

Orientação a Objetos

JavaScript

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


Departamento de Engenharia Informática e de Sistemas

Cristiana Areias < cris@isec.pt >

2023/2024

JavaScript

Orientação a Objetos

- › Orientação a Objetos em JavaScript 
- › *Criação de Objectos*
 - › *Factory Functions*
 - › *Construtor Functions*
 - › *Classes*
- › *Prótipos*

> 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

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

Membros
nome
morada
idade
informacao

Propriedades
nome
morada
Idade
Define o estado

Método
informacao
O membro é uma função
Define comportamento
e contém alguma
Lógica

> 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.

- **Class**

> 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 Constructor Function Class

Criação de vários objetos

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

- **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

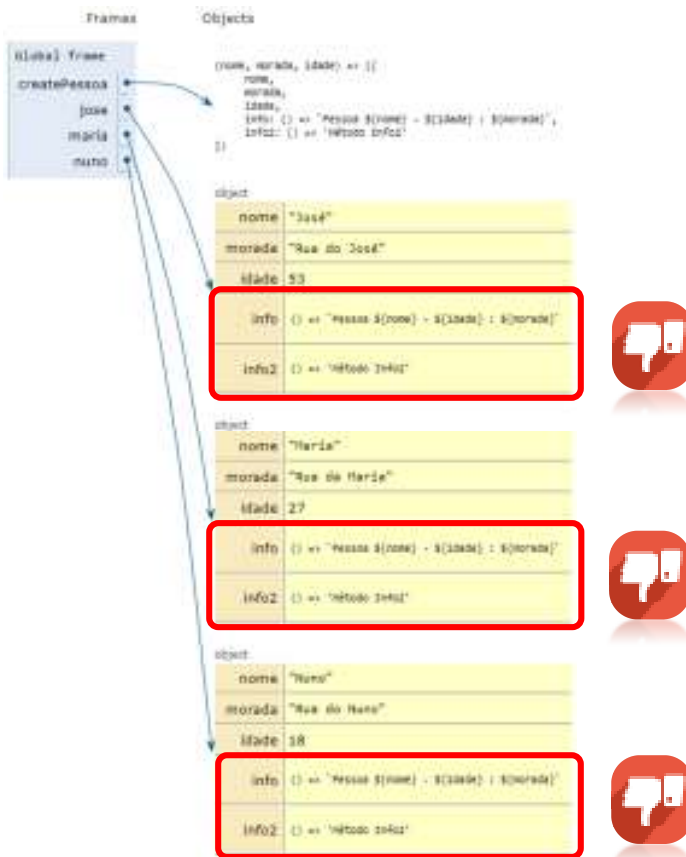
```
const createPessoa = (nome, morada, idade) => ({  
  nome,  
  morada,  
  idade,  
  info: () => `Pessoa ${nome} - ${idade} : ${morada}`,  
  info2: () => 'Método Info2'  
});  
  
const jose = createPessoa("José", "Rua do José", 53);  
const maria = createPessoa("Maria", "Rua da Maria", 27);  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



Mais adiante...

Prótipos

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

> 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!

↓

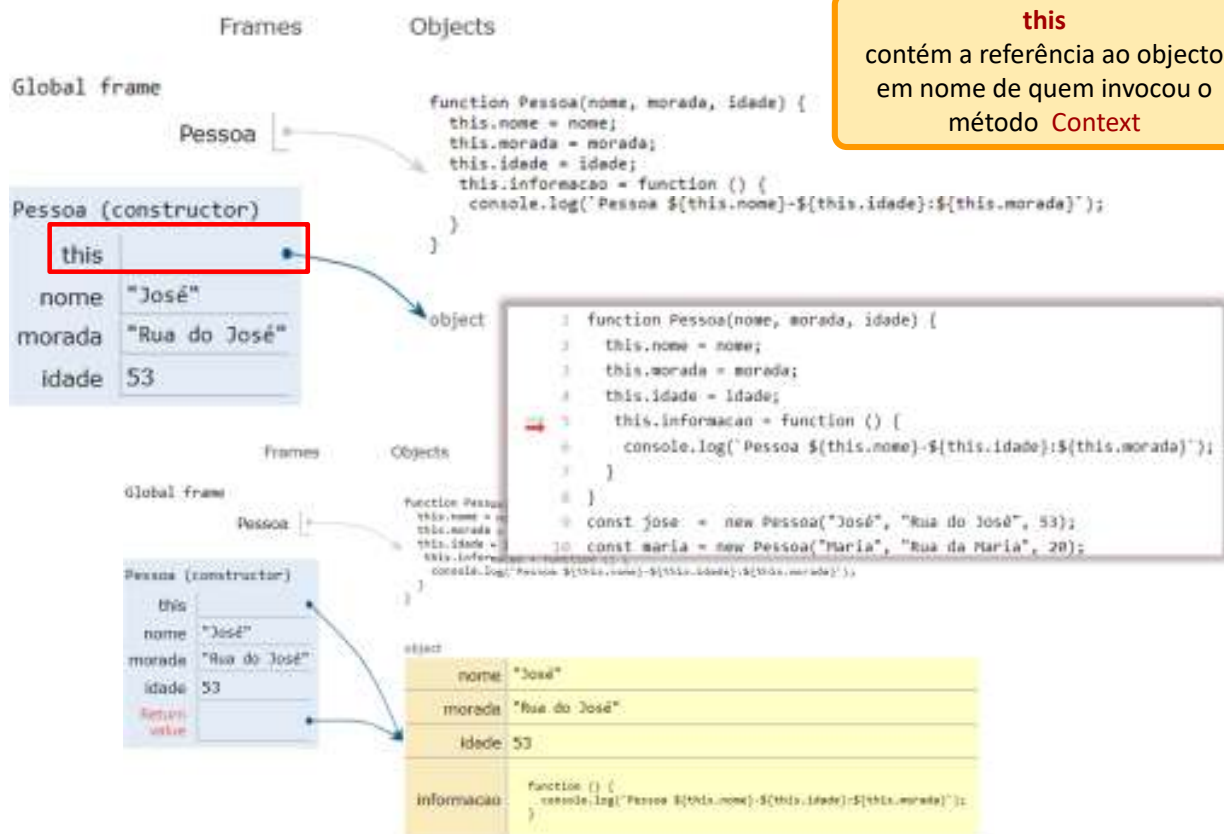
✓

> Construtor Function - new

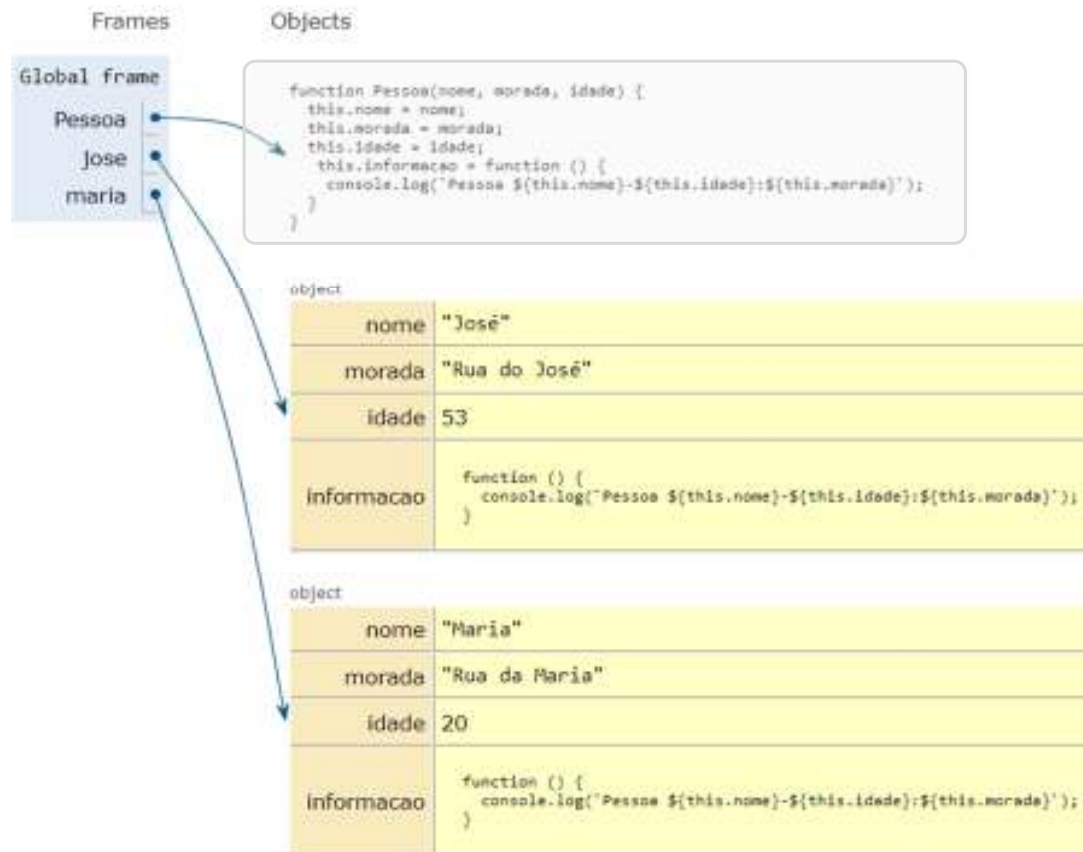
```
function Pessoa(nome, morada, idade) {  
    this.nome = nome;  
    this.morada = morada;  
    this.idade = idade;  
}  
  
