
Orientação a Objetos

JavaScript

Linguagens Script @ LEI / LEI-PL / LEI-CE


Departamento de Engenharia Informática e de Sistemas

Cristiana Areias < cris@isec.pt >

2022/2023

JavaScript

Orientação a Objetos

- › Orientação a Objetos em JavaScript 
- › *Factory Functions*
- › *Construtor Functions*
- › Classes
- › Prótótipos

> JavaScript e Orientação a Objetos

■ Multi-Paradigma:

- O JavaScript combina aspetos de vários paradigmas de programação: procedimental, funcional e de programação orientada a objetos (POO), mas....



O JavaScript permite a programação orientada a objectos (POO), embora não siga os paradigmas mais puros associados à POO.

- *Tal como o resto do JavaScript o objetivo é o resultado final e as funcionalidades oferecidas, em contraste com o respeito por paradigmas ou padrões de programação mais rígidos.*

> Objetos > Criação

Criação de um único objeto com
Notação Literal
(initializer notation)



```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  info: function () {  
    console.log(`Info do ${this.nome}...`);  
  }  
};
```



Criação de um único objeto com
new Object()

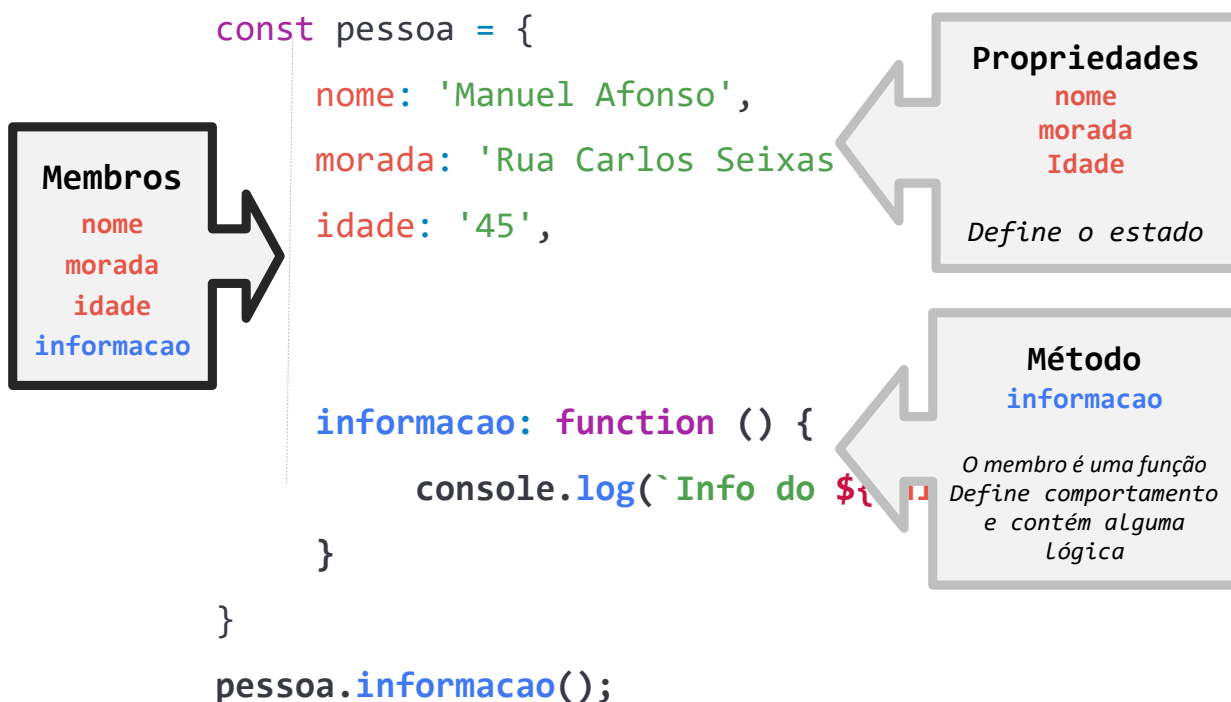
Definir um **Construtor** de objeto e criar objectos do tipo especificado

Criar um objecto usando
Object.create()



> Objetos > Breve Revisão...

Criação de Objecto: Notação Literal



> Objetos > Breve Revisão...

```
const pessoa1 = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  informacao: function () {  
    console.log(`Infssso do ${this.nome}...`);  
  }  
}  
  
pessoa1.fazQualquerCoisa();
```



```
const pessoa2 = {  
  nome: 'José Afonso',  
  morada: 'Rua do Jose',  
  idade: '42',  
  informacao: function () {  
    console.log(` Infssso do ${this.nome}...`);  
  }  
}  
  
pessoa2.fazQualquerCoisa();
```

Objetos com
comportamento!



Duplicação de código?
E se os métodos estão com
erros?



> Objetos > Breve Revisão...

```
const pessoa1 = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  informacao: function () {  
    console.log(`Infssso do ${this.nome}...`);  
  }  
}  
pessoa1.fazQualquerCoisa();
```



```
const pessoa2 = {  
  nome: 'José Afonso',  
  morada: 'Rua do Jose',  
  idade: '42',  
  informacao: function () {  
    console.log(` Infssso do ${this.nome}...`);  
  }  
}  
pessoa2.fazQualquerCoisa();
```

Criação de vários objetos

■ Constructor Function

Consiste na declaração de propriedades e métodos a serem adicionados a cada novo objeto com a estrutura definida – *new*.

■ Factory Function

Cria e *retorna* um novo objeto.

> Objetos > Criação

```
const pessoa1 = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  informacao: function () {  
    console.log(`Infssso do ${this.nome}...`);  
  }  
}  
pessoa1.fazQualquerCoisa();
```



Factory Function

ou

Constructor Function

```
const pessoa2 = {  
  nome: 'José Afonso',  
  morada: 'Rua do Jose',  
  idade: '42',  
  informacao: function () {  
    console.log(` Infssso do ${this.nome}...`);  
  }  
}  
pessoa2.fazQualquerCoisa();
```

Criação de vários objetos

■ Factory Function

Cria e *retorna* um novo objeto.

