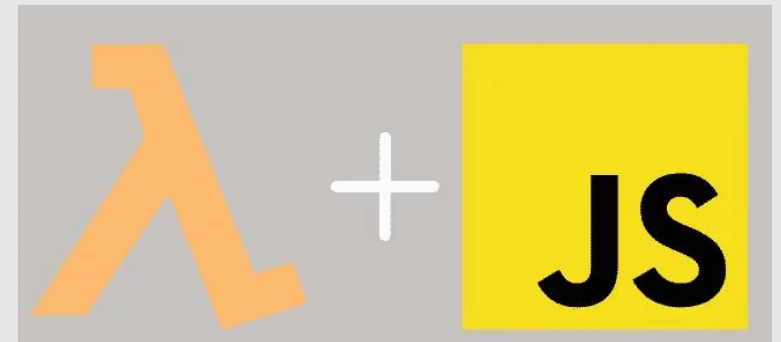
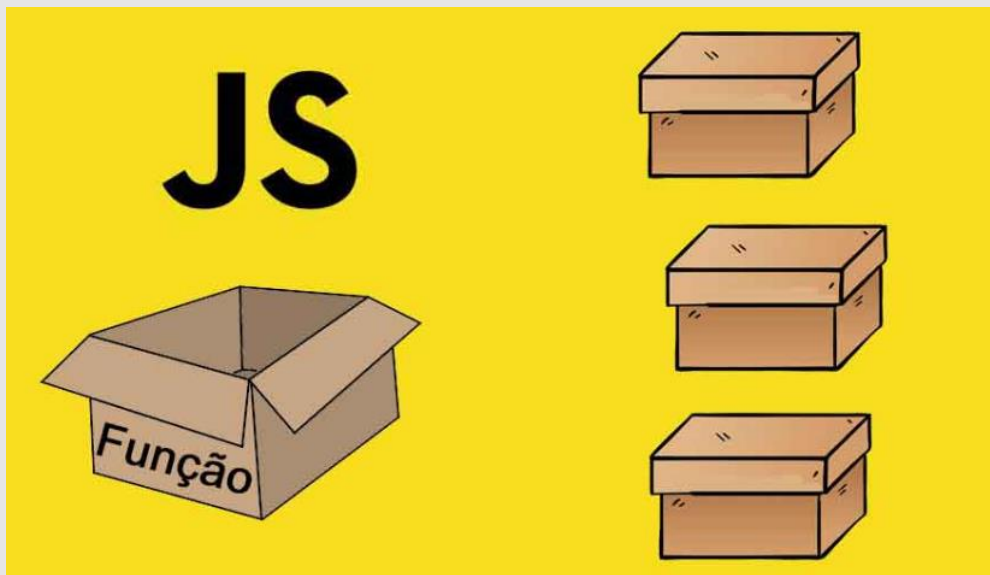


Desenvolvimento Web

Unidade 2 – Parte 5

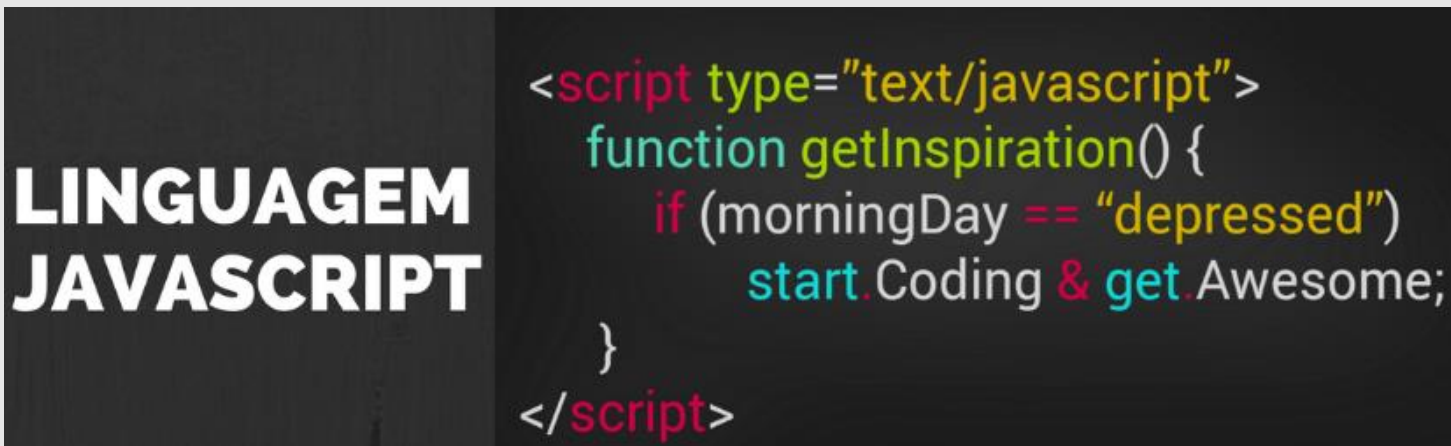
Programação Funcional em JavaScript



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@online.uscs.edu.br
aparecidovfreitas@gmail.com

