

JavaScript

<Funções>

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

Departamento de Engenharia Informática e de Sistemas

Cristiana Areias < cris@isec.pt >

2022/2023

JavaScript

Funções

- › Tipos de Funções
 - › Declaração de Função
 - › *Function Expression (anonymous function)*
 - › *Arrow Functions*
 - › *Functions Constructor*
- › Invocação de Funções
- › Exemplos

> Funções

- Bloco de código, conjunto de *statements*, que permitem a execução de uma tarefa específica ou cálculo de um valor;
- Funções em JavaScript são também conhecidas como **objetos de primeira classe (first-class objects)**.
 - Tudo o que se faz com um objecto, é possível fazer com funções.
 - Uma função é um objeto do tipo **Function**.
- Este tipo especial de **objeto** tem capacidade adicional de poder ser invocada
- As funções em JavaScript são executadas na sequência em que são invocadas, não na sequência em que são definidas.

> Funções > Benefícios

- Reutilização de código;
- Torna o programa mais simples, permitindo que as tarefas possam ser divididas por várias funções;
- Aumenta a legibilidade;

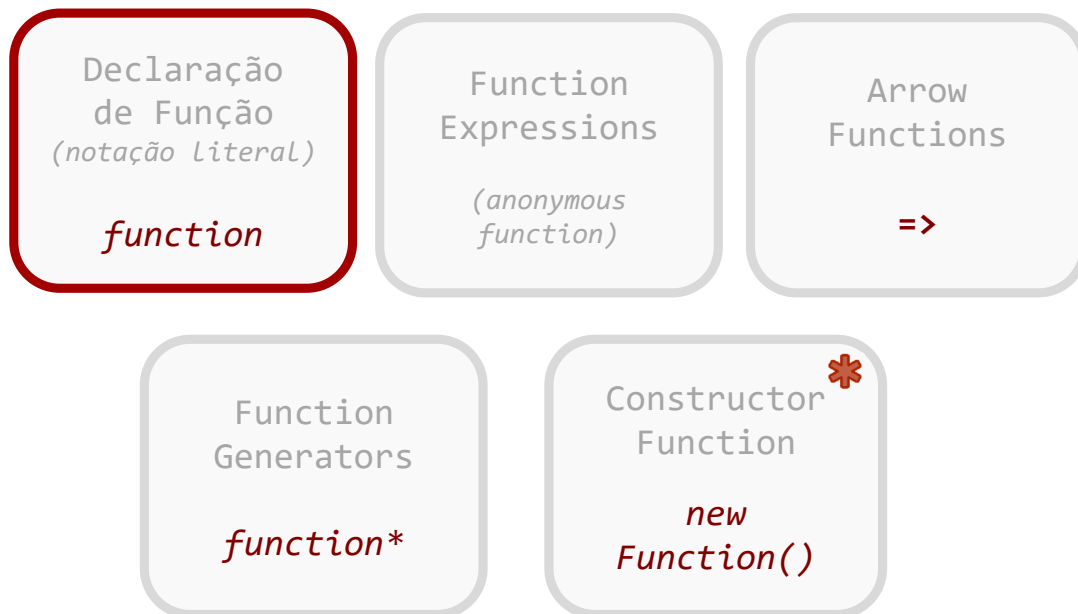


```
funcaoOla('José');  
funcaoOla('Maria', 'Pereira');
```

```
function funcaoOla(nome, apelido) {  
    console.log('Olá ' + nome + ' ' + apelido + '!');  
}
```

> Funções > Criação

- Existem diferentes formas de se criar e invocar uma função em JavaScript



> Funções > Declaração de Funções

- A **declaração** inicia com a palavra-reservada **function** seguida de:
 - Nome** da função e lista de **argumentos** para a função, entre parênteses e separados por vírgulas;
 - Declarações que definem a função entre `{ }`
 - Podem ser invocadas antes ou depois de serem definidas no código!

```
function funcao01a() {  
  console.log('Olá!');  
}  
funcao01a();
```

```
funcao01a();  
function funcao01a() {  
  console.log('Olá!');  
}
```

**Function
hoisting**

```
function funcao01a(nome) {  
  console.log('Ola '+nome+'!');  
}  
funcao01a('José');  
funcao01a('Maria');
```

<Parâmetros>

> Parâmetros

> Parâmetros em funções

> Parâmetros por omissão (default)

> Argumentos

< 136 >

> Funções > Parâmetros e Argumentos

- Parâmetros são variáveis específicas à função

```
function funcaoOla(nome, apelido = '') {
  console.log('Ola ' + nome + ' ' + apelido + '!');
}
```

```
funcaoOla('José');
funcaoOla('Maria', 'Pereira');
```

- Tipos de funções

- Implementar uma tarefa

- Calcular um valor

```
function dobro(num) {
  return num * 2;
}
console.log(dobro(5));
```

Quantas chamadas a funções existem nesta linha de código?



> Funções > Parâmetros por Omissão

- Em JavaScript, os parâmetros de função são, por omissão, *undefined*.
- Muitas vezes, é necessário definir um valor diferente podendo ser usados parâmetros *default*
- Simplificam a necessidade de verificar no corpo da função, se o parâmetro é ou não *undefined*!

```
function verifica(num = 5) {  
    console.log(typeof num);  
    console.log(num)  
}  
verifica();  
verifica(undefined);  
verifica('');  
verifica(null);
```

number
5
number
5
string
object
null

> Funções > Parâmetros por Omissão

```
function funcaoOla(nome, apelido) {  
    console.log('Olá ' + nome + ' ' + apelido + '!');  
}  
funcaoOla('José');  
funcaoOla('Maria', 'Pereira');
```

```
Olá José undefined!  
Olá Maria Pereira!
```

```
function funcaoOla(nome, apelido) {  
    apelido=typeof apelido!=='undefined' ? apelido : '';  
    console.log('Olá ' + nome + ' ' + apelido + '!');  
}
```

```
Olá José !  
Olá Maria Pereira!
```

> Funções > Parâmetros por Omissão

Simplificando...