> Criação > Factory Function

Criação de Objecto: *Factory Function*

```
function createPessoa(nome, morada, idade) {  
  return {  
    nome: nome,  
    morada, ES6  
    idade,  
    info() {  
      return `Pessoa ${nome} - ${idade} : ${morada}`;  
    },  
    info2: function () {  
      return 'Método Info2';  
    }  
  };  
}  
  
const jose = createPessoa("José", "Rua do José", 53);  
const maria = createPessoa("Maria", "Rua da Maria", 27);  
const nuno = createPessoa("Nuno", "Rua do Nuno", 18);
```

> Criação > Factory Function

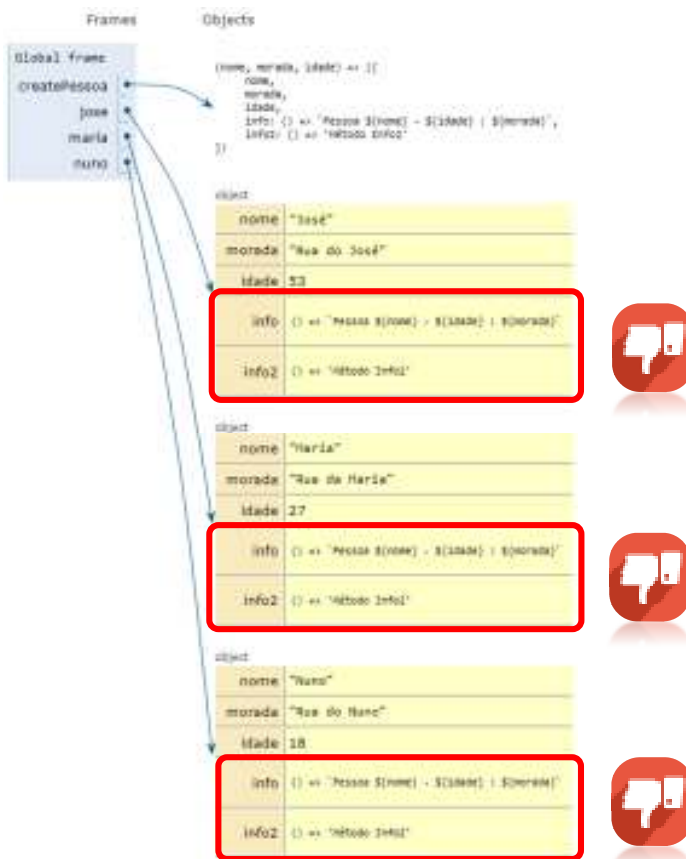
```
const createPessoa = (nome, morada, idade) => ({  
  nome,  
  morada,  
  idade,  
  info: () => `Pessoa ${nome} - ${idade} : ${morada}`,  
  info2: () => 'Método Info2'  
});  
  
const jose = createPessoa("José", "Rua do José", 53);  
const maria = createPessoa("Maria", "Rua da Maria", 27);  
const nuno = createPessoa("Nuno", "Rua do Nuno", 18);  
  
console.log(jose, maria, nuno);  
console.log(jose.info());  
console.log(maria.info());  
console.log(nuno.info());
```

Factory Function
usando
Arrow Function

```
▶ {nome: 'José', morada: 'Rua do José', idade: 53, info: f, info2: f}  
▶ {nome: 'Maria', morada: 'Rua da Maria', idade: 27, info: f, info2: f}  
▶ {nome: 'Nuno', morada: 'Rua do Nuno', idade: 18, info: f, info2: f}  
Pessoa José - 53 : Rua do José  
Pessoa Maria - 27 : Rua da Maria  
Pessoa Nuno - 18 : Rua do Nuno
```

> Factory Function

JavaScript



<https://pythontutor.com/javascript.html#mode=display>



isec
Engenharia

> Criação > Construtor Function

JavaScript

Criação de Objecto: **Constructor Function**

```
function Pessoa(nome, morada, idade) {  
  this.nome = nome;  
  this.morada = morada;  
  this.idade = idade;  
  this.informacao = function () {  
    console.log(`Pessoa ${this.nome}-${this.idade}:${this.morada}`);  
  }  
}  
  
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 20);
```

Também é possível com funções de expressão!