Funções

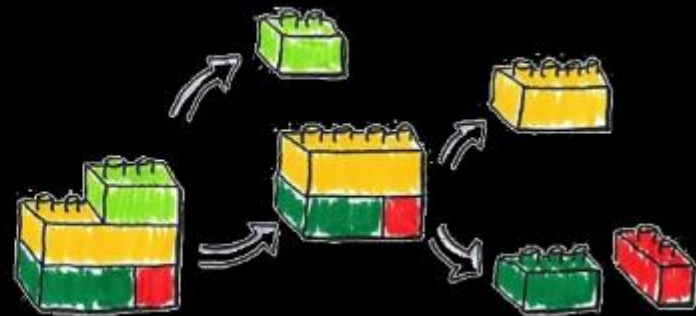
- Uma **função** é um trecho de código de programa independente de qualquer parte do programa, mas relacionado ao programa com atribuições bem definidas;
- **Funções** representam um conjunto de instruções que efetuam uma tarefa específica. **Funções** também podem ser chamados de métodos ou sub-rotinas;
- De forma geral, **funções** podem receber valores de entrada (argumentos) e gerar valores de saída.



Funções

- ❖ Pode-se assim, dividir-se o código em módulos (**divisão e conquista**), e cada módulo desempenha uma ação particular;
- ❖ Isso facilita a manutenção e possibilita reusabilidade do código.

Divisão e
Conquista



Definição de Funções

- ❖ Uma função é um bloco de código que é usado para realizar uma tarefa específica. As funções são úteis para tornar o código mais organizado e reutilizável;
- ❖ Para criar uma função em **Javascript**, use a palavra-chave **function** seguida pelo nome da função, parênteses e um bloco de código entre chaves.

Esta é a forma padrão de definir uma função:

```
javascript
```

```
function nomeDaFuncao(parametro1, parametro2) {  
    // corpo da função  
}
```

Funções

Exemplo:

javascript

```
function saudacao(nome) {  
    console.log("Olá, " + nome + "!");  
}
```

Funções

Exemplo:

javascript

```
const quadrado = function(x) {  
    return x * x;  
};
```

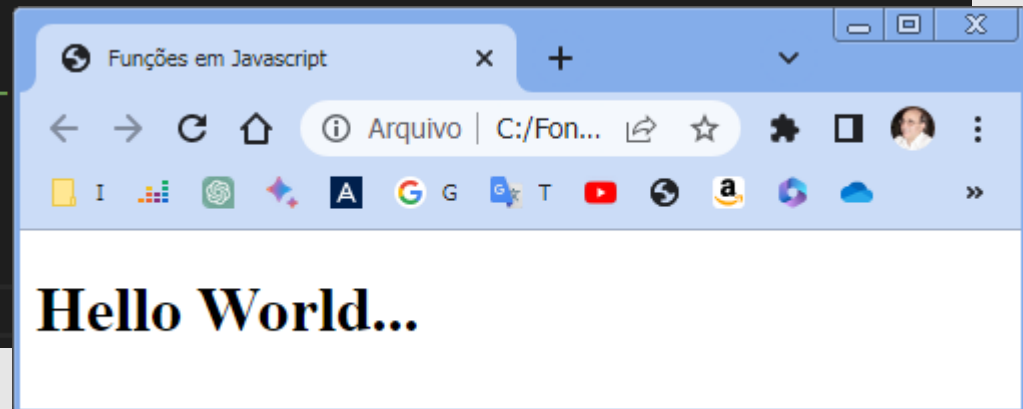
Definição de Funções

```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!--
4
5  <head>
6      <meta charset="UTF-8">
7      <meta http-equiv="X-UA-Compatible" content="IE=edge">
8      <meta name="viewport" content="width=device-width, initial-scale=1.0">
9      <script>
10         function helloWorld() {
11             document.write("<h1>" + "Hello World..." + "</h1>");
12         }
13         helloWorld();
14     </script>
15     <title>Funções em Javascript</title>
16 </head>
17 <!--
18
19 <body></body>
20
21 </html>
  