pos-ECMAScript 2015

```
function funcaoOla(nome, apelido = '') {  
    console.log('Olá ' + nome + ' ' + apelido + '!');  
}
```

```
function funcaoOla(nome, apelido='', nCompleto=nome+' '+apelido)  
{  
    console.log('Olá ' + nCompleto + '!');  
}  
  
funcaoOla('José');  
funcaoOla('Maria', 'Pereira');
```

```
Olá José !  
Olá Maria Pereira!
```



> Funções > Parâmetros e Argumentos

```
function funcao(a, b) {  
    console.log(`Função! a = ${a}`);  
    console.log(`Parametros =${a} e ${b}`);  
}  
  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```

```
Função! a = undefined  
Parametros =undefined e undefined  
Função! a = 1  
Parametros =1 e 3  
Função! a = 5  
Parametros =5 e 6  
Função! a = 7  
Parametros =7 e undefined
```



> Funções > Parâmetros e Argumentos

```
function dobro(num) {  
    return num * 2;  
}
```

```
console.log(dobro);
```

```
console.log(dobro());
```

```
console.log(dobro(2));
```

```
console.log(dobro(2,3));
```

```
f dobro(num) {  
    return num * 2;  
}
```

NaN

4

4

> Funções > Parâmetros e Argumentos

```
let numero = 5;  
function contador(num) {  
    num++;  
}  
contador(numero);  
console.log(numero);
```

5

```
let numero = { valor: 5 };  
function contador(num) {  
    num.valor++;  
}  
contador(numero);  
console.log(numero);
```

{valor: 6}



> Funções > Parâmetros e Argumentos

```
function completaNome(objeto) {  
    objeto.nome = "José Maria";  
}  
  
let pessoa = { nome: "José", morada: "Carlos Seixas"};  
let pX, pY;  
pX = pessoa.nome;  
completaNome(pessoa);  
pY = pessoa.nome;  
  
console.log(`pX = ${pX} py=${pY}`);
```

pX = José py=José Maria



> Funções > Parâmetros e Argumentos

- Parâmetros primitivos (como um número) são passados para as funções por **valor**
 - O valor é passado para a função, mas se a função altera o valor do parâmetro, esta mudança não reflete globalmente ou na função chamada.
- Ao passar um **objeto** como um parâmetro e a função alterar as propriedades do objeto, essa **mudança é visível fora da função**.

Primitivos são
copiados pelo
seu **valor**!

Objectos são
copiados pela
sua **referência**!



> Funções > Parâmetros e Argumentos

```
function novoElemento(array, texto) {  
    array.push(texto)  
    array[array.length] = texto;  
}  
  
let frutas = ["Banana", "Laranja", "Maça", "Pera"];  
  
console.table(frutas);  
  
novoElemento(frutas, 'Pessego!');  
  
console.table(frutas);
```

(index)	Value
0	"Banana"
1	"Laranja"
2	"Maça"
3	"Pera"
4	"Pessego!"
5	"Pessego!"

> Funções > Parâmetros e Argumentos

```
function funcao() {  
    console.log('Olá na função! Argumento[0] = ' +  
        arguments[0]);  
    console.log('Argumentos =' + Array.from(arguments));  
}  
  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```



Obsoleto!

```
Olá na função! Argumento[0] = undefined  
Argumentos =  
Olá na função! Argumento[0] = 1  
Argumentos =1,3,4  
Olá na função! Argumento[0] = 5  
Argumentos =5,6  
Olá na função! Argumento[0] = 7  
Argumentos =7
```

> Funções > Parâmetros e Argumentos

... *Rest parameters*

```
function funcao(...valores) {  
  console.log('Olá na função! Argumento[0] = '+valores[0]);  
  console.log('Argumentos ='+ valores);  
}  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```

```
Olá na função! Argumento[0] = undefined  
Argumentos =  
Olá na função! Argumento[0] = 1  
Argumentos =1,3,4  
Olá na função! Argumento[0] = 5  
Argumentos =5,6  
Olá na função! Argumento[0] = 7  
Argumentos =7
```

> Funções > Parâmetros e Argumentos

... *Rest parameters*

```
function funcao(a, b = 0, ...valores) {  
  console.log('Olá na função! a + b = ' + (a + b));  
  console.log('...valores = ' + valores);  
}  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```

```
Olá na função! a + b = NaN  
...valores =  
Olá na função! a + b = 4  
...valores = 4  
Olá na função! a + b = 11  
...valores =  
Olá na função! a + b = 7  
...valores =
```

> Funções > Parâmetros e Argumentos

JavaScript

```
function append(valor, array = []) {  
    array.push(valor)  
    return array  
}  
  
let array = append(1)  
console.log(array);  
array = append(2);  
console.log(array);  
array = append(3, array);  
console.log(array);
```

```
▶ [1]  
▶ [2]  
▶ (2) [2, 3]
```



> Funções > Parâmetros e Argumentos

JavaScript

```
function completaNome(objeto) {  
    objeto.nome = "José Maria";  
}  
  
let pessoa = { nome: "José", morada: "Carlos Seixas"};  
let pX, pY;  
pX = pessoa.nome;  
completaNome(pessoa);  
pY = pessoa.nome;  
  
console.log(`pX = ${pX} py=${pY}`);
```

```
pX = José py=José Maria
```



E funções com mesmo nome?

› Overloading em JavaScript ?



< 152 >

> Funções > Várias com mesmo Nome

```
function funcao(param1, param2, param3) {  
    console.log(`Olá na função com 3 parametros!! ${param1}  
    ${param2} ${param3}`);  
}  
  
function funcao(param1, param2) {  
    console.log(`Olá na função com 2 parâmetros! ${param1}  
    ${param2}`);  
}  
  
function funcao() {  
    console.log('Olá na função!');  
}  
  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```



> Funções > Várias com mesmo Nome

```
function funcao(param1, param2, param3) {  
    console.log(`Olá na função com 3 parametros!! ${param1}  
    ${param2} ${param3}`);  
}  
function funcao(param1, param2) {  
    console.log(`Olá na função com 2 parâmetros! ${param1}  
    ${param2}`);  
}  
function funcao() {  
    console.log('Olá na função!');  
}  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```

Olá na função!
Olá na função!
Olá na função!
Olá na função!