↓

✓

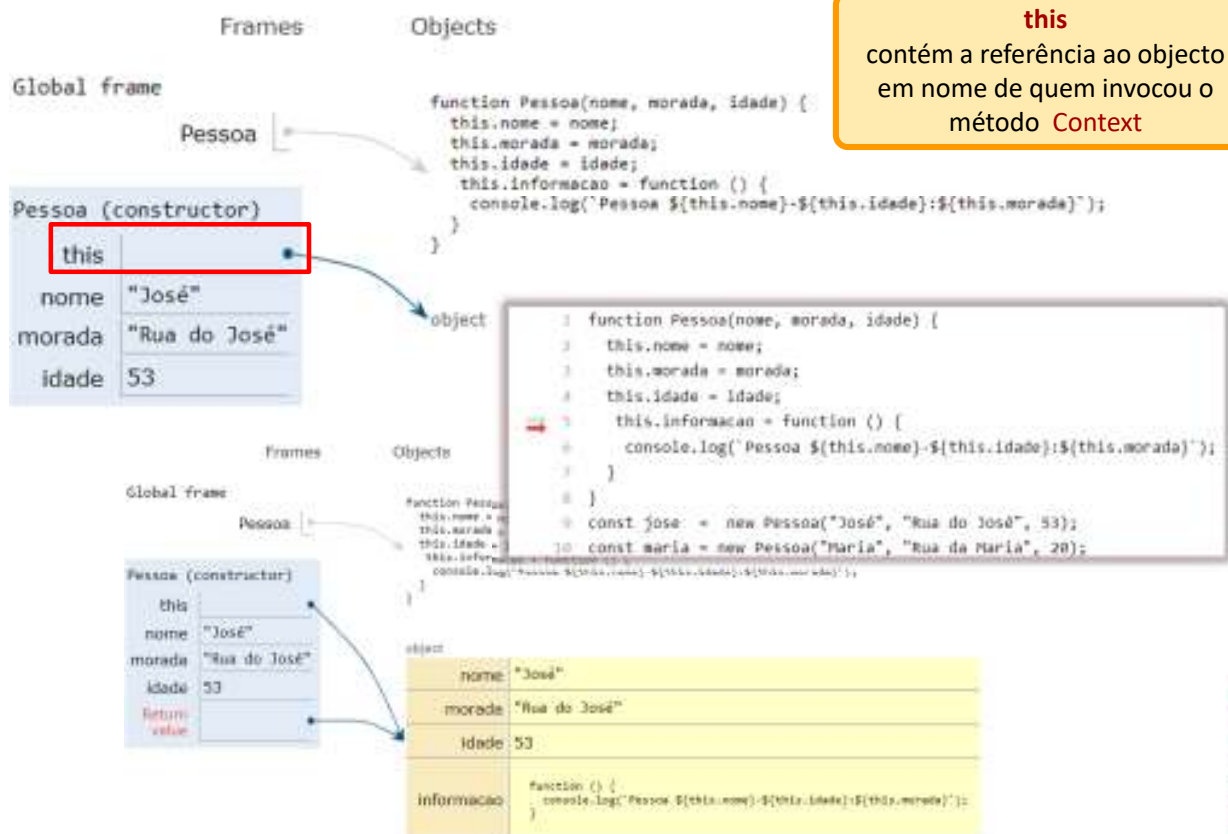
isec
Engenharia

> Construtor Function - new

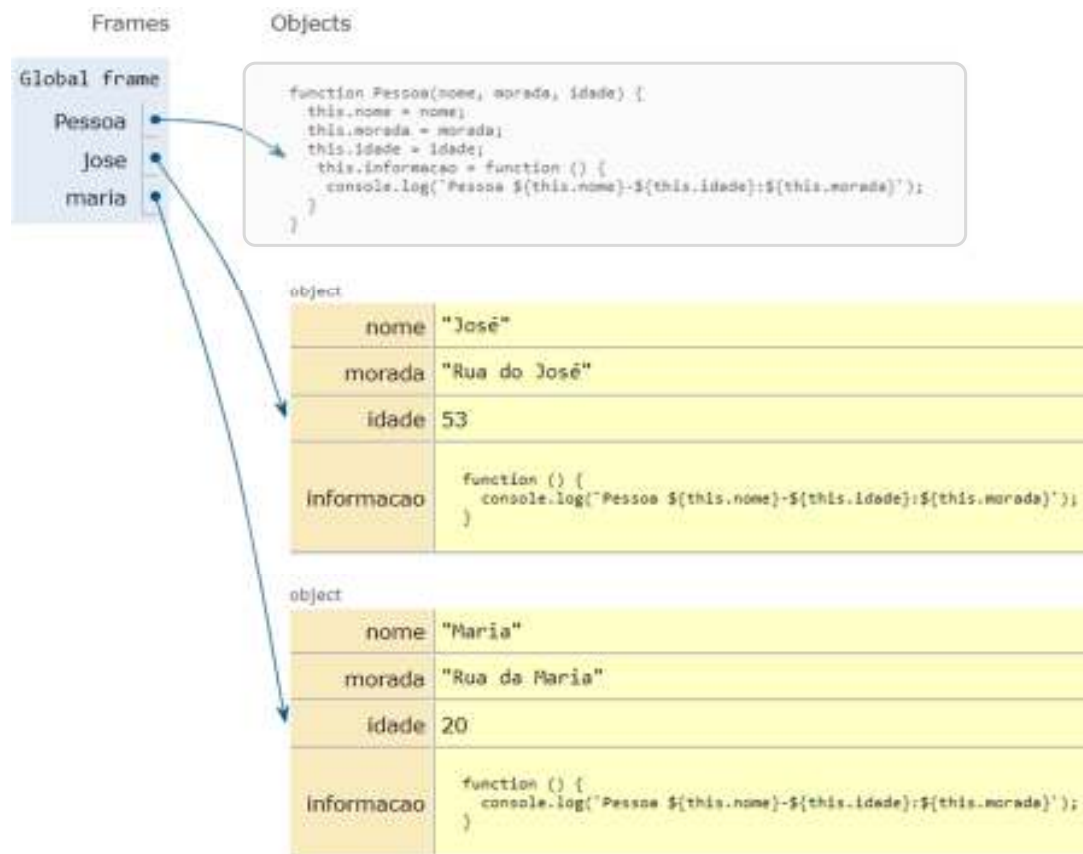
```
function Pessoa(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
}  
  
const jose = new Pessoa("José", "Rua do José", 53);
```

- 1) Cria um objecto vazio {}
- 2) A função é invocada no qual this = {}
- 3) {} é vinculado ao *prototype*, ao objecto Pessoa
- 4) A Função faz o retorno automático quando se recorre ao operador **new**

> Construtor Function



> Construtor Function



> Objetos > Criação > *new*

- O operador **new** permite criar e inicializar objetos
- Este operador deve ser seguido da invocação da função
 - `let x = new Object();` **Instância de um Objecto**
 - Igual a {}
 - `let a = new Array();`
 - Igual a []
 - `let z = new String();`
 - Igual a '', '', ''
 - `let z = new Number();`
 - 1, 2, 3, ...
 - `let y = new Date();`

Operador new

Funções usadas desta forma são referidas como **funções construtoras**

> Propriedade > *Constructor*


- Todos os objectos incluem a propriedade constructor de forma automática, mesmo se forem criados recorrendo à **notação literal**.
- Todas as funções são objectos, portanto, todas as funções também incluem esta propriedade

```
let obj = new Object({ a: 1, b: 2 });
```


```
console.log("obj", obj);
```

```
let array = new Array('a', 'b');
```

```
console.log("array", array);
```



```
obj ▾ {a: 1, b: 2} 0
  a: 1
  b: 2
  ▾ [[Prototype]]: Object
    * constructor: f Object()
    * hasOwnProperty: f hasOwnProperty()
    * isPrototypeOf: f isPrototypeOf()
    * propertyIsEnumerable: f propertyIsEnumerable()
    * toLocaleString: f toLocaleString()
    * toString: f toString()
    * valueOf: f valueOf()
    * __defineGetter__: f __defineGetter__()
    * __defineSetter__: f __defineSetter__()
    * __lookupGetter__: f __lookupGetter__()
    * __lookupSetter__: f __lookupSetter__()
    * __proto__: {...}
    * get __proto__: f __proto__()
    * set __proto__: f __proto__()
```



```
array
  ▾ (2) ["a", "b"] 0
    0: "a"
    1: "b"
    length: 2
    ▾ [[Prototype]]: Array(0)
      * at: f at()
      * concat: f concat()
      * constructor: f Array()
```

> *Constructor vs Factory Function*

Constructor Function



- Recorre ao **new** para criar um novo objeto
- Especifica o **this** dentro da função para esse objeto;
- Retorna o objeto de forma automática.



Factory Function



- A função é chamada como uma função "regular" do javascript
- Necessita de retornar nova instância de um objeto.

> Objetos > Funções > Métodos

```
function Pessoa(nome, morada, idade) {  
  this.nome = nome;  
  this.morada = morada;  
  this.idade = idade;  
  
  this.info = function () {  
    console.log("Pessoa:" + nome);  
  }  
  this.info2 = function () {  
    console.log("Método info2");  
  }  
}
```



```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 27);  
const nuno = new Pessoa("Nuno", "Rua do Nuno", 18);
```

> Objetos > Funções > Métodos



```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 27);  
const nuno = new Pessoa("Nuno", "Rua do Nuno", 18);
```

Como resolver
a **duplicação**
de **código** na
função
construtora?

> Objetos > Funções > Métodos

```
function Pessoa(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
}
```



```
const pessoaMetodos = {  
    info() {  
        console.log(`Pessoa ${this.nome}${this.idade}:${this.morada}`);  
    }  
}
```

```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 20);  
jose.info = pessoaMetodos.info;  
maria.info = pessoaMetodos.info;  
jose.info();  
maria.info();
```



> Prototype

- JavaScript é uma linguagem de programação **baseada em protótipos** (**Prototype-based programming**), permitindo **fornecer um certo tipo de herança**, diferente das heranças existentes nas linguagens de POO mais “puras” ou “tradicionais” como o C++, ou Java

*“**Prototype-based programming** is a style of object-oriented programming in which behaviour reuse (known as inheritance) is performed via a process of reusing existing objects that serve as prototypes. This model can also be known as **prototypal**, **prototype-oriented**, classless, or instance-based programming.”, en.wikipedia.org, 2022*

- O conceito de **prototypal inheritance** permite a reutilização de código, sendo uma característica muito importante do *JavaScript*.