```

Definição de Funções

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!--
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1.0">
9    <script>
10     function helloWorld() {
11       document.write("<h1>" + "Hello World..." + "</h1>");
12     }
13     helloWorld();
14   </script>
15   <title>Funções em Javascript</title>
16 </head>
17 <!--
18
19 <body></body>
20
21 </html>
```



Parâmetros e Argumentos

- **Parâmetros:** São os nomes listados na definição da função.
- **Argumentos:** São os valores reais passados para a função quando ela é invocada.

Exemplo com parâmetros padrão:

Com ES6, podemos definir valores padrão para os parâmetros:

javascript

```
function saudacao(nome = "visitante") {  
    console.log("Olá, " + nome + "!");  
}  
saudacao(); // Saída: "Olá, visitante!"
```

Retornando Valores

O valor retornado por uma função pode ser usado ou armazenado em uma variável:

javascript

```
function somar(a, b) {  
    return a + b;  
}  
  
const resultado = somar(3, 4); // resultado é 7
```

Funções sem Retorno

- ❖ Quando uma **função** é chamada o fluxo de execução do programa, é desviado para a função em questão e, após executá-la, o fluxo do programa retorna ao ponto de chamada para dar continuidade à execução do programa.

```

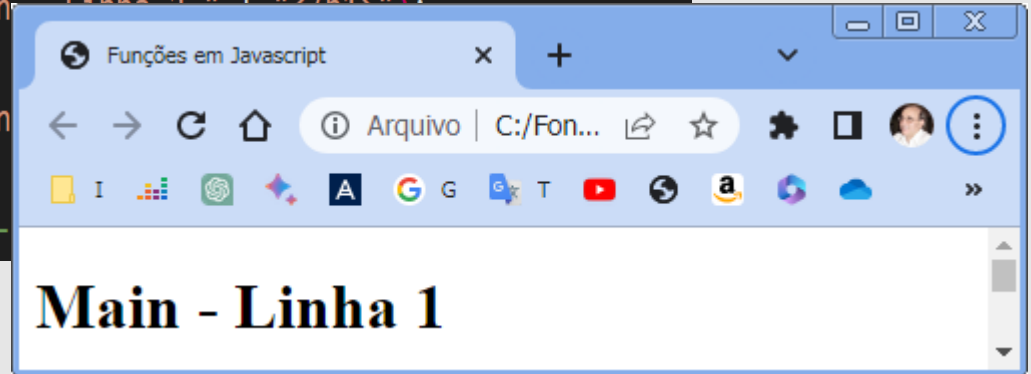
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!------->
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1.0">
9    <script>
10     //-----
11     function Main() {
12       document.write("<h1>" + "Main - Linha 1 " + "</h1>");
13       document.write("<h1>" + "Main - Linha 2 " + "</h1>");
14       Exemplo();
15       document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16     }
17     //-----
  
```

Funções sem Retorno

```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!------->
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1.0">
9    <script>
10     //-----
11     function Main() {
12       document.write("<h1>" + "Main - Linha 1 " + "</h1>");
13       document.write("<h1>" + "Main - Linha 2 " + "</h1>");
14       Exemplo();
15       document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16     }
17     //-----

```

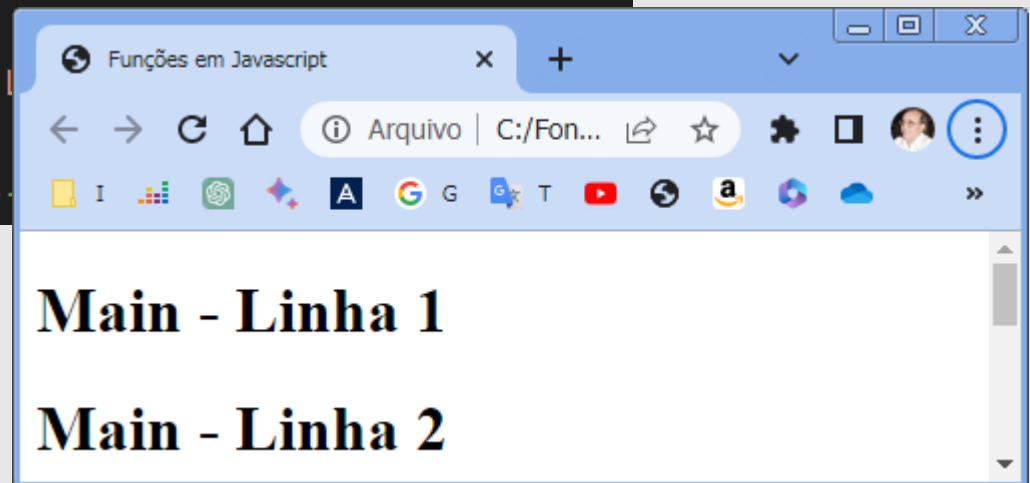


Funções sem Retorno