> Funções > Várias com mesmo Nome

```
function funcao(param1, param2, param3) {  
    console.log(`Olá na função com 3 parametros!! ${param1}  
    ${param2} ${param3}`);  
}  
function funcao() {  
    console.log('Olá na função!');  
}  
function funcao(param1, param2) {  
    console.log(`Olá na função com 2 parâmetros! ${param1}  
    ${param2}`);  
}  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```



> Funções > Várias com mesmo Nome

```
function funcao(param1, param2, param3) {  
    console.log(`Olá na função com 3 parametros!! ${param1}  
    ${param2} ${param3}`);  
}  
function funcao() {  
    console.log('Olá na função!');  
}  
function funcao(param1, param2) {  
    console.log(`Olá na função com 2 parâmetros! ${param1}  
    ${param2}`);  
}  
funcao();  
funcao(1, 3, 4);  
funcao(5, 6);  
funcao(7);
```

Olá na função com 2 parâmetros! undefined
undefined
Olá na função com 2 parâmetros! 1 3
Olá na função com 2 parâmetros! 5 6
Olá na função com 2 parâmetros! 7 undefined

> Funções > Variáveis com mesmo Nome

```
let funcao = 3;  
function funcao() {  
    console.log('Olá na função!');  
}  
funcao();
```

Uncaught SyntaxError: Identifier 'funcao' has already been declared

```
funcao = 3;  
function funcao() {  
    console.log('Olá na função!');  
}  
funcao();
```

Uncaught TypeError: funcao is not a function

```
console.log(funcao);
```

3

> Funções > Variáveis com mesmo Nome

```
function funcao() {  
    console.log('Olá na função!');  
}
```

```
let funcao = 3;
```

```
funcao();
```

Uncaught SyntaxError: Identifier 'funcao' has already been declared

```
function funcao() {  
    console.log('Olá na função!');  
}
```

```
funcao = 3;
```

```
funcao();
```

Uncaught TypeError: funcao is not a function

```
console.log(funcao);
```

3

> Funções > *Overloading*

- *Overloading* é a **capacidade de conter duas funções com mesmo nome** distinguíveis pelo contexto.
- JavaScript é fracamente tipado e por padrão não suporta *overloading* por não conseguir distinguir as assinaturas de métodos com mesmo identificador pelo tipo de dados dos argumentos passados.
- Diferente de outras linguagens de programação como o C#, Java, ... o JavaScript não suporta **Function Overloading**
- Funções com o mesmo nome provoca um **Overriding** da função



JavaScript
não suporta
Function Overloading

Funções *CaLLBack*?



< 160 >

> Funções como argumentos

```
function funcaoOla() {  
    console.log('Olá na Função!');  
    return "Ola";  
}
```

```
function funcaoNome(nomeUtilizador, func) {  
    const mensagem = func();  
    console.log(`${mensagem} ${nomeUtilizador}`);  
}
```

```
funcaoNome('José', funcaoOla);  
funcaoNome('Maria', funcaoOla);  
funcaoNome('Nuno', funcaoOla);
```

```
Olá na Função!  
Ola José  
Olá na Função!  
Ola Maria  
Olá na Função!  
Ola Nuno
```


> Funções como argumentos

```
function funcaoOla(a) {  
    console.log('Olá na Função!');  
    if (a === 1) return "Olá! Seja bem vinda "  
    return "Olá! Seja bem vindo "  
}
```

```
function funcaoNome(nomeUtilizador, func) {  
    const mensagem = func;  
    console.log(`${mensagem}${nomeUtilizador}`);  
}  
  
funcaoNome('José', funcaoOla());  
funcaoNome('Maria', funcaoOla(1));  
funcaoNome('Nuno', funcaoOla(2));
```

```
Olá na Função!  
Olá! Seja bem vindo José!  
Olá na Função!  
Olá! Seja bem vinda Maria!  
Olá na Função!  
Olá! Seja bem vindo Nuno!
```

> Funções *Callbacks*

- Um **callback** é uma função passada como argumento para outra função.

```
function printConsola(msg) {  
    console.log("printConsola = " + msg);  
}  
  
function calcSoma(n1, n2) {  
    return n1 + n2;  
}  
  
let soma = calcSoma(10, 5);  
printConsola(soma);
```

printConsola = 15

```
function printConsola(msg) {  
    console.log("printConsola = " + msg);  
}  
  
function calcSoma(n1, n2) {  
    let soma = n1 + n2;  
    printConsola(soma);  
}  
  
calcSoma(10, 5);
```

**Desvantagens
destas duas
abordagens?**



> Funções *Callbacks*

- Recorrendo a uma função *callback*, é possível chamar a função `calcSoma` com um *callback*, e deixar a função efetuar o *callback* depois do calculo estar concluído;

```
function printConsola(msg) {  
    console.log("printConsola = " + msg);  
}  
function calcSoma(n1, n2, func) {  
    let soma = n1 + n2;  
    func(soma);  
}  
calcSoma(10, 5, printConsola);
```

- Um função *callback* pode executar depois de outra função ter terminado;
- Os *callbacks* são excelentes opções em funções assíncronas, onde uma função tem de esperar por outra para ser executada (por exemplo, quando se obtém dados de outro servidor).

JavaScript

Nested Functions



> Funções Aninhadas (*nested*)

```
let pessoa = { nome: "José", morada: "Carlos Seixas" };
let completaNome;
let num = 0;
if (num == 0) {
    completaNome = function (objeto) {
        objeto.nome = "Jose Maria"
    }
}
completaNome(pessoa);
console.log(`Nome = ${pessoa.nome}`)
```

Nome = Jose Maria

> Funções Aninhadas (*nested*)

- Funções aninhadas permitem organização de código