> Prototype

- Um **protótipo** (*prototype*) é o mecanismo pelo qual objetos *JavaScript* **herdam** recursos de outros objetos.
 - Funciona como um objeto modelo;
 - O objeto protótipo de um objeto, também pode ter um objeto de protótipo- **cadeia de protótipos**
- O objeto **Object.prototype** disponibiliza um conjunto de métodos *built-in* e propriedades, como exemplo:

- `toString()`
- `valueOf()`
- ...

```
> console.log(Object.prototype.constructor === Object);  
true
```

Herança em
JavaScript



> Prototype

```
function Pessoa(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
}  
console.log(Pessoa);  
console.log("Prototype", Pessoa.prototype);
```

```
f Pessoa(nome, morada, idade) {  
  this.nome = nome;  
  this.morada = morada;  
  this.idade = idade;  
}
```

```
Prototype {constructor: f} ⓘ  
  ▶ constructor: f Pessoa(nome, morada, idade)  
  ▶ [[Prototype]]: Object
```

> Prototype

```
function Pessoa(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
}
```

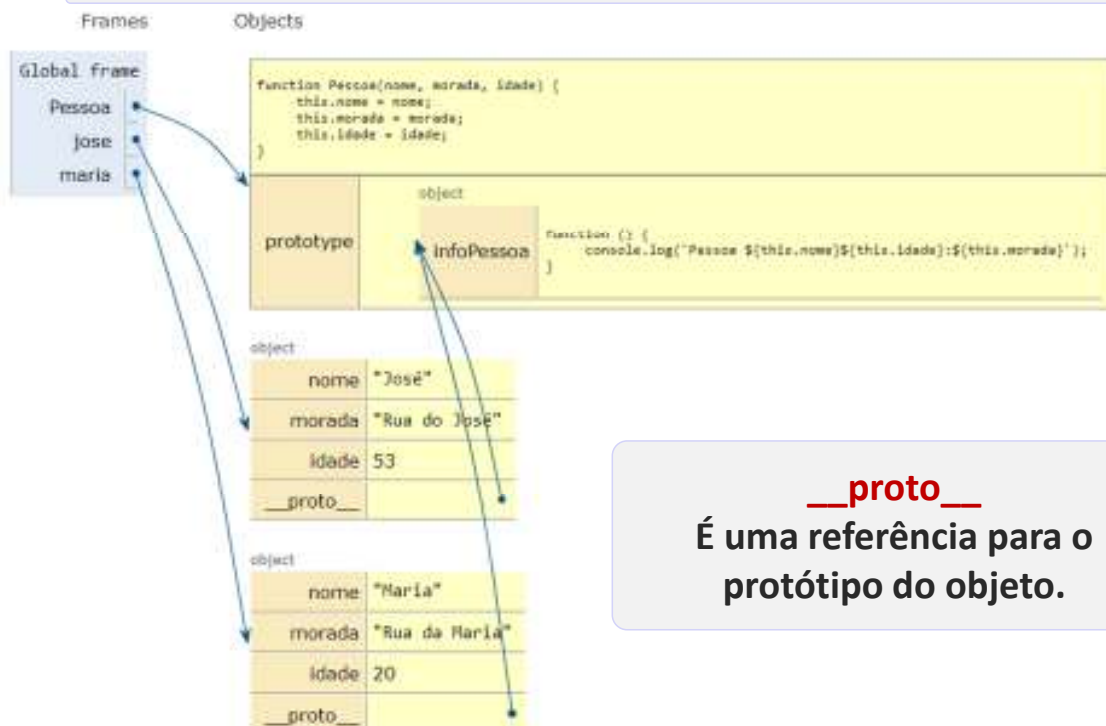
```
▼ Pessoa {nome: 'José', morada: 'Rua do José', idade: 53} ⓘ  
  idade: 53  
  morada: "Rua do José"  
  nome: "José"  
  → [[Prototype]]: Object  
  → infoPessoa: f ()  
  → constructor: f Pessoa(nome, morada, idade)  
  → [[Prototype]]: Object
```

```
Pessoa.prototype.infoPessoa = function () {  
    console.log(`Pessoa${this.nome}${this.idade}:${this.morada}`);  
}  
  
const jose = new Pessoa("José", "Rua do José", 53);
```



> Prototype e __proto__

```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 20);
```



__proto__
É uma referência para o
protótipo do objeto.

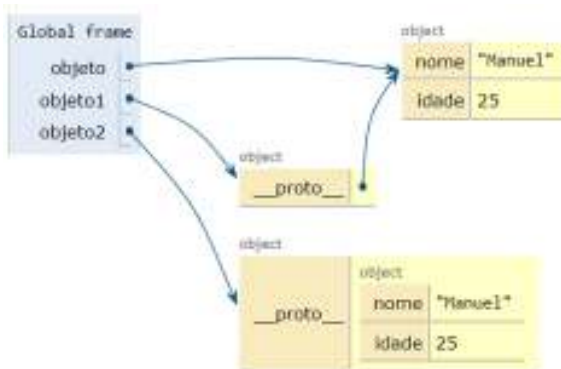
> Prototype > create()

- O método **create()** do **Object** permite adicionar **protótipos** a objetos, isto é, permite criar um objeto usando um objeto já existente **como protótipo** de um novo objeto.

```
let objeto = { nome: 'Manuel', idade: 25 };
```

```
let objeto1 = Object.create(objeto);
```

```
let objeto2 = Object.create({ nome: 'Manuel', idade: 25 });
```



```
console.log(objeto)
console.log(objeto1)
console.log(objeto2)

> {nome: 'Manuel', idade: 25}
> {}
> {}
```

> Prototype > create()

```
function createPessoa(nome, morada, idade) {
```

```
  let pessoa = Object.create(pessoaMetodos);
```

```
  pessoa.nome = nome;
```

```
  pessoa.morada = morada;
```

```
  pessoa.idade = idade;
```

```
  return pessoa;
```

```
}
```

```
const pessoaMetodos = {
```

```
  info() {
```

```
    console.log(`${this.nome} ${this.idade} Anos`);
```