```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!------->
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1.0">
9    <script>
10     //-----
11     function Main() {
12       document.write("<h1>" + "Main - Linha 1 " + "</h1>");
13       document.write("<h1>" + "Main - Linha 2 " + "</h1>");
14       Exemplo();
15       document.write("<h1>" + "Main - 
16     }
17     //-----

```



Funções sem Retorno

```

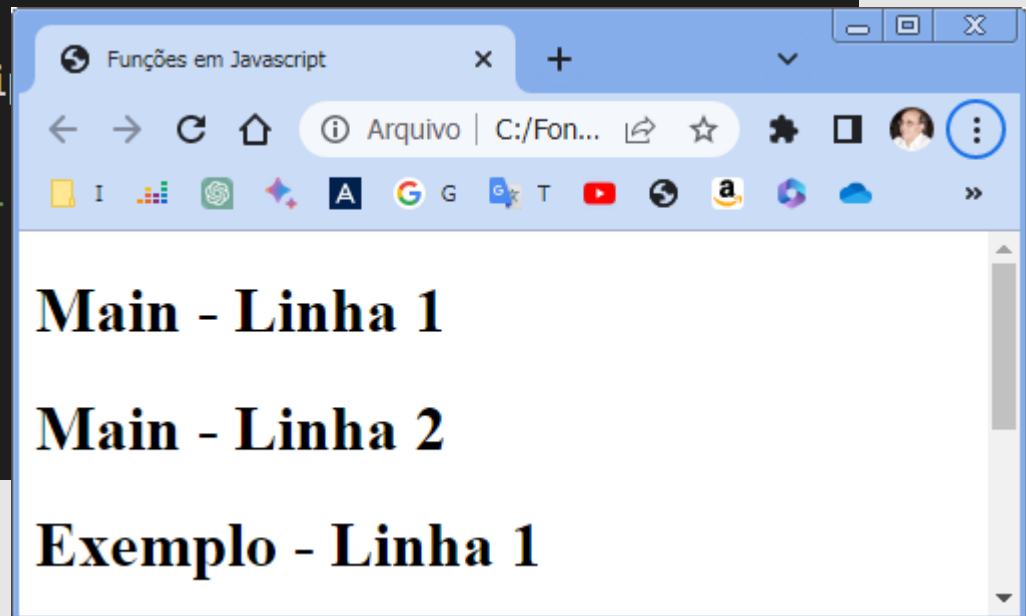
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!------->
4
5  <head>
6      <meta charset="UTF-8">
7      <meta http-equiv="X-UA-Compatible" content="IE=edge">
8      <meta name="viewport" content="width=device-width, initial-scale=1.0">
9      <script>
10         //-----
11         function Main() {
12             document.write("<h1>" + "Main - Linha 1 " + "</h1>");
13             document.write("<h1>" + "Main - Linha 2 " + "</h1>");
14             Exemplo();
15             document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16         }
17         //-----
  
```

Funções sem Retorno

```

17  //-----
18  function Exemplo() {
19  → document.write("<h1>" + "Exemplo - Linha 1 " + "</h1>");
20    document.write("<h1>" + "Exemplo - Linha 2 " + "</h1>");
21    document.write("<h1>" + "Exemplo - Linha 3 " + "</h1>");
22  }
23  //-----
24  Main();
25  </script>
26  <title>Funções em Javascript
27  </head>
28  <!-------
29
30  <body></body>
31
32  </html>