```
function funcaoOla(primeiroNome, ultimoNome = '') {
    function getNomeCompleto() {
        return primeiroNome + " " + ultimoNome;
    }

    console.log("Olá, " + getNomeCompleto());
}
funcaoOla("Nuno", "Afonso");
getNomeCompleto()
```

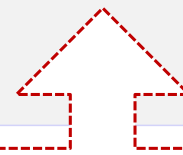
Olá, Nuno Afonso

✖ ▶ Uncaught ReferenceError:
getNomeCompleto is not defined

> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa: function () {  
    console.log("Pessoa faz Qualquer coisa!");  
  }  
}  
pessoa.fazQualquerCoisa();
```



Método

Propriedade que contém a declaração de uma função



> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa () { // Versão ES6  
    console.log("Pessoa faz Qualquer coisa!");  
  }  
}  
pessoa.fazQualquerCoisa();  
  
pessoa.fazQualquerCoisa;
```



> Funções > Objetos > Métodos

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa: function () {  
    console.log("Cria novo método!");  
  }  
}
```

Cria novo método!

```
{  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  corOlhos: 'Castanhos',  
  fazQualquerCoisa: f, ...  
}
```

corOlhos: "Castanhos"

▶ fazOutraCoisa: f ()

▶ fazQualquerCoisa: f ()

idade: "45"

morada: "Rua Carlos Seixas"

nome: "Manuel Afonso"

▶ [[Prototype]]: Object

```
pessoa.corOlhos = "Castanhos";  
pessoa.fazOutraCoisa = function () {  
  console.log("Cria novo método!");  
}
```

```
pessoa.fazOutraCoisa();  
console.log(pessoa);
```

> Funções > Objetos > Métodos


```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa() {  
    console.log("Pessoa faz Qualquer coisa!");  
  },  
  infoPessoa() {  
    console.log(`Informações de '${nome}' \nMorada:  
    ${morada}\nIdade: ${idade}`);  
  }  
}
```

```
pessoa.infoPessoa();
```

> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa() {  
    console.log("Pessoa faz Qualquer coisa!");  
  },  
  infoPessoa() {  
    console.log(`Informações de '${nome}' \n morada  
    ${morada}\nIdade: ${idade}`);  
  }  
}
```

A red-bordered box containing a red 'x' icon and the text 'Uncaught ReferenceError: nome is not defined'.


```
pessoa.infoPessoa();
```



> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45',  
  fazQualquerCoisa() {  
    console.log("Pessoa faz Qualquer coisa!");  
  },  
  infoPessoa() {  
    console.log(`Informações de '${nome}' \n morada  
    ${morada}\nIdade: ${idade}`);  
  }  
}
```

A red-bordered box containing a red 'x' icon and the text 'Uncaught ReferenceError: nome is not defined'.

```
pessoa.infoPessoa();
```



> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: 45  
}  
  
pessoa.infoPessoa() {  
  console.log(`Informações de '${pessoa.nome}'\n  
  Morada: ${pessoa.morada}\nIdade: ${pessoa.idade}`);  
}
```



E se copiar este código?

Ou

```
let admin = pessoa;  
pessoa=null;
```

• uncaught TypeError: Cannot read properties of null (reading 'infoPessoa')

isec
Engenharia

> Funções > Objetos > Métodos

JavaScript

```
let pessoa = {  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: 45  
}  
  
pessoa.infoPessoa() {  
  console.log(`Informações de '${this.nome}'\n  
  Morada: ${this.morada}\nIdade: ${this.idade}`);  
}
```

Palavra reservada

this

corresponde ao
"objecto corrente"

isec
Engenharia

> Funções > Métodos > this

- **this** é uma palavra-chave, como uma variável especial, que permite aceder a propriedades do objecto.

`this.propriedade;`

- O valor de **this** é dinâmico!
 - O valor do **this** é definido em tempo de execução
 - **this** refere-se a diferentes elementos, diferentes objectos
 - Depende do local/função onde é chamada
 - Localizado num método de um objecto, refere-se ao objecto.
 - Sozinho, refere-se ao objeto global.
 - Numa função, **em modo restrito**, é *undefined*, senão refere-se objeto global.
 - Na definição de um evento, refere-se ao elemento que recebeu o evento.
 - Assunto a tratar mais tarde.

> Funções > Métodos > this

```
const retangulo = {
  lado: 12,
  altura: 4,
  desenha: function () {
    console.log("Desenha Retangulo!");
  },
  calculaArea: function () { // Pré ES6
    return this.lado * this.altura;
  },
  calculaPerimetro() { // Pós ES6
    return this.lado * 2 + this.altura * 2;
  }
}

retangulo.desenha();
console.log(retangulo.calculaArea());
console.log(retangulo.calculaPerimetro());
```

Desenha Retangulo!

48

32

> Funções > Métodos > this

```
let professor = { nome: "Professor José Afonso" };
let aluno = { nome: "Aluno Ricardo Manuel" };
function funcao01a() {
    console.log(this.nome);
}
professor.f = funcao01a;
aluno.f = funcao01a;

professor.f();
aluno.f();

aluno['f']();
```

Professor José Afonso
Aluno Ricardo Manuel
Aluno Ricardo Manuel

> Funções > Métodos > this

```
const aluno1 = {
    nome: 'Nuno Afonso',
    numero: 2102124,
    anoNascimento: 2000,
    disciplinasConcluidas: [
        { uc: 'AP', nota: 12 }, { uc: 'LS', nota: 15 },
        { uc: 'AP', nota: 13 }, { uc: 'TAC', nota: 15 }
    ],
    calcIdade() {
        this.idade = 2021 - this.anoNascimento;
        return this.idade;
    },
    media: function () {
        let soma = 0;
        for (let i = 0; i < this.disciplinasConcluidas.length; i++)
            soma = soma + this.disciplinasConcluidas[i].nota;
        this.media = soma / this.disciplinasConcluidas.length;
        return this.media;
    }
}
```

Nota: O cálculo da idade apenas está a considerar o ano. Cálculo correcto deveria considerar também dia e mês.



> Funções > Métodos > **this**

JavaScript