const jose = new Pessoa("José", "Rua do José", 53);
```

- 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}
  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: "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();
```



Solução
não escalável...
Object.create()

> 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 behavior 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, 2023*

- 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;  
}
```

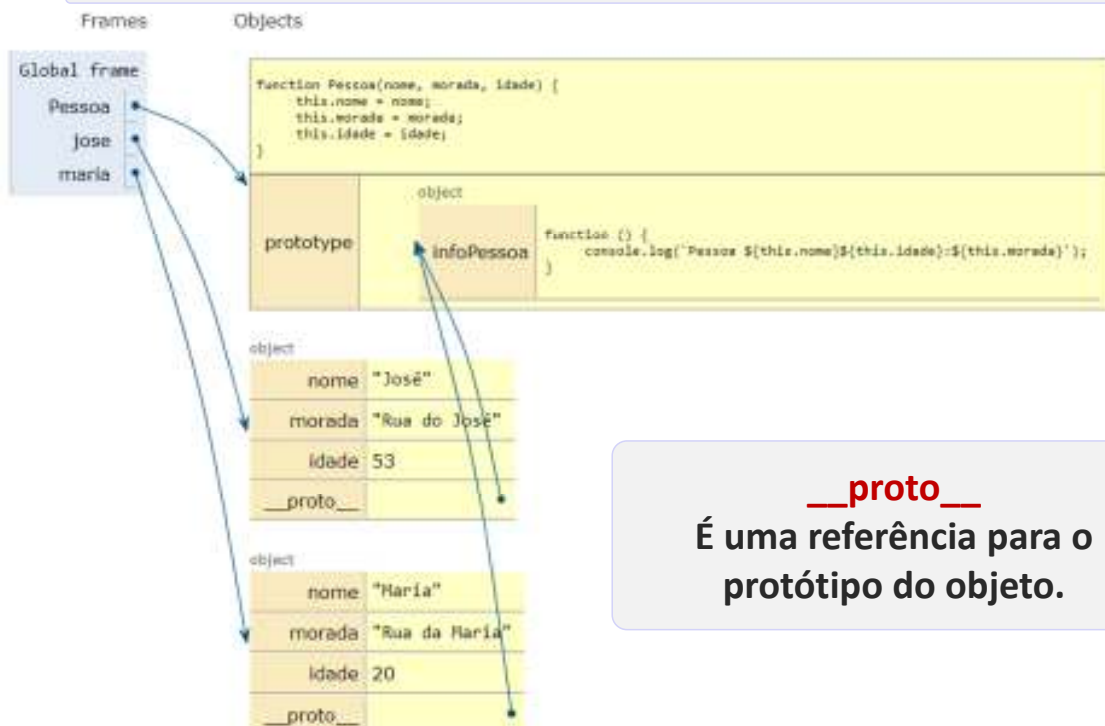
```
*Pessoa {nome: 'José', morada: 'Rua do José', idade: 53}  
  idade: 53  
  morada: "Rua do José"  
  nome: "José"  
  → [[Prototype]]: Object  
  → infoPessoa: f ()  
  → constructor: f Pessoa(nome, morada, idade)  
  → [[Prototype]]: Object
```

```
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.