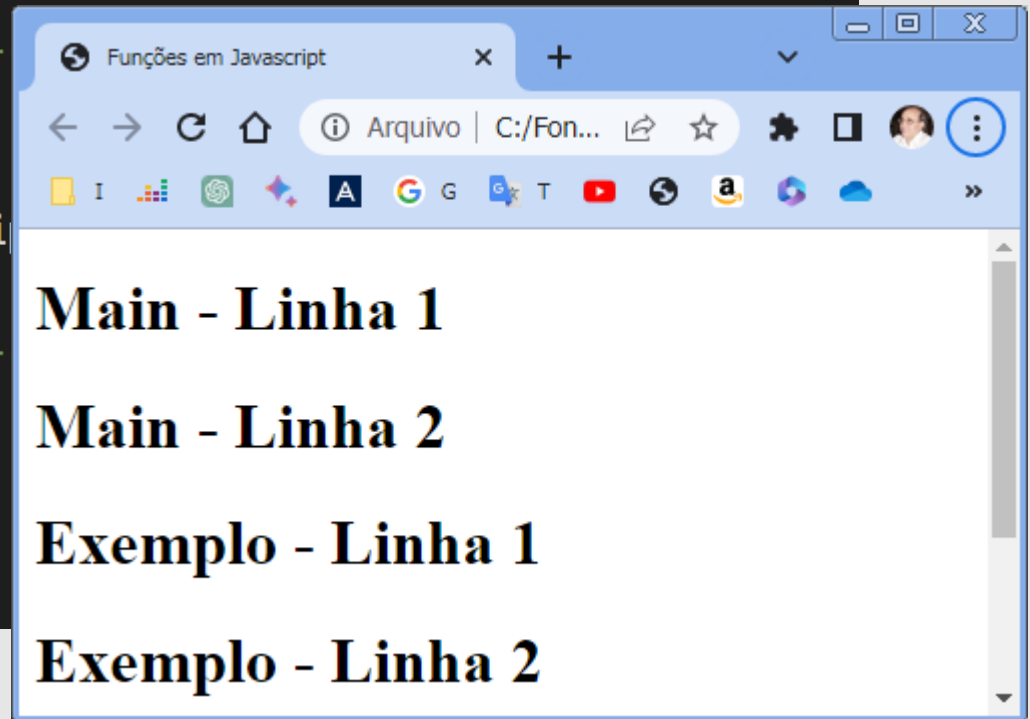
```



Funções sem Retorno

```

17  //-----
18  function Exemplo() {
19      document.write("<h1>" + "Exemplo - Linha 1 " + "</h1>");
20      document.write("<h1>" + "Exemplo - Linha 2 " + "</h1>");
21      document.write("<h1>" + "Exemplo - Linha 3 " + "</h1>");
22  }
23  //-----
24  Main();
25  </script>
26  <title>Funções em Javascript
27  </head>
28  <!-------
29
30  <body></body>
31
32  </html>
  
```

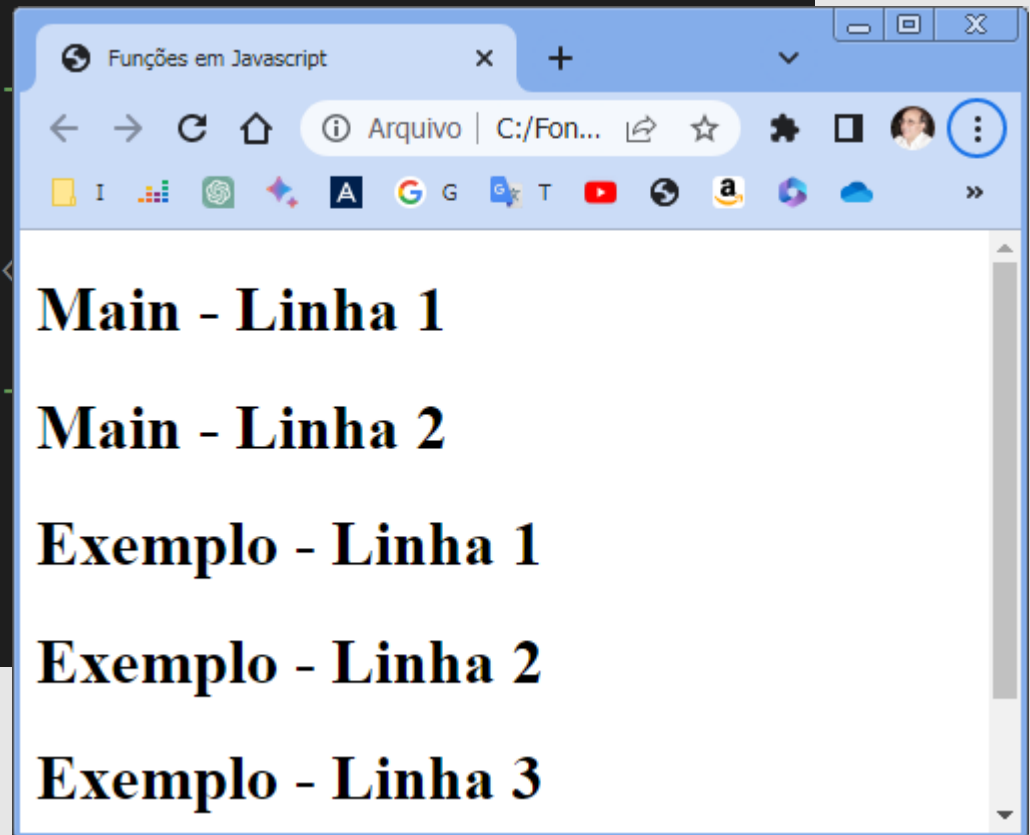


Funções sem Retorno