```
const aluno1 = {
  nome: 'Nuno Afonso',
  numero: 2102124,
  anoNascimento: 2000,
  disciplinasConcluidas: [
    { uc: 'AP', nota: 12 },
    { uc: 'AP', nota: 13 },
  ],
  calcIdade() {
    this.idade = 2024 - this.anoNascimento;
    return this.idade;
  },
  media: function() {
    let soma = 0;
    for (let i = 0; i < this.disciplinasConcluidas.length; i++) {
      soma = soma + this.disciplinasConcluidas[i].nota;
    }
    this.media = soma / this.disciplinasConcluidas.length;
    return this.media;
  }
}
```

```
console.log(typeof aluno1);
console.log(aluno1.calcIdade());
console.log(aluno1.idade);
console.log(aluno1.media);
```

```
object
21
21
f () {
  let soma = 0;
  for (let i = 0; i < this.disciplinasConcluidas.length; i++) {
    soma = soma + this.disciplinasConcluidas[i].nota;
  }
  this.media = soma / this.disciplinasConcluidas.length;
  return this.media;
}
```

isec
Engenharia

> Funções > Métodos > **this**

JavaScript

```
const aluno1 = {
  nome: 'Nuno Afonso',
  numero: 2102124,
  anoNascimento: 2000,
  disciplinasConcluidas: [
    { uc: 'AP', nota: 12 },
    { uc: 'AP', nota: 13 },
    { uc: 'TAC', nota: 15 }
  ],
  calcIdade() {
    this.idade = 2024 - this.anoNascimento;
    return this.idade;
  },
  media: function() {
    let soma = 0;
    for (let i = 0; i < this.disciplinasConcluidas.length; i++) {
      soma = soma + this.disciplinasConcluidas[i].nota;
    }
    this.media = soma / this.disciplinasConcluidas.length;
    return this.media;
  }
}
```

```
console.log(aluno1.media());
console.log(aluno1.media);
console.log(aluno1);
console.log(aluno1.media());
```

```
13.75
13.75
{nome: 'Nuno Afonso', numero: 2102124, anoNascimento: 2000, disciplinasConcluidas: (4) [{-}, {-}, {-}, {-}], calcIdade: f, media: f}
  calcIdade: f calcIdade(ano)
  disciplinasConcluidas: (4) [{-}, {-}, {-}, {-}]
  media: 13.75
  nome: "Nuno Afonso"
  numero: 2102124
  [[Prototype]]: Object
Uncaught TypeError: aluno1.media is not a function
```


isec
Engenharia

> Funções > Métodos > this

```
const aluno1 = {
  nome: 'Nuno Afonso',
  numero: 2102124,
  anoNascimento: 2001,
  disciplinasConcluidas: ['AP', 'LS', 'TW', 'TAC'],
  notas: [12, 15, 13, 15],
  media: function () {
    let soma = 0;
    for (let i = 0; i < this.disciplinasConcluidas.length; i++)
      soma = soma + this.disciplinasConcluidas[i].nota;
    this.media = soma / this.disciplinasConcluidas.length;
    return this.media;
  }
};

aluno1.calcIdade(2021);
```

Diagram illustrating the use of `this` in a JavaScript object method. A red box highlights the `calcIdade` method call and its definition. A green arrow points from the `calcIdade` property access to the `calcIdade` function definition. A blue arrow points from the `this` keyword in the function body to the `aluno1` object, indicating that `this` refers to the object that called the method.



> Funções > Objetos > Métodos

```
const aluno1 = {
  nome: 'Nuno Afonso',
  numero: 2102124,
  disciplinas: ['AP', 'LS', 'TW', 'TAC'],
  notas: [12, 15, 13, 15],
  media: function() {
    let soma=0;
    for (let i=0; i<this.notas.length;i++)
      soma=soma+this.notas[i];
    this.media=soma/this.notas.length;
    return this.media;
  }
};
```

```
aluno1.media();
aluno2.media();

console.log(`O aluno "${aluno1.nome}" tem ${aluno1.media} de média em ${aluno1.notas.length} disciplinas concluidas e o "${aluno2.nome}" tem ${aluno2.media} de média em ${aluno2.notas.length} disciplinas concluidas!`);
```

top Filter All levels

O aluno "Nuno Afonso" tem 13.75 de média em 4 disciplinas concluidas e o "Ricardo Afonso" tem 13 de média em 3 disciplinas concluidas!