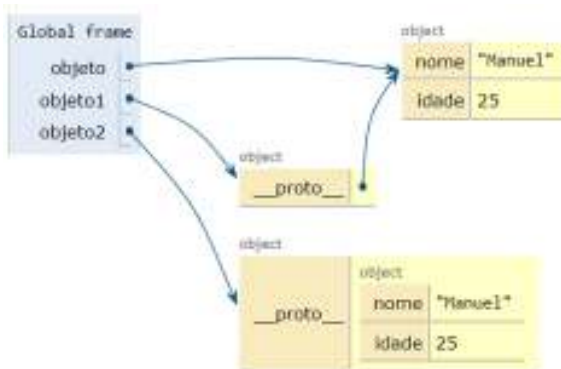
> Prototype > create()

- 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`);
```

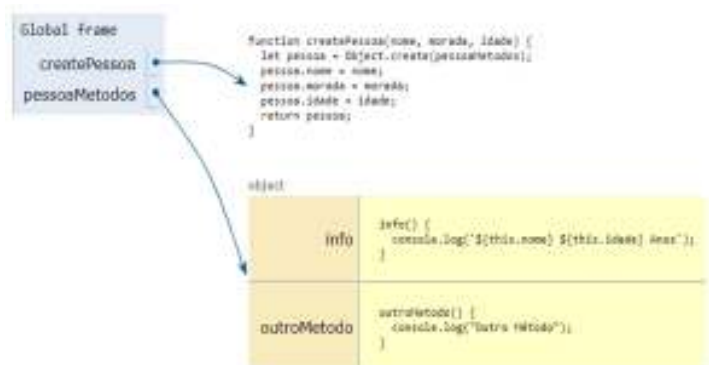
```
  },
```

```
  outroMetodo() {
```

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

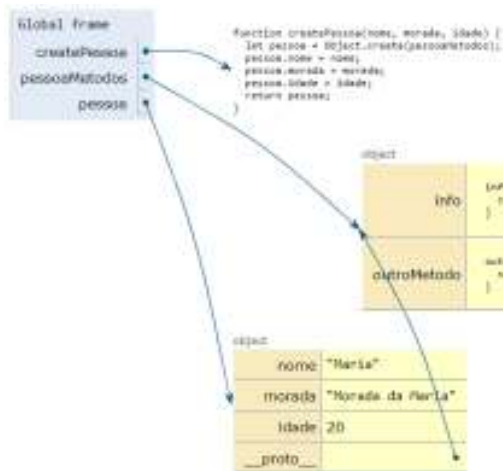
```
  }
```

```
};
```



> Prototype > create()

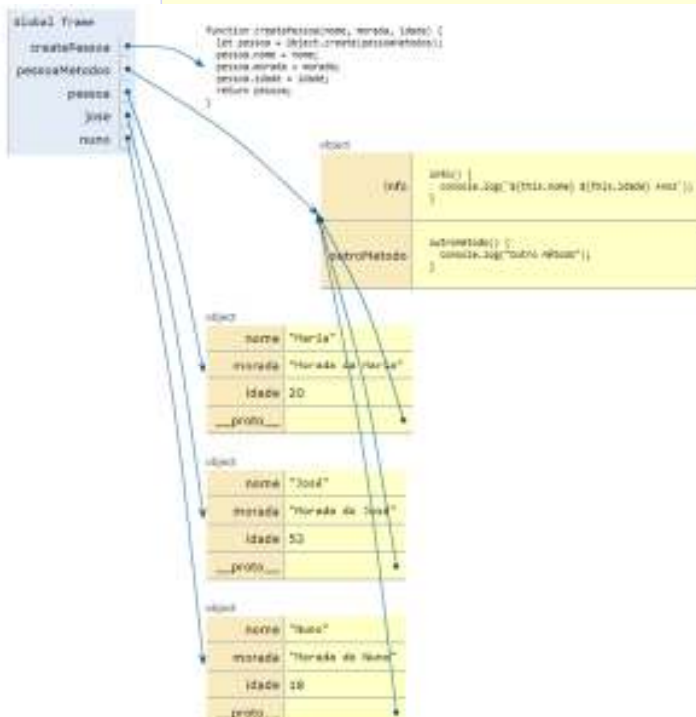
```
let pessoa = createPessoa("Maria", "Morada da Maria", 20);
console.log("Pessoa", pessoa)
pessoa.info();
pessoa.outroMetodo();
```



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



> Prototype > create()

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



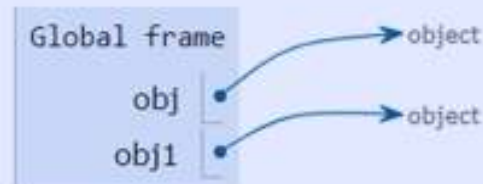
No properties



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*



< 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;  
  }
```

