Isso faz com que a classe seja considerada abstrata, impedindo que seja instanciada.