```
const aluno2 = {
  nome: 'Ricardo Afonso',
  numero: 2102125,
  disciplinas: ['SO', 'TW', 'TAC'],
  notas: [16, 11, 12],
  media: function() {
    let soma=0;
    for (let i=0; i<this.notas.length;i++)
      soma=soma+this.notas[i];
    this.media=soma/this.notas.length;
    return this.media;
  }
};
```



A ver mais tarde...

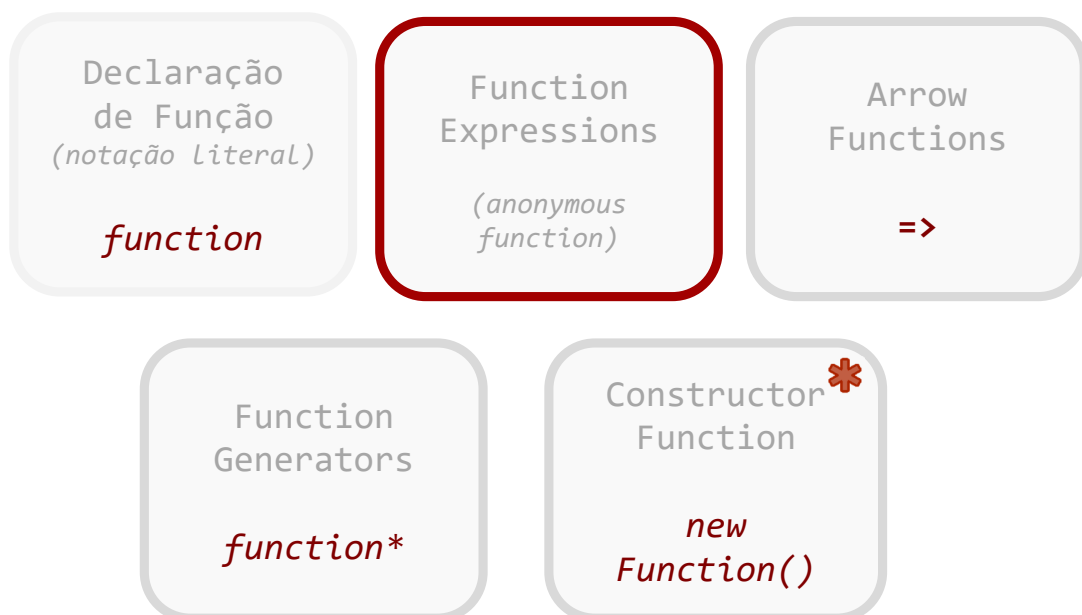
> Funções > Invocação

- Existem várias formas de invocar uma função:
 - Pelo nome da função...
 - `nomeFuncao(argumentos)`
 - Se for um método...
 - `objeto.nomeDoMetodo(argumentos)`
 - `objeto["nomeMetodo"](argumentos)`
 - Através do construtor
 - `new nomeFuncao(argumentos)`
 - Através do método *apply*
 - `nomeFuncao.apply(objeto,[argumentos])`



> Funções > Criação

- Existem diferentes formas de se criar e invocar uma função em JavaScript



> Funções > Function Expressions

- As funções em JavaScript também podem ser definidas por uma **expressão de função**
 - Pode ser uma **função anónima** (*Anonymous function*) ou **não** (*Named Function Expression*)
 - Expressão produz valores!
- Funções Anónimas são funções sem nome!
 - Na prática, as funções anónimas são normalmente usadas como **argumentos para outras funções** ou podem ser chamadas imediatamente atribuindo-as a uma variável que pode ser usada para invocar a função.
- Ao contrário de *function declarations* (funções de declaração), **function expressions não podem ser invocadas antes de sua definição no código**
 - Força o código a ficar mais organizado

> Funções > Function Expressions

```
function dobro(num) {  
    return num * 2;  
}  
console.log(dobro(5))
```



```
let dobro = function (num)  
{  
    return num * 2;  
}  
const dobro10 = dobro(10);  
console.log(dobro(5), dobro10);
```



Função tratada como uma expressão!

A função é chamada como função anónima!

Anonymous function!

JavaScript Expressions são muitas vezes escritas como
arrow functions a partir do ES2016

> Function Expressions > Hoisting

```
console.log(dobro(5));

let dobro = function (num)
{
    return num * 2;
}
```



✖ ▶ Uncaught ReferenceError: Cannot access 'dobro' before initialization

Function expressions **não** são **'hoisted'**

> Named Function Expression

- Quando uma função expression não é anónima mas tem nome
 - Possibilita que essa função possa ela própria seja referenciada internamente;
 - Não é visível fora da função

```
let funcaoOla = function (nome) {
    console.log(`Olá, ${nome}`);
}
funcaoOla();
funcaoOla('Maria');
```

Olá, undefined
Olá, Maria

```
let funcaoOla = function funcao(nome) {
    if (nome)
        console.log(`Olá, ${nome}`);
    else
        funcao("Desconhecido!");
}
funcaoOla();
funcaoOla('José');
funcao();
```

Olá, Desconhecido!
Olá, José

✖ ▶ Uncaught ReferenceError: funcao is not defined

> Named Function Expression

```
let funcaoOla = function funcao(nome) {  
  if (nome)  
    console.log(`Olá, ${nome}`);  
  else  
    funcao("Desconhecido!")  
}  
funcaoOla();  
funcaoOla('José');
```

Olá, Desconhecido!
Olá, José

```
let funcaoOla = function (nome) {  
  if (nome)  
    console.log(`Olá, ${nome}`);  
  else  
    funcaoOla("Desconhecido!")  
}  
funcaoOla();  
funcaoOla('José');
```

Olá, Desconhecido!
Olá, José



> Named Function Expression

```
let funcaoOla =function funcao(nome)  
{  
  if (nome)  
    console.log(`Olá, ${nome}`);  
  else  
    funcao("Desconhecido!")  
}
```

```
let bemVindo = funcaoOla;  
funcaoOla = 0;  
bemVindo();  
bemVindo('José');
```

Olá, Desconhecido!
Olá, José

```
let funcaoOla = function (nome)  
{  
  if (nome)  
    console.log(`Olá, ${nome}`);  
  else  
    funcaoOla("Desconhecido!")  
}
```

```
let bemVindo = funcaoOla;  
funcaoOla = 0;  
bemVindo();  
bemVindo('José');
```

✖ ▶ Uncaught TypeError:
funcaoOla is not a function

> Funções > Function Expressions

```
let idade = 18; // Obtém a idade em tempo de execução...
let funcao;
if (idade >= 18) {
  funcao = function () {
    console.log("Maior de Idade!");
  };
} else {
  funcao = function () {
    console.log("Menor de Idade!");
  };
}
funcao();
```

Mais flexibilidade em
relação ao *scope* e
tempo de execução



> Funções > Criação

- Existem diferentes formas de se criar e invocar uma função em JavaScript

Declaração
de Função
(*notação literal*)

function

Function
Expressions

(*anonymous
function*)

Arrow
Functions

=>

Function
Generators

*function**

Constructor
Function *

*new
Function()*



> Funções *Arrow* (\Rightarrow)

- Introduzidas no ES6, usada como uma expressão $() \Rightarrow \{ \}$
 - Sintaxe reduzida quando comparadas com *function expression*
- São sempre **anónimas** (*anonymous functions*)
- Tornam o código mais simples e reduzido \rightarrow **legível**.
- Se a função tem apenas uma linha de código, e esse retorna um valor, é possível remover $\{ \}$
 - Atenção que $\{ \}$ e $()$ e "return" são necessários em determinadas situações
 - $()$ Se tiver múltiplos ou sem argumentos
 - $\{ \}$ e return se várias linhas de código
- Uma função de seta não pode conter uma quebra de linha entre seus parâmetros e seta.

> Func. Expression > Arrow Func.

- Transformar **função anónima** em **função seta** (arrow function):
 - Remover a palavra "**function**" e colocar \Rightarrow entre o argumento e os $\{ \}$

```
function (num) {  
    return num * 2;  
}
```



```
(num) => {  
    return num * 2;  
}
```

- Remover o corpo de $\{ \}$ e a palavra return. O Return está implícito.

