BACK-END

PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

JAVASCRIPT

AULA 4 e 5

**PARADIGMA**

Maneira ou abordagem para pensar e resolver problemas dentro de um campo. Na programação, o paradigma é uma forma de como você irá organizar o seu código.

**Paradigma Interativo**

Você escreve um conjunto de instruções, um passo a passo de como o computador deve realizar a tarefa.

Foco em **como fazer as coisas**, você dá uma sequência para que o computador siga.

**Paradigma Declarativo**

Você não dá mais uma sequência de como fazer uma tarefa, você só descreve o que quer fazer, você define o resultado sem especificar os passos para alcança-lo.

Exemplo SQL:

SELECT nome FROM alunos WHERE idade > 18;

Aqui, não é dito **COMO** buscar os alunos. Apenas **O QUE** queremos.

**Paradigma Funcional**

Neste paradigma, a ideia está sempre em separar as coisas, utilizando **FUNÇÕES**. O foco é total nas funções, sem utilização de várias variáveis mutáveis.

Exemplo:

const soma = (a,b) => a + b;

const dobro = (x) => x \* 2;

Ao invés de criar variáveis para fazer essas somas e dobrar números, são criadas apenas funções que realizam essa função.

**Paradigma Orientado a Objetos**

Paradigma focado no conceito de objetos, que são instâncias de classes. Você modela o mundo em termos de objetos que interagem entre si. Cada objeto possui propriedades e métodos e você organiza o código em torno desses objetos.

Um **objeto** é uma entidade que representa um elemento do mundo real ou um conceito abstrato, possuindo **propriedades (dados; eram as variáveis)** e **métodos (ações ou comportamentos; eram as funções)**.

Ex:

Caneta

Propriedades:

Cor – Azul

Marca – BIC

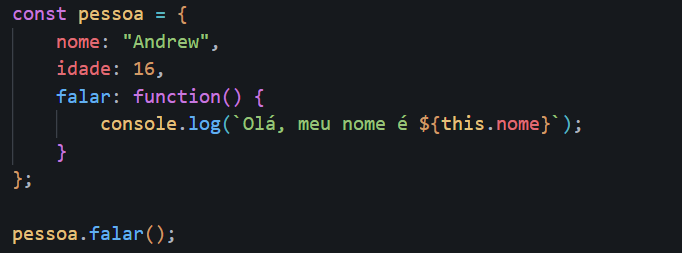
Tipo – Esferográfica

Métodos:

Escrever

**Objeto Literal**

Um objeto literal é definido por um par de chaves, onde definimos um objeto sem utilizar um modelo (classe).

Ex: Exibe “Andrew”

Onde –

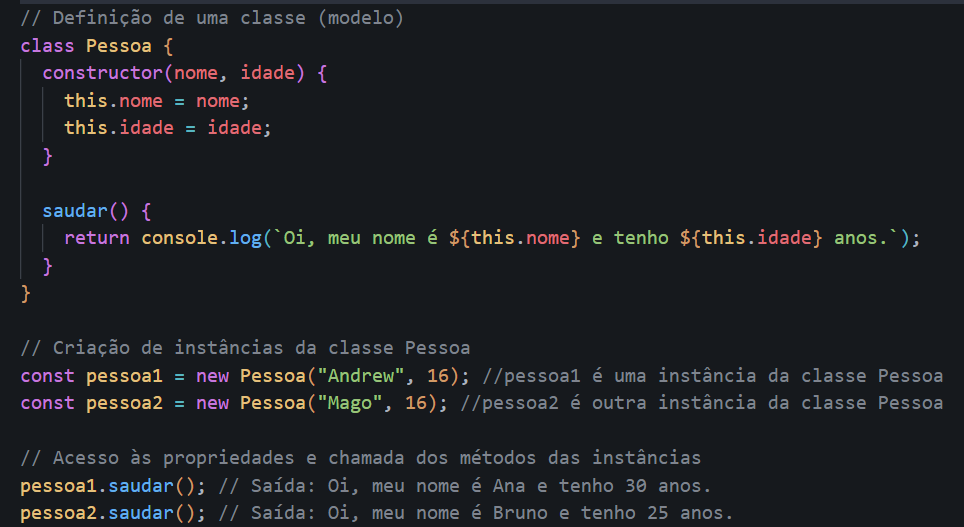
Nome e idade = Atributos (Propriedades)

Falar = Método

**Objeto Instanciado (Classe)**

Um objeto criado a partir de um **modelo** (classe), nesse tipo de objeto, é possível herdar propriedades e códigos.