Semelhante ao
constructor
functions

```
  infoPessoa () {
```

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

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

```
}
```

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

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

```
jose.infoPessoa();
```

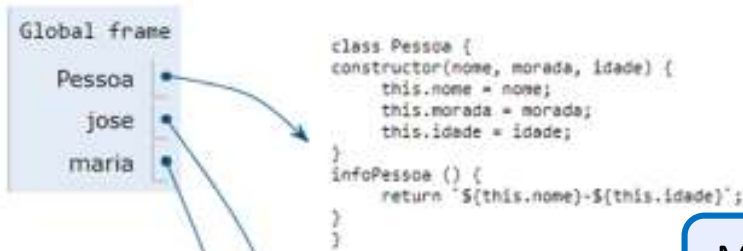
```
maria.infoPessoa();
```

> 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);
```



object

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

object

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

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



Objecto 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__);
```

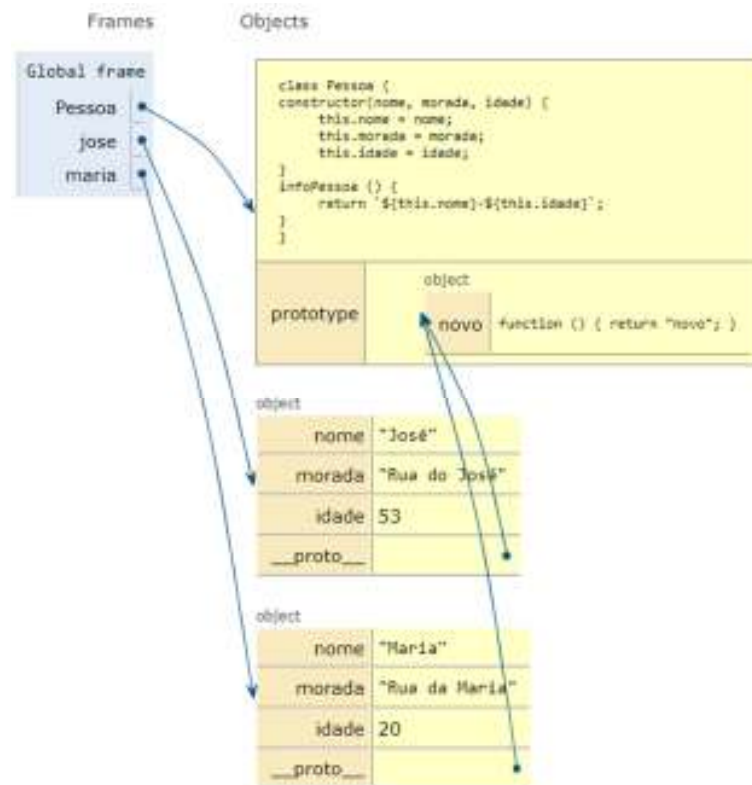


jose ▼ Pessoa {nome: "José", morada: "Rua do José", idade: 53} ⓘ
 idade: 53
 morada: "Rua do José"
 nome: "José"
 ▼ [[Prototype]]: Object
 ▶ novo: f ()
 ▶ constructor: class Pessoa
 ▶ infoPessoa: f infoPessoa()
 ▶ [[Prototype]]: Object