```

17  //-----
18  function Exemplo() {
19      document.write("<h1>" + "Exemplo - Linha 1 " + "</h1>");
20      document.write("<h1>" + "Exemplo - Linha 2 " + "</h1>");
21      document.write("<h1>" + "Exemplo - Linha 3 " + "</h1>");
22  }
23  //-----
24  Main();
25  </script>
26  <title>Funções em Javascript<
27  </head>
28  <!-------
29
30  <body></body>
31
32  </html>

```



Funções sem Retorno

```

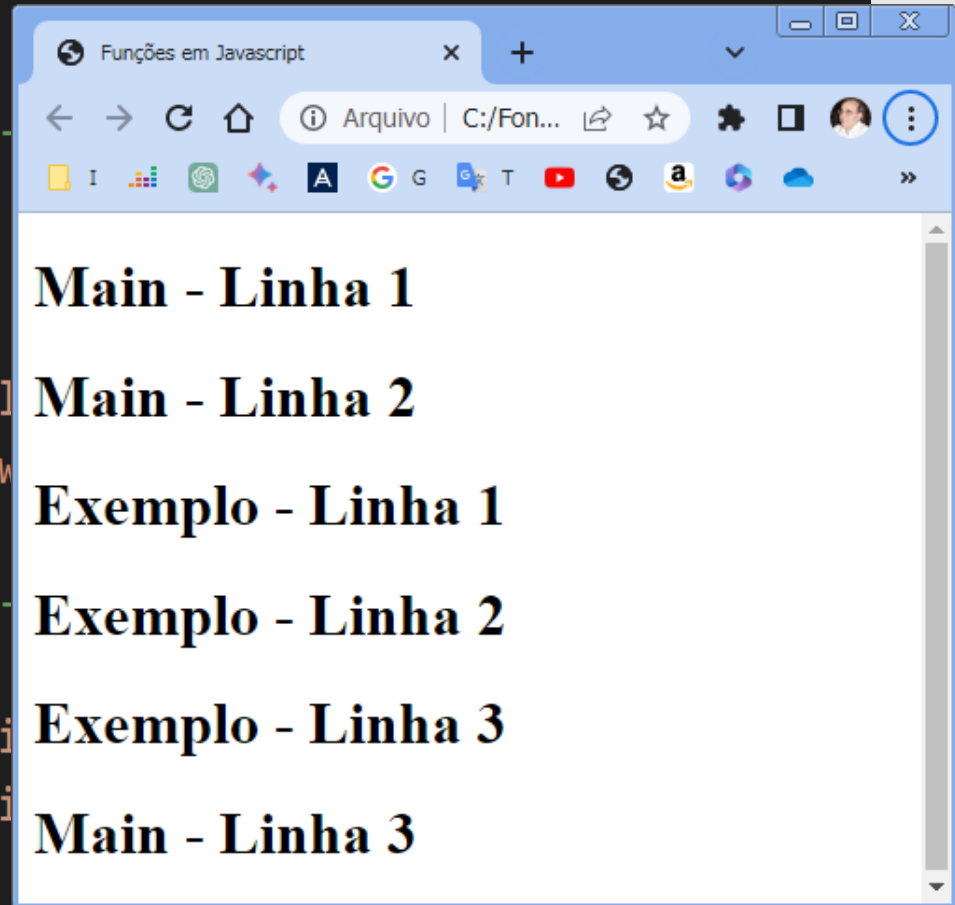
17  //-----
18  function Exemplo() {
19      document.write("<h1>" + "Exemplo - Linha 1 " + "</h1>");
20      document.write("<h1>" + "Exemplo - Linha 2 " + "</h1>");
21      document.write("<h1>" + "Exemplo - Linha 3 " + "</h1>");
22  }
23  //-----
24  Main();
25  </script>
26  <title>Funções em Javascript</title>
27  </head>
28  <!------->
29
30  <body></body>
31
32  </html>
  
```

Funções sem Retorno