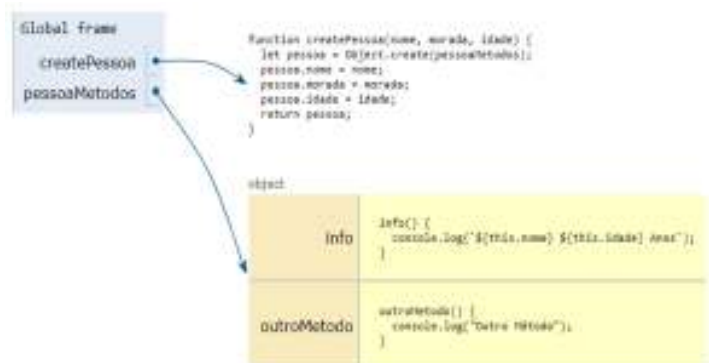
```
  },
```

```
  outroMetodo() {
```

```
    console.log("Outro Método");
```

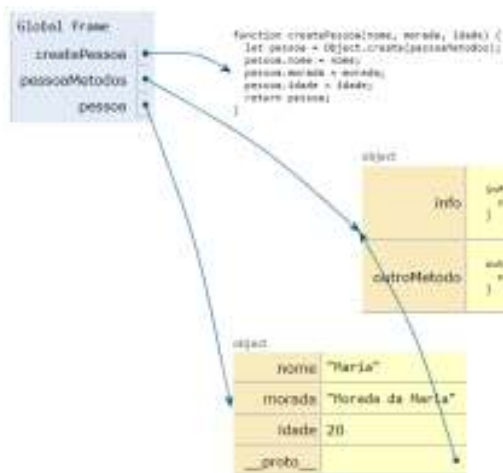
```
  }
```

```
};
```



> Prototype > create()

```
let pessoa = new createPessoa("Maria", "Morada da Maria", 20);
console.log("Pessoa", pessoa)
pessoa.info();
pessoa.outroMetodo();
```

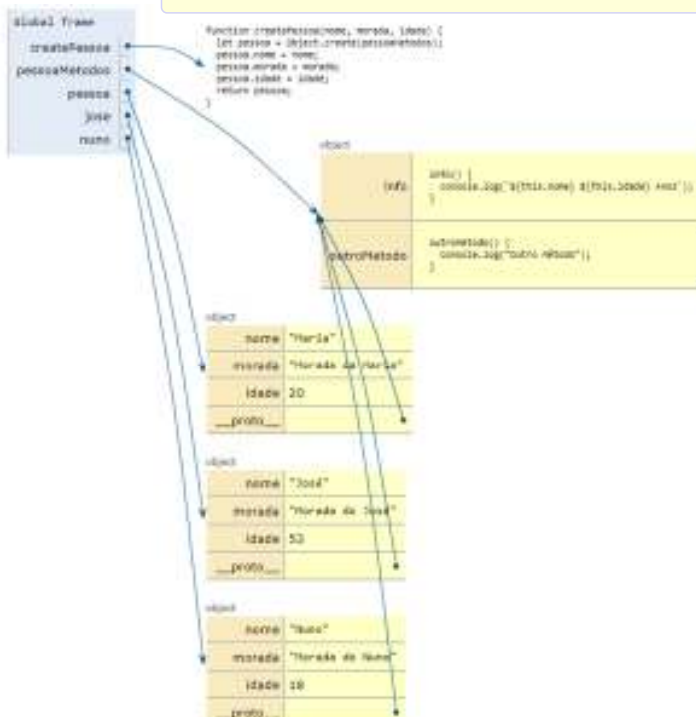


```
Pessoa {nome: 'Maria', morada: 'Morada da Maria', idade: 20}
  idade: 20
  morada: "Morada da Maria"
  nome: "Maria"
  [[Prototype]]: Object
    info: f info()
    outroMetodo: f outroMetodo()
    [[Prototype]]: Object
```

Maria 20 Anos
Outro Método

> Prototype > create()

```
const pessoa = new createPessoa("Maria", "Morada da Maria", 20);
const jose = new createPessoa("José", "Morada do José", 53);
const nuno = new createPessoa("Nuno", "Morada do Nuno", 18);
```



> Prototype > create()

```
let obj = Object.create(null);  
console.log(obj);
```



No properties



Iguais?

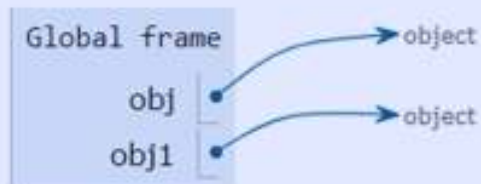
```
let obj = {};  
let obj1 = Object.create(Object.prototype);  
console.log(obj);  
console.log(obj1)
```



[[Prototype]]: Object



[[Prototype]]: Object



> Criação de Objetos > Class

**Class em
JavaScript?**



Classes em JavaScript

- › Classes em Javascript
- › Herança com classes
 - › Prototype e `__proto__`
 - › *extends*
- › *Getters* e *Setters*
- › Propriedades e Métodos Privados
- › Propriedades Estáticas



< 302 >

> Classes

- O conceito de **class** surgiu a partir da versão ES6, podendo ser vista como uma sintaxe diferente para implementar *constructor functions* e *prototypes* (mas não só!);
- Forma alternativa para especificação de um “template” para criação de objetos;
- Permitem organizar código conceptualmente, encapsular dados relacionados e simplificar a reutilização de código (herança);
- Permitem também simular o conceito de *prototypal inheritance*;
- Ao contrário das funções, as classes não são **hoisted**;

> Class > Estrutura básica

class Pessoa {

constructor(nome, morada, idade) {

this.nome = nome;

this.morada = morada;

this.idade = idade;

}

infoPessoa () {

return ` \${this.nome} - \${this.morada} - \${this.idade} `;

}

}

const jose = **new Pessoa**("José", "Rua do José", 53);

const maria = **new Pessoa**("Maria", "Rua da Maria", 20);

jose.infoPessoa();

maria.infoPessoa();

Semelhante ao
constructor
functions

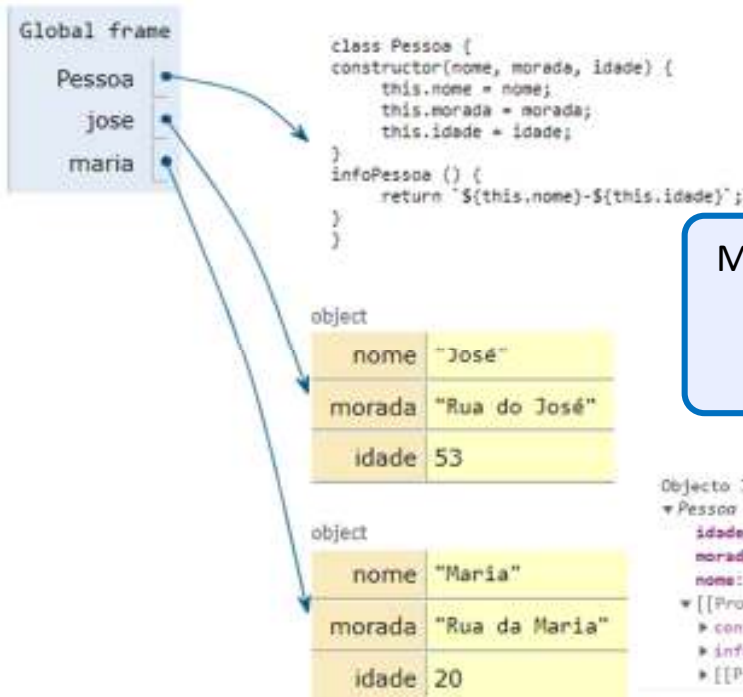
As propriedades que
se pretende adicionar
ao protótipo, são
definidas no corpo da
classe!