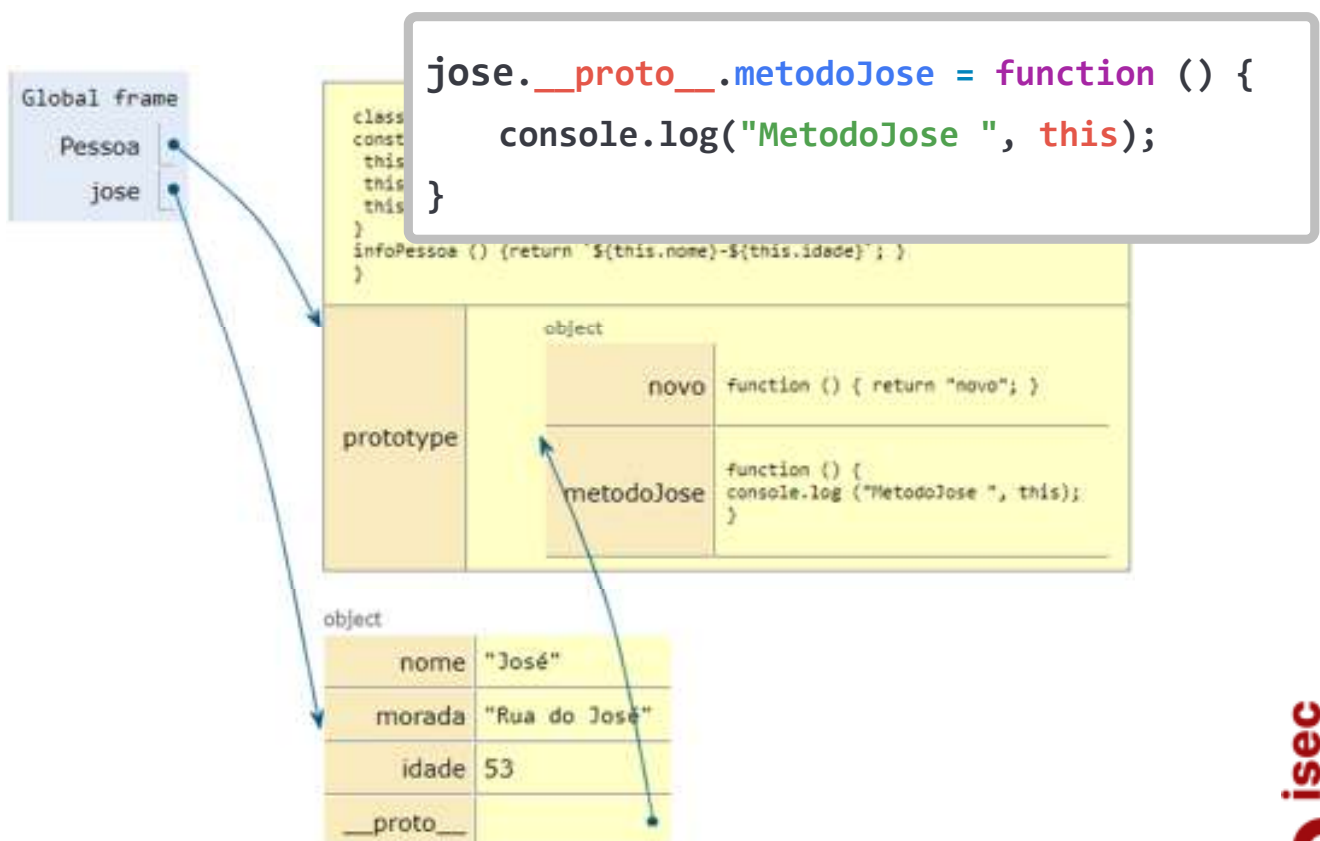
jose.__proto__ ▼ {novo: f, constructor: f, infoPessoa: f} ⓘ
 ▶ novo: f ()
 ▶ constructor: class Pessoa
 ▶ infoPessoa: f infoPessoa()
 ▶ [[Prototype]]: Object