```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!--
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1">
9    <script>
10      //-----
11      function Main() {
12        document.write("<h1>" + "Main - Linha 1");
13        document.write("<h1>" + "Main - Linha 2");
14        Exemplo();
15        document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16      }
17      //-----

```

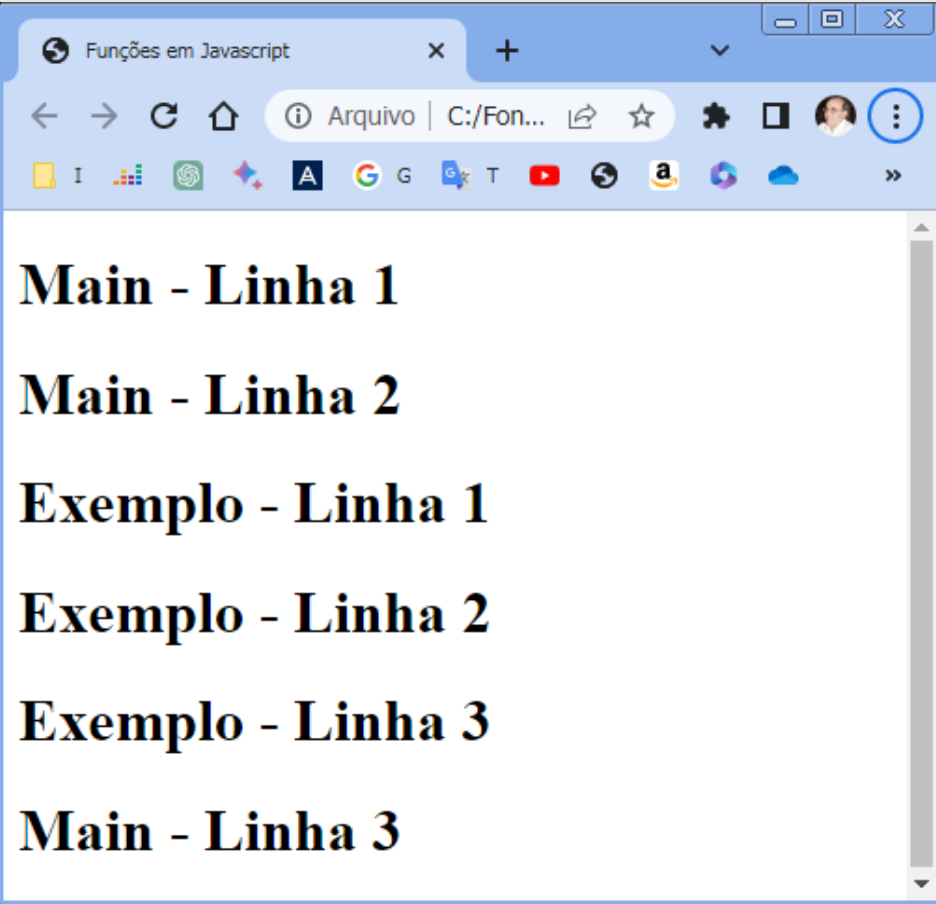


Funções sem Retorno

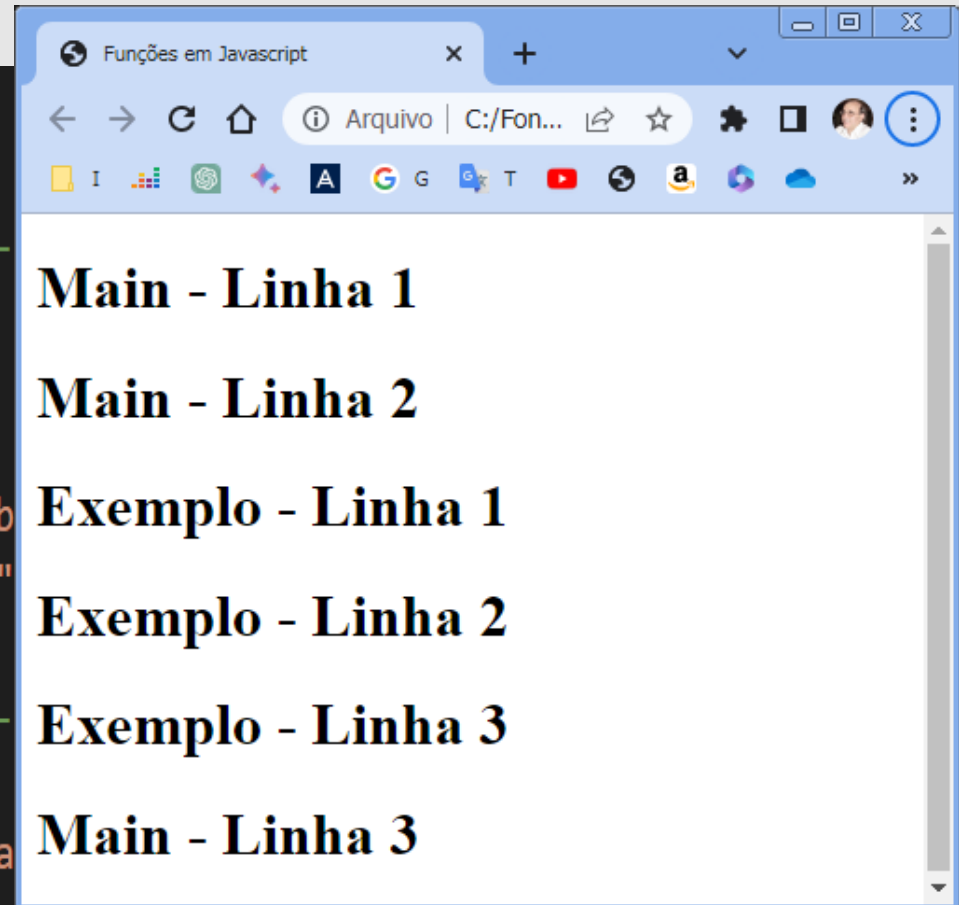
```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!--
4
5  <head>
6    <meta charset="UTF-8">
7    <meta http-equiv="X-UA-Compatible" content="IE=edge">
8    <meta name="viewport" content="width=device-width, initial-scale=1">
9    <script>
10     //-----
11     function Main() {
12       document.write("<h1>" + "Main - Linha 1 " + "</h1>");
13       document.write("<h1>" + "Main - Linha 2 " + "</h1>");
14       Exemplo();
15       document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16     }
17     //-----

```



Funções sem Retorno