```
(num) => {  
    return num * 2;  
}
```



```
(num) => num * 2;
```

- Remover os parentheses nos argumentos

```
(num) => num * 2;
```



```
num => num * 2;
```

> Função Anônima > Arrow Function

JavaScript

```
let dobro = function (num) {  
  return num * 2;  
}
```

```
console.log(dobro(1));  
console.log(dobro(2));  
console.log(dobro(3));
```



```
let dobro = num => num * 2;
```

2

4

6

```
const array = [1, 2, 3];  
const dobro = array.map(num => num * 2);  
console.log(array);  
console.log(dobro);
```

```
▼ (3) [1, 2, 3] ⓘ  
  0: 1  
  1: 2  
  2: 3  
  length: 3  
  ► [[Prototype]]: Array(0)  
▼ (3) [2, 4, 6] ⓘ  
  0: 2  
  1: 4  
  2: 6  
  length: 3  
  ► [[Prototype]]: Array(0)
```

isec
Engenharia

> Arrow Functions > return

JavaScript

```
let f = function (n1, n2) {  
  return n1 + n2 + 1;  
}
```

```
console.log(f(1, 2))
```



```
f = (n1, n2) => n1+n2+1;
```

```
let n1 = 1;  
let n2 = 2;
```

```
function f() {  
  return n1 + n2 + 1;  
}
```

```
console.log(f())
```



```
f = () => n1 + n2 + 1;
```

return

fica implícito!



4

isec
Engenharia

> Arrow Functions > Um argumento

```
let inc = (n1) => {  
  return ++n1;  
}  
console.log(inc(2))
```



```
let inc = (n1) => ++n1;
```



```
let inc = n1 => ++n1;
```



3

```
let inc = n1 => n1++;
```

2

> Arrow Functions > Objectos

- E na criação de objectos ?

```
const pessoa = () => {  
  return {  
    nome: 'Manuel Afonso',  
    morada: 'Rua Carlos Seixas',  
    idade: '45'  
  }  
}  
console.log(pessoa());
```

**return
explícito**

```
const pessoa2 = () => ({  
  nome: 'Manuel Afonso',  
  morada: 'Rua Carlos Seixas',  
  idade: '45'  
});  
console.log(pessoa2());
```

**return
implícito**



▶ {nome: 'Manuel Afonso', morada: 'Rua Carlos Seixas', idade: '45'}

> Funções > Arrow Functions

```
let frutas = ['Banana', 'Pera', 'Maça', 'Laranja'];  
let dim1 = frutas.map(function (f) { return f.length; });  
let dim2 = frutas.map(f => f.length);  
console.log(dim1);  
console.log(dim2);
```

▶ (4) [6, 4, 4, 7]

▶ (4) [6, 4, 4, 7]

```
let f = (num1, num2) => {  
    let valor = 10;  
    return num1 + num2 + valor;  
}
```

- Vários parâmetros ()
- Várias linhas de código no corpo da função {}



> Funções > Arrow Function > **this**

JavaScript

- A palavra-chave **this** tem um comportamento diferente nas *arrow functions* quando comparador com declaração de funções ou funções de expressão;
- Ao aceder ao **this** através de uma *arrow function*, o seu valor é obtido de fora da função, isto é, do scope onde a função está definida, portanto o valor do **this** dentro de uma função seta é sempre herdado do escopo delimitador.
- O **this** dentro de uma *arrow function* vai ser sempre referência ao objeto ao qual ele já era, no momento em que a função foi criada.



> Funções > Arrow Function > **this**

```
function obtemThis() {  
    return this;  
}  
console.log(obtemThis());
```

```
Window {window: Window, self: Window,  
▶ document: document, name: '', location: Location, ...}
```

```
'use strict';  
function obtemThis() {  
    return this;  
}  
console.log(obtemThis());
```

undefined

```
const obtemThis = () => {  
    return this;  
}
```

```
Window {window: Window, self: Window,  
▶ document: document, name: '', location: Location, ...}
```

```
'use strict';  
const obtemThis = () => {  
    return this;  
}
```

```
Window {window: Window, self: Window,  
▶ document: document, name: '', location: Location, ...}
```

> Funções > Arrow Function > **this**

```
function funcao() {  
    console.log(this);  
}  
  
const obj = {  
    pro1: "qualquerNome",  
    pro2: 2  
};  
  
funcao();  
funcao(obj);  
funcao.call(obj);
```

```
const funcao1 = () => {  
    console.log(this);  
}
```

```
VM2865:1  
Window {0: Window, window: Window, self: Window, document: document,  
▶ name: '', location: Location, ...}
```

this
de forma explícita

```
VM253:2  
Window {0: Window, window: Window, self: Window, document: document,  
▶ name: '', location: Location, ...}
```

```
VM253:2  
Window {0: Window, window: Window, self: Window, document: document,  
▶ name: '', location: Location, ...}  
▶ {pro1: 'qualquerNome', pro2: 2}
```

> Funções > Arrows > Métodos

```
const retangulo = {  
  lado: 10,  
  altura: 5,  
  desenha: () => console.log("Desenha Retangulo!"),  
  calculaArea: () => this.lado * this.altura,  
  calculaPerimetro: () => this.lado*2 + this.altura*2  
}
```

```
retangulo.desenha();  
console.log(retangulo.calculaArea());  
console.log(retangulo.calculaPerimetro());
```

Desenha Retangulo!

NaN

NaN

NaN

> Funções > Arrows > Métodos