> Classes > Considerações

- O método **constructor()**, opcional, é chamado automaticamente, permitindo criar e inicializar um objeto (nova instância da class);
- Só pode existir um único método **constructor()**, caso contrário, será lançado um erro de sintaxe;
- A declaração de uma class, ao contrário das funções, não é hoisted, ou seja é necessário declarar a classe e só depois aceder-lhe, caso contrário é gerado um *ReferenceError*;
- Não se usa a keyword *function* para definir os métodos;
- Numa class, é sempre necessário referir a outros métodos com o prefixo **this**;

> Class e Prototype

```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 20);
```



Método **infoPessoa** é adicionado à propriedade **prototype** da classe Pessoa



```
Object Jose =  
▼ Pessoa {nome: "José", morada: "Rua do José", idade: 53} ⓘ  
  idade: 53  
  morada: "Rua do José"  
  nome: "José"  
  ▼ [[Prototype]]: Object  
    ▶ constructor: class Pessoa  
    ▶ infoPessoa: f infoPessoa()  
    ▶ [[Prototype]]: Object
```

> Class e Prototype

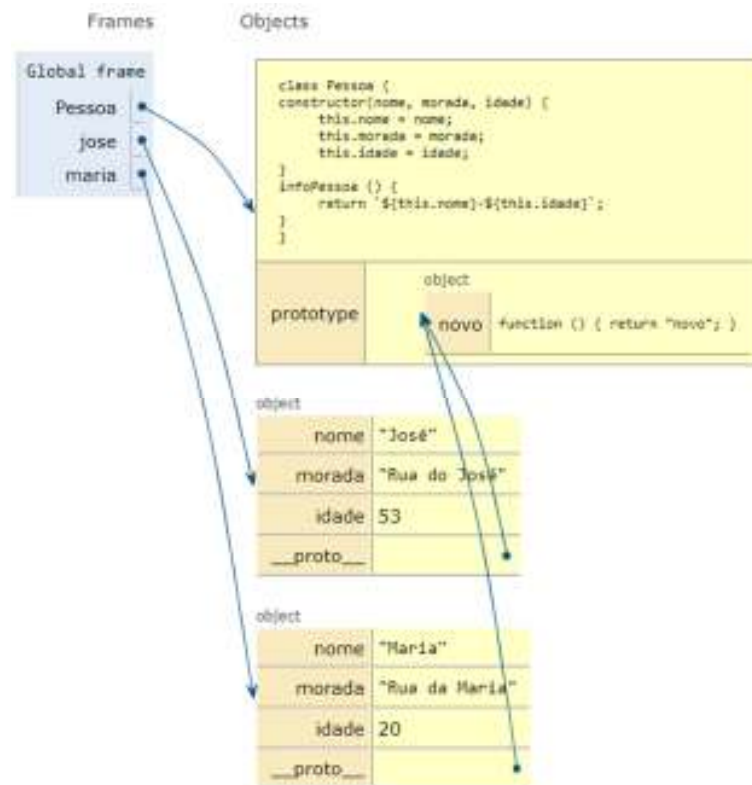
```
const jose = new Pessoa("José", "Rua do José", 53);  
const maria = new Pessoa("Maria", "Rua da Maria", 20);  
Pessoa.prototype.novo = function () { return "novo"; };  
console.log('jose', jose);  
console.log('jose.__proto__', jose.__proto__);
```



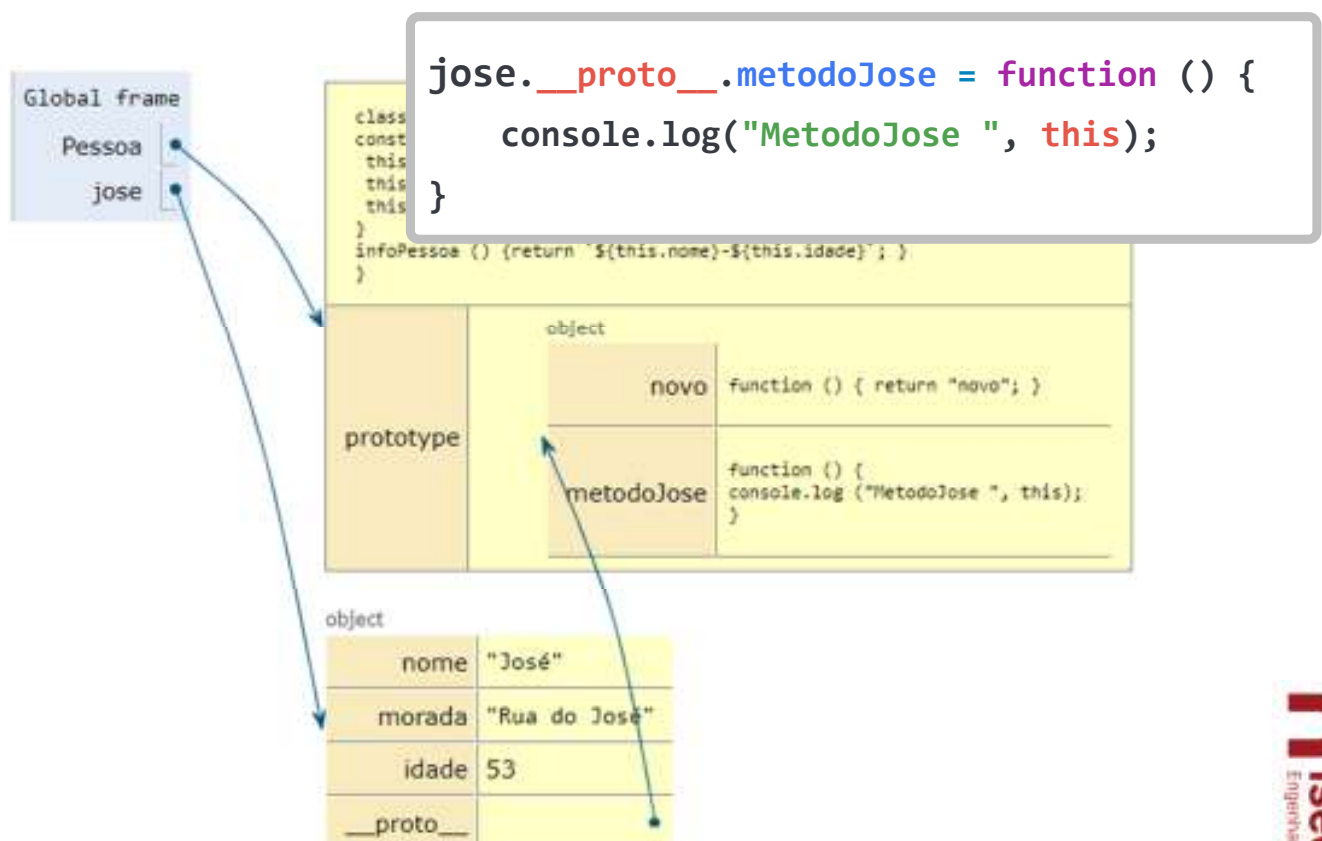
```
jose ▼ Pessoa {nome: "José", morada: "Rua do José", idade: 53} ⓘ  
  idade: 53  
  morada: "Rua do José"  
  nome: "José"  
  ▼ [[Prototype]]: Object  
    ▶ novo: f ()  
    ▶ constructor: class Pessoa  
    ▶ infoPessoa: f infoPessoa()  
    ▶ [[Prototype]]: Object  
  
jose.__proto__ ▼ {novo: f, constructor: f, infoPessoa: f} ⓘ  
  ▶ novo: f ()  
  ▶ constructor: class Pessoa  
  ▶ infoPessoa: f infoPessoa()  
  ▶ [[Prototype]]: Object
```