Exemplo:



Nesse caso, pessoa1 e pessoa2 são **Objetos Instanciados (Uma Instância)**, pois são criados a partir do **modelo (classe) Pessoa.**

**CLASSE é MODELO do OBJETO**, ela não é um objeto, pois está vazia.

Uma classe deve sempre ser criada em **PascalCase**, sempre **começando** as palavras com letras **maiúsculas**.

Ex: **class** Pessoa {}

**Instância**

Instância é um objeto criado a partir de uma classe. Por exemplo, se “pessoa1” é criado **a** **partir** da classe “pessoa”, “pessoa1” é uma **instância** de **pessoa.** Você sempre cria a **classe** **primeiro**, **depois** a **instância.**

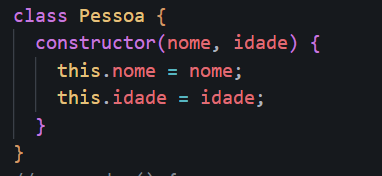
**Construtor**

O construtor é um método especial da Classe que permite a inicialização do Objeto e também a declaração dos atributos.

Quando a classe é instanciada, o construtor é chamado automaticamente para que os atributos sejam passados e a classe seja construída a partir deles.

Para definir os atributos que serão passados é utilizada a palavra-chave “**this**”. “this” referencia o objeto

Ex:



constructor(parâmetros) 🡪 permite que os objetos sejam baseados nesses atributos. Ex: 

“new” 🡪 Criar um novo objeto baseado em uma classe (“Pessoa” no caso)

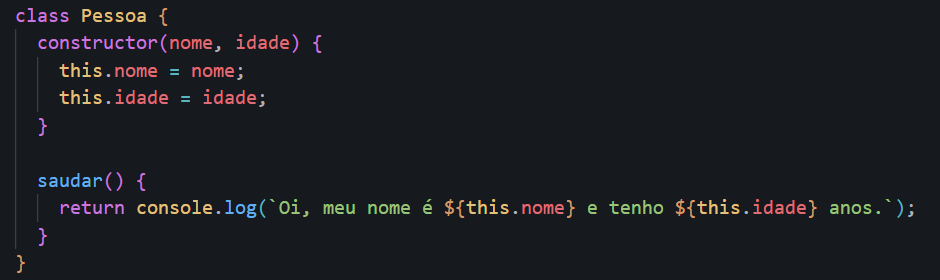
Pessoa(“Ana”(**Atributo** “Nome” **definido** pelo **construtor**), 30(Atributo “idade” **definido** pelo **construtor**)

Nesse caso, “this.nome” na classe seria “pessoa1.nome” no objeto.

**Métodos**

Basicamente funções que os objetos realizam.

Exemplo:



“saudar()” é o método.

**Instanciando uma classe**

Para instanciar uma classe, criamos utilizando a palavra-chave “new”. Ex:



Para chamar o método:



**Os 4 pilares da Orientação a Objetos**

* Abstração
* Encapsulamento
* Herança
* Polimorfismo

**Abstração**

Ideia de ocultar os detalhes complexos e apenas mostrar apenas o que é relevante.

Exemplo: Se você tem uma televisão, você não **precisa** saber como ela funciona internamente, só precisa saber que poder ligar ela e desligá-la.

Quando o usuário chamar uma área de um retângulo por exemplo, ele não precisa saber como é feito, ele só precisa saber o resultado. Não é necessário saber todo o cálculo.

**Encapsulamento**

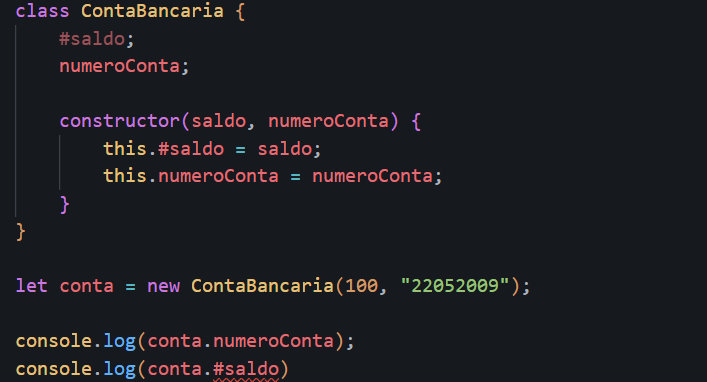
O encapsulamento é usado para **proteger** dados de **acessos externos**. Quando utilizado, a classe ou o objeto não podem ser modificados por nada externo, eles só podem ser **alterados** através de **métodos da própria classe**.