> Class e Prototype

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

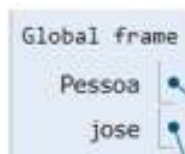


> class > prototype e __proto__

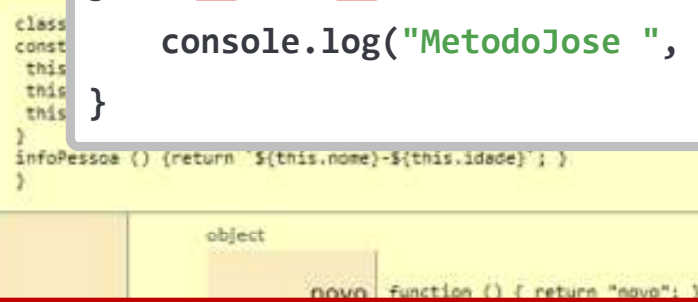


> class > prototype e __proto__

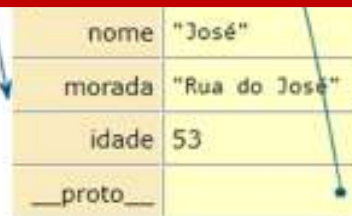
JavaScript



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

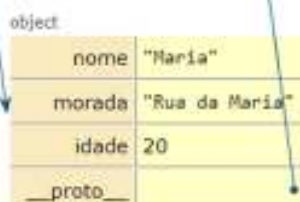
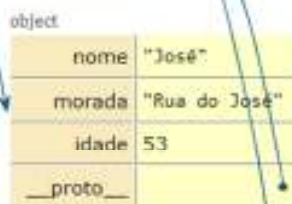
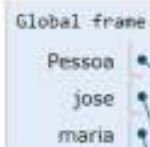


```
const maria = new Pessoa("Maria", "Rua da Maria", 20);  
maria.metodoJose(); // ?
```



> class > prototype e __proto__

JavaScript



```
MetodoJose ▾ Pessoa {nome: 'Maria', morada: 'Rua da Maria', idade: 20}  
  idade: 20  
  morada: "Rua da Maria"  
  nome: "Maria"  
  ▾ [[Prototype]]: Object  
    ▸ metodoJose: f ()  
    ▸ novo: f ()  
    ▸ constructor: class Pessoa  
    ▸ infoPessoa: f infoPessoa()  
    ▸ [[Prototype]]: Object
```

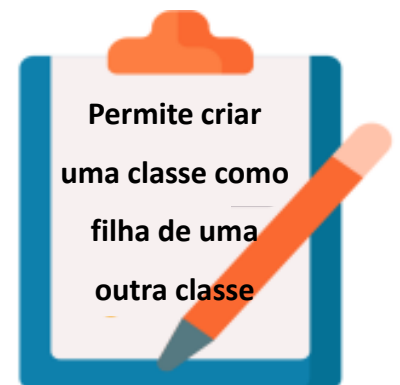

> class > extends

- As classes são simplificações da linguagem JS para as heranças baseadas em protótipos, no qual a implementação recorre 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

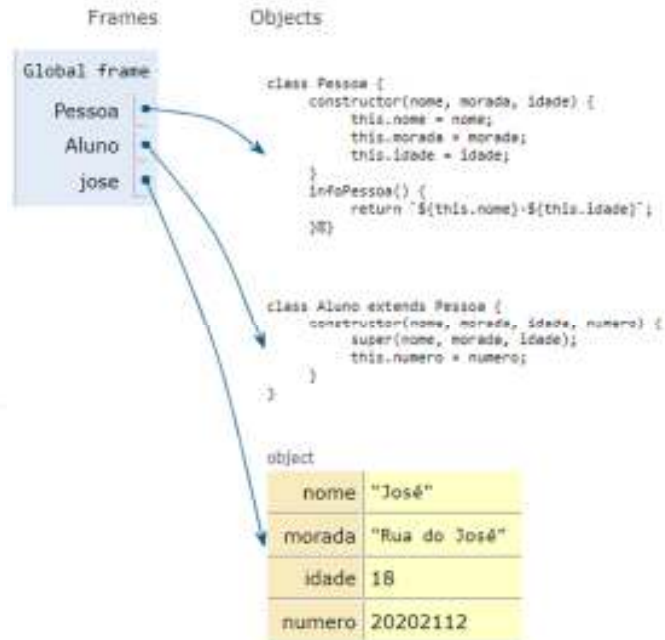
```
class Pessoa {
  constructor(nome, morada, idade) {
    this.nome = nome;
    this.morada = morada;
    this.idade = idade;
  }
  infoPessoa() {
    return `${this.nome}-${this.idade}`;
  }
}
class Aluno extends Pessoa {
  constructor(nome, morada, idade, numero) {
    super(nome, morada, idade);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



Qual abordagem optar?

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





Existe um conjunto de outros conceitos no contexto de POO em JavaScript, e muito importantes, tais como encapsulamento em JavaScript, getters e setters, métodos e propriedades privadas, propriedades estáticas, entre outros, que não serão abordados no contexto da UC de LS.

</Orientação a Objetos>