> Class e Prototype

```
Pessoa.prototype.novo = function () { reAturn "novo"; };
```

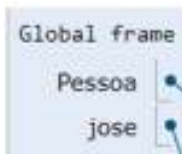


> class > prototype e __proto__



> class > prototype e __proto__

JavaScript



```
jose.__proto__.metodoJose = function () {  
    console.log("MetodoJose ", this);  
}
```

```
class  
const  
this  
this  
this  
}
```

```
infoPessoa () {return `${this.nome}-${this.idade}`; }  
}
```

object

novos function () { return "novos"; }

```
const maria = new Pessoa("Maria", "Rua da Maria", 20);  
maria.metodoJose(); // ?
```

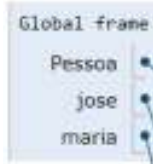


nome	"José"
morada	"Rua do José"
idade	53
__proto__	

isec
Engenharia

> class > prototype e __proto__

JavaScript



```
class Pessoa {  
    constructor(nome, morada, idade) {  
        this.nome = nome;  
        this.morada = morada;  
        this.idade = idade;  
    }  
    infoPessoa () {return `${this.nome}-${this.idade}`; }  
}
```

prototype

object

novos

function () { return "novos"; }

metodoJose

```
function () {  
    console.log ("MetodoJose ", this);  
}
```

object

nome	"José"
morada	"Rua do José"
idade	53
__proto__	

object

nome	"Maria"
morada	"Rua da Maria"
idade	20
__proto__	

MetodoJose ▾ Pessoa {nome: 'Maria', morada: 'Rua da Maria', idade: 20}

idade: 20

morada: "Rua da Maria"

nome: "Maria"

▾ [[Prototype]]: Object

▸ metodoJose: f ()

▸ novos: f ()

▸ constructor: class Pessoa

▸ infoPessoa: f infoPessoa()

▸ [[Prototype]]: Object

isec
Engenharia

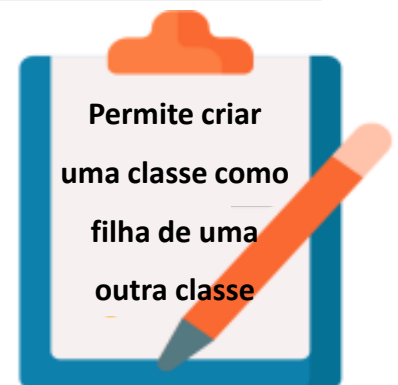
> class > *extends*

- As classes permitem implementar herança com recurso ao *extend*;
- Uma classe pode fazer o *extend* de outra classe;
- A classe filha herda as propriedades e métodos da classe pai;
- É obrigatório invocar **super()** na classe filha, de forma a ser executado o constructor da classe pai, ainda que sem todos os parâmetros.

```
class Pessoa {  
  constructor(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
  }  
  infoPessoa() {  
    return `${this.nome}-${this.idade}`;  
  }  
}  
class Aluno extends Pessoa {  
  constructor(nome, morada, idade, numero) {  
    super();  
    this.numero = numero;  
  }  
}
```

> class > *extends*

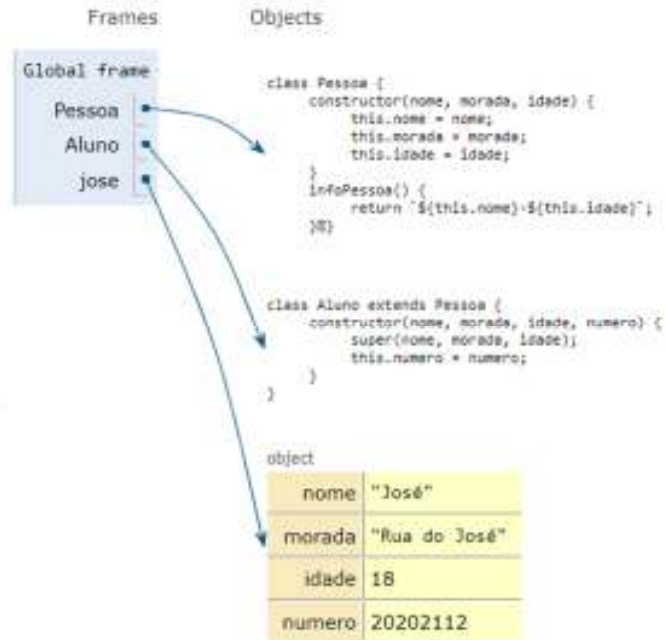
```
class Pessoa {  
  constructor(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
  }  
  infoPessoa() {  
    return `${this.nome}-${this.idade}`;  
  }  
}  
class Aluno extends Pessoa {  
  constructor(nome, morada, idade, numero) {  
    super(nome, morada, idade);  
    this.numero = numero;  
  }  
}
```



> class > extends

```
const jose = new Aluno("José", "Rua do José", 18, 20202112);
```

```
▼ Aluno {nome: 'José', morada: 'Rua do José', idade: 18, numero: 20202112}
  idade: 18
  morada: "Rua do José"
  nome: "José"
  numero: 20202112
  ▼ [[Prototype]]: Pessoa
    ► constructor: class Aluno
    ▼ [[Prototype]]: Object
      ► constructor: class Pessoa
      ► infoPessoa: f infoPessoa()
      ► [[Prototype]]: Object
```