Isso não poderia acontecer com o encapsulamento, ele protege qualquer dado de ser manipulado fora da própria classe.

* **Atributo privado**

Para encapsular um atributo no JS, basta colocar um “#” antes do nome do atributo. Exemplo:



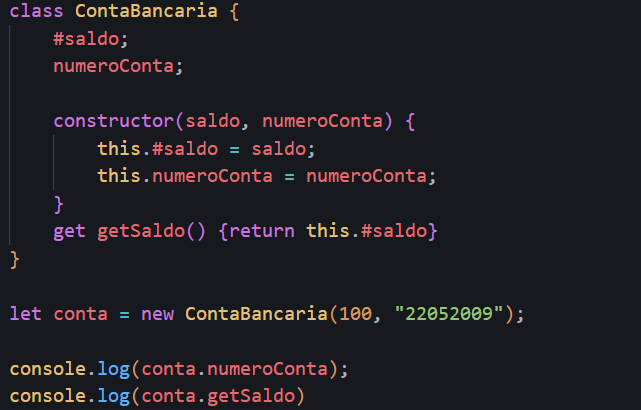
O Display seria: 22052009 e 100, porém, como saldo está encapsulado, essa mensagem é exibida:



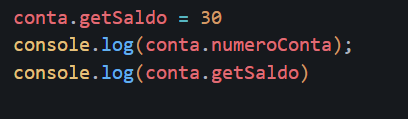
“#saldo” não é acessível fora da classe porque ele tem propriedade privada”

**Getter**

Getter é um método usado para ler o valor de um atributo de forma controlada. Ele é geralmente usado para retornar um valor.

Getter funcionando, agora “(conta.getSaldo)” funciona corretamente.

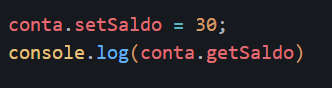
Qualquer tentativa de mudar saldo não mudaria nada. Ex:



Continua exibindo “100”.

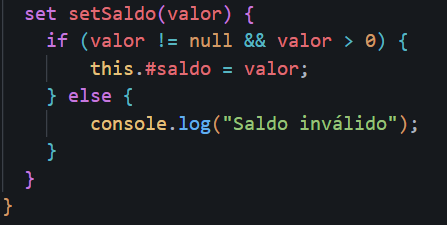
**Setter**

Utilizado para alterar o valor de um produto de forma controlada e também para validações. Ex:

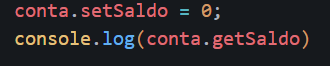


Set altera e get exibe o valor

Validações:



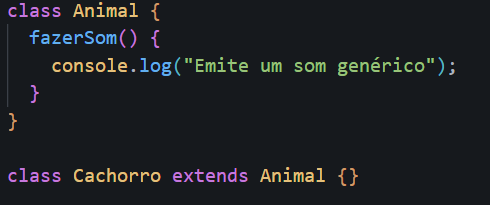
Quando executado novamente esse código:



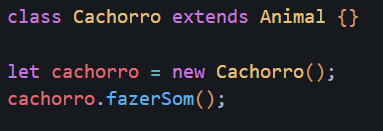
Exibicão: “Saldo inválido”  
 “100” (exibe 100 porque o valor já era 100, ele não muda, só mudaria se a validação do “set” desse certo)

* **Herança**

A herança é uma classe que deixa atributos e métodos a outras classes. Por exemplo, uma classe “animal” possui um método “fazerSom”, a classe que for criada **extendida** vai ser a **Classe filha** da **Classe pai**, essa classe filha seria a classe “cachorro”, para fazer a herança é utilizado o termo “extends”.



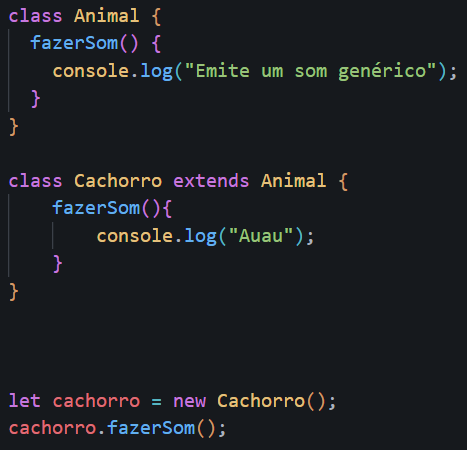
A herança funciona como uma biblioteca, que “importa” códigos.