```
var lado = 5;  
let altura = 5;  
const retangulo = {  
  lado: 10,  
  altura: 5,  
  getThis: () => console.log(this),  
  getThisLado: () => console.log(this.lado),  
  desenha: () => console.log("Desenha Retangulo!"),  
  calculaArea: () => this.lado * this.altura,  
  calculaAreaV2: () => lado * altura,  
  calculaPerimetro: () => this.lado * 2 + this.altura * 2  
}  
console.log(this.lado); console.log(this.altura);  
retangulo.desenha();  
retangulo.getThis();  
retangulo.getThisLado();  
console.log(retangulo.calculaArea());  
console.log(retangulo.calculaAreaV2());  
console.log(retangulo.calculaPerimetro());
```

5
undefined
Desenha Retangulo!

Window {window: window, location: Location, ...}
5
NaN
25
NaN

> Funções > Arrows > Métodos

```
const pessoa1 = {  
  nome: "Nuno",  
  saudacao: function () {  
    console.log(`Menu nome é ${this.nome}`);  
  }  
}
```

```
const pessoa2 = {  
  nome: "Ricardo"  
}
```

```
pessoa1.saudacao();  
pessoa1.saudacao.call(pessoa2);
```

Menu nome é Nuno

Menu nome é Ricardo

```
const fPessoa2 = pessoa1.saudacao.bind(pessoa2);  
funcaoPessoa2();
```

Menu nome é Ricardo

this pré-configurado



> Funções > Criação

- Existem diferentes formas de se criar e invocar uma função em JavaScript

Declaração
de Função
(*notação literal*)

function

Function
Expressions

(*anonymous
function*)

Arrow
Functions

=>

Function
Generators

*function**

Constructor
Function

*new
Function()*



> Funções > Generator function

- A declaração `function*` define uma função geradora que retorna um objecto *Generator*
- Geradores são funções cuja **execução pode ser interrompida e posteriormente reconduzida**.
- Quando uma função geradora é invocada, ela não executa o seu código imediatamente, o que faz é retornar um objeto especial designado como “Objeto gerador” para gerir a execução.
 - Quando o método `next()` do objeto iterator é chamado, o conteúdo da função do gerador é executado até a primeira expressão *yield*, que especifica o valor a ser devolvido do iterator

> Funções > *Generator function*

```
function* gerador(i) {  
  yield i;  
  yield i + 10;  
}
```

```
▼ {value: 10, done: false} ⓘ  
  done: false  
  value: 10  
  ► [[Prototype]]: Object
```

```
const gen = gerador(10);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);
```

10

20

undefined

> Funções > Generator function

```
function* gerador(inicio) {  
    let indice = inicio;  
    while (true)  
        yield indice++;  
}  
  
const gen = gerador(20);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);  
console.log(gen.next().value);
```

20
21
22
23
24
25

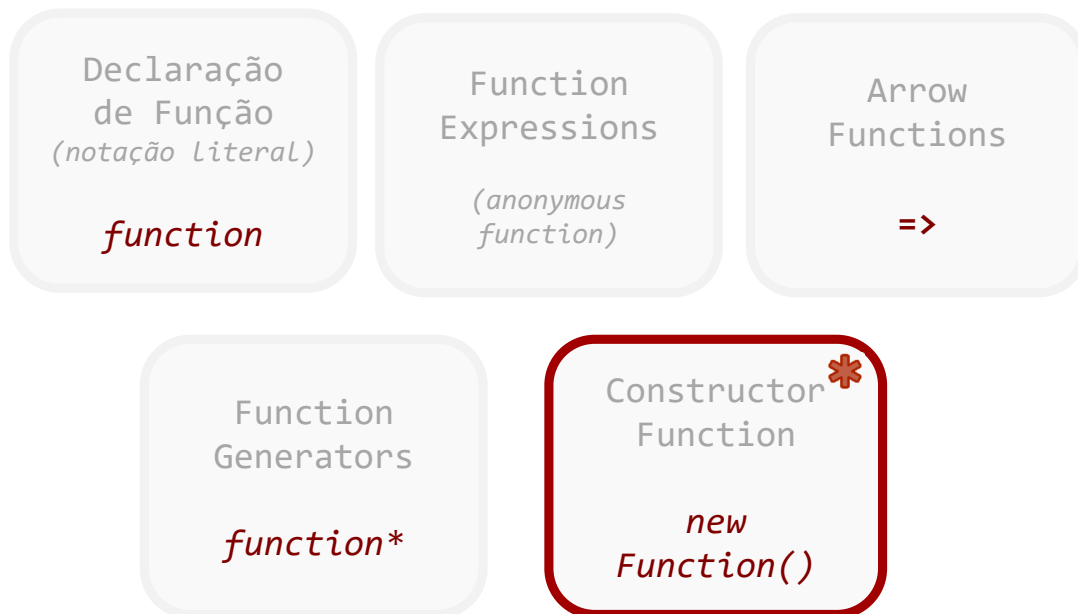
> Funções > Generator function

```
function* logGenerator() {  
    console.log(0);  
    console.log('Pára aqui');  
    console.log(1, yield);  
    console.log(2, yield);  
    console.log(3, yield);  
}  
  
var gen = logGenerator();  
gen.next();  
gen.next('Linguagens');  
gen.next('Script');  
gen.next('JavaScript');  
gen.next('E agora?');
```

0
Pára aqui
1 'Linguagens'
2 'Script'
3 'JavaScript'

> Funções > Criação

- Existem diferentes formas de se criar e invocar uma função em JavaScript



> Function() Constructor

- Permite criar um novo objecto do tipo Function, no qual a função é criada literalmente a partir de uma string que é passada em tempo de execução

- Sintaxe:

```
let func = new Function([arg1, arg2, ...argN], functionBody);
```

- Menos eficientes comparadas com a declaração de função e expressões de função.

```
const dobro = new Function('a', 'return a*a');  
console.log(dobro(2));
```

```
const soma = new Function('a', 'b', 'return a + b');  
console.log(soma(2, 6)); // 8
```

> Constructor functions

- Um constructor é uma função especial que cria e inicializa uma instância de um objeto, recorrendo à palavra chave **new**.



**Conceito a ser abordado em
profundidade na secção
*"Orientação a Objetos"***



</Funções>