> Getters e Setters

- **Funções** que permitem obter ou alterar um valor, sendo **vistos** como simples **propriedades**.

```
const aluno = {
  nome: "Maria Filipa Carvalho",
  estadoCivil: "Soleira",
  notas: [10, 15, 18, 9],

  get maxNotas() {
    return Math.max(...this.notas);
  },
  set nota(nota) {
    this.notas.push(nota);
  }
}
```

```
console.log('Nota Máxima', aluno.maxNotas)
aluno.nota = 10;
console.log(aluno)
```

```
Nota Máxima 18
* {nome: 'Maria Filipa Carvalho', estadoCivil: 'Soleira', notas: Array(5)}
  estadoCivil: "Soleira"
  maxNotas: (...)
  nome: "Maria Filipa Carvalho"
  notas: (5) [10, 15, 18, 9, 10]
  get maxNotas: f maxNotas()
  set nota: f nota(nota)
  [[Prototype]]: Object
```

> class > Getters e Setters

```
class Pessoa {  
  constructor(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
  }  
  infoPessoa() {  
    return `${this.nome}-${this.idade}`;  
  }  
  
  set idade(i) {  
    if ((i < 0) || (i > 120))  
      console.log("Idade Incorrecta!");  
    else  
      this.idade = i;  
  }  
  get idade() {  
    return this.idade;  
  }  
}
```

Encapsulamento
em JavaScript



> class > Getters e Setters

```
class Pessoa {  
  constructor(nome,  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
  }  
  infoPessoa() {  
    return `${this.nome}-${this.idade}`;  
  }  
}
```

```
Uncaught RangeError: Maximum call stack size exceeded  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]  
at Pessoa.set idade [as idade]
```

```
set idade(i) {  
  if ((i < 0) || (i > 120))  
    console.log("Idade Incorrecta!");  
  else  
    this.idade = i;  
}  
get idade() {  
  return this.idade;  
}
```



this.idade = i;



> class > Getters e Setters

```
class Pessoa {  
  constructor(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
  }  
  infoPessoa() {  
    return `${this.nome}-${this.idade}`;  
  }  
}
```

```
set idade(i) {  
  if ((i < 0) || (i > 120))  
    console.log("Idade Incorrecta!");  
  else  
    this._idade = i;  
}  
get idade() {  
  return this._idade  
}
```



> class > Getters e Setters

```
let jose = new Pessoa("José", "Rua do José", 153); Idade Incorrecta! ←  
const maria = new Pessoa("Maria", "Rua da Maria", 20);
```

```
console.log('Idade Jose', jose.idade);  
console.log('Idade Maria', maria.idade);  
console.log(jose);  
console.log(maria);
```

Idade Incorrecta!

Idade Jose undefined

Idade Maria 20

```
▼ Pessoa {nome: 'José', morada: 'Rua do José'} ⓘ  
  morada: "Rua do José"  
  nome: "José"  
  idade: (...)  
  ► [[Prototype]]: Object
```

```
▼ Pessoa {nome: 'Maria', morada: 'Rua da Maria', _idade: 20} ⓘ  
  morada: "Rua da Maria"  
  nome: "Maria"  
  _idade: 20  
  idade: (...)  
  ► [[Prototype]]: Object
```

> Propriedades Privadas

```
class Pessoa {
  #_idade
  #nome
  constructor(nome, morada, idade) {
    this.#nome = nome;
    this.morada = morada;
    this.idade = idade;
  }
  infoPessoa = () => `${this.nome}-${this.idade}`;
  get nome() { return this.#nome; }
  set nome(nome) { this.#nome = nome; }
  get idade() { return this.#_idade; }
  set idade(i) {
    if ((i < 0) || (i > 120))
      console.log("Idade Incorrecta!");
    else
      this.#_idade = i;
  }
}
```

> Propriedades Privadas

```
const jose = new Pessoa("José", "Rua do José", 153);
jose.nome = "NovoNome";
jose.idade = 16;
console.log('Idade Jose:', jose.idade);
console.log('Nome Jose:', jose.nome);
console.log(jose);
```

```
Idade Incorrecta!
Idade Jose: 16
Nome Jose: NovoNome
▼ Pessoa {morada: "Rua do José", #_idade: 16, #nome: "NovoNome", infoPessoa: f}
  ► infoPessoa: () => `${this.nome}-${this.idade}`
  morada: "Rua do José"
  #_idade: 16
  #nome: "NovoNome"
  idade: (...)
  nome: (...)
  ► [[Prototype]]: Object
    ► constructor: class Pessoa
    idade: (...)
    nome: (...)
    ► get idade: f idade()
    ► set idade: f idade(i)
    ► get nome: f nome()
    ► set nome: f nome(nome)
    ► [[Prototype]]: Object
```

```
jose.#_idade = 25;
```

```
Uncaught SyntaxError: Private field '#_idade' must be declared in an enclosing class
Atual: 35:65:15)
```

> class > Métodos Privados

```
class Pessoa {
  constructor(nome, morada, idade) {
    this.nome = nome;
    this.morada = morada;
    this.idade = idade;
  }
  infoPessoa = () => `${this.nome}-${this.idade}`;
  #metodoPrivado() {
    return "...método privado...";
  }
}
const jose = new Pessoa("José", "Rua do José", 153);
console.log('Jose:', jose.infoPessoa());
console.log('Jose:', jose.metodoPrivado());

console.log('Jose:', jose.#metodoPrivado());
```

Jose: José-153

Uncaught TypeError: jose.metodoPrivado is not a function
at scriptAwla.js:52:27

Uncaught SyntaxError: Private field '#metodoPrivado' must



> class > Propriedades Estáticas

```
class Pessoa {
  static #idadeAdulto = 18;
  constructor(nome, morada, idade) {
    this.nome = nome;
    this.morada = morada;
    this.idade = idade;
  }
  infoPessoa = () => `${this.nome}-${this.idade}`;
  get isMaiorIdade() {
    console.log(this.#metodoPrivado())
    return this.idade >= Pessoa.#idadeAdulto;
  }
  #metodoPrivado() {
    return "...método privado...";
  }
}
```



> class > Propriedades Estáticas

```
const jose = new Pessoa("José", "Rua do José", 153);
const maria = new Pessoa("Maria", "Rua da Maria", 17);

console.log("Jose Maior idade? " + jose.isMaiorIdade)
console.log(jose);
console.log("Maria Maior idade? " + maria.isMaiorIdade)
console.log(maria);
```

```
....método privado...
Jose Maior idade? true
▶ Pessoa {nome: 'José', morada: 'Rua do José', idade: 153, #metodoPrivado: f, infoPessoa: f}
....método privado...
Maria Maior idade? false
▶ Pessoa {nome: 'Maria', morada: 'Rua da Maria', idade: 17, #metodoPrivado: f, infoPessoa: f}
```

Qual abordagem optar?

- › *Constructor Function ?*
- › *Factory Function ?*
- › *Class ?*



</Orientação a Objetos>