Mesmo a classe e o objeto estando “vazios”, é possível utilizar o método “fazerSom()”, pois foi herdado da classe “Animal”. Exibe:

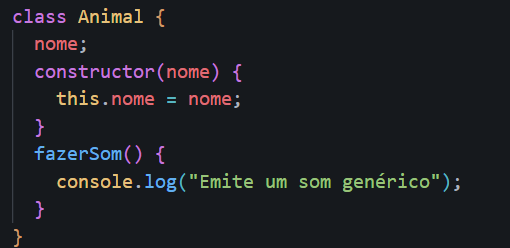


Além de poder reutilizar todos os métodos e atributos da **classe pai**, a **classe filha** pode reescrever os códigos conforme a necessidade:



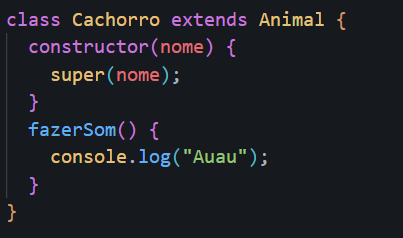
Exibe: “Auau”

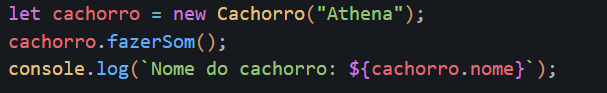
**Super (Classe Pai)**

Super é a palavra reservada para se referir a classe pai, ela é utilizada dentro das classes filhas, para chamar métodos ou valores da classe pai. Para testar, vamos definir um constructor com o atributo “nome”. ****

**Super (Classe Filha)**

Na classe filha vamos criar um constructor para utilizar o atributo herdado da classe pai, o “nome”.



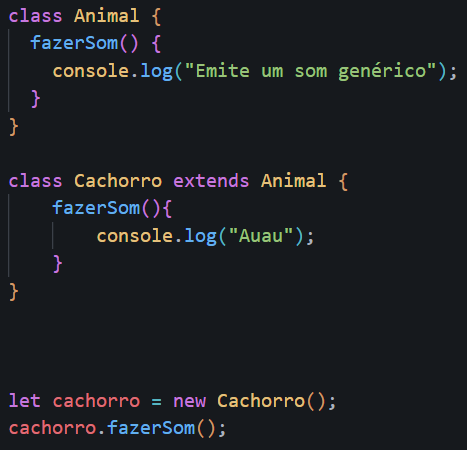


Note que o atributo “nome” não foi criado na classe “Cachorro”, mas sim na classe “Animal”, ele foi reaproveitado em “Cachorro” através de “super()”. Só é possível acessar os atributos da classe pai com “super()”, diferente dos métodos.

* **Polimorfismo**

Polimorfismo significa “muitas formas”. Ele permite que diversas classes possam executar o mesmo método de maneiras diferentes por exemplo. É executado junto com herança. Ex:

**Override**

Sobrescrita, ou overriding, é quando uma classe filha reimplementa um método de sua classe pai, ou seja, ela “substitui” o comportamento do método herdado. 

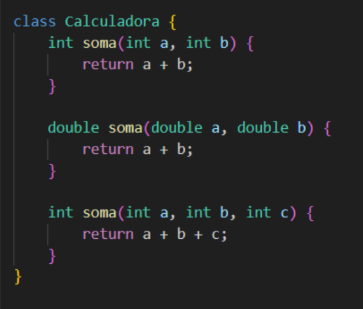
É o mesmo método, mas a classe “Cachorro” modificou essa classe conforme sua necessidade.

**Overload**

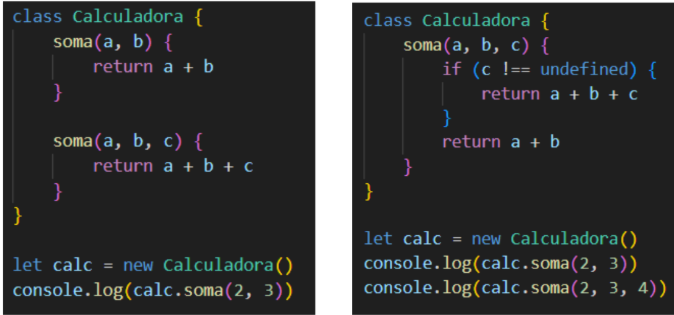
Polimorfismo de sobrecarga acontece quando um mesmo método tem várias versões, diferenciadas pelo número ou tipo de parâmetros.

O Polimorfismo de Sobrecarga é um tema que gera muita dúvida em JavaScript, porque ele não funciona da mesma forma que em linguagens fortemente tipadas como Java ou C#.

Exemplo em Java:



Exemplo em JS (Não funciona da mesma forma):



Os dois códigos funcionam iguais.

Note que todos os métodos são iguais: “soma”.