```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <!--
4
5  <head>
6      <meta charset="UTF-8">
7      <meta http-equiv="X-UA-Compatib
8      <meta name="viewport" content="
9  <script>
10     //-----
11     function Main() {
12         document.write("<h1>" + "Ma
13         document.write("<h1>" + "Main - Linna 2 " + "</n1>");
14         Exemplo();
15         document.write("<h1>" + "Main - Linha 3 " + "</h1>");
16     }
17     //-----

```

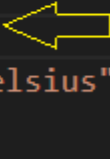
Comunicação entre funções

- ❖ Funções receber dados (parâmetros) que são passados à função quando ela for chamada;
- ❖ Os parâmetros podem ser usados para fornecer entrada para a função ou para retornar valores da função.
- ❖ Existem basicamente dois métodos empregados para passagem e recebimento de dados entre funções:

✓ **Passagem de parâmetros por Valor**

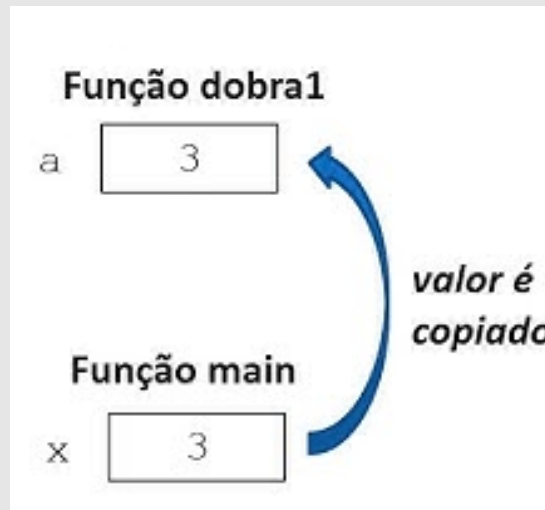
✓ **Passagem de parâmetros por Referência**

```
JS funcao1.js x
1  function ConverteFahrenheitCelsius(fahrenheit) {
2      return (5/9) * (fahrenheit-32);
3  }
4
5  var x = ConverteFahrenheitCelsius(88);
6  var texto = "A temperatura é " + x + " Celsius";
7  console.log(texto);
8
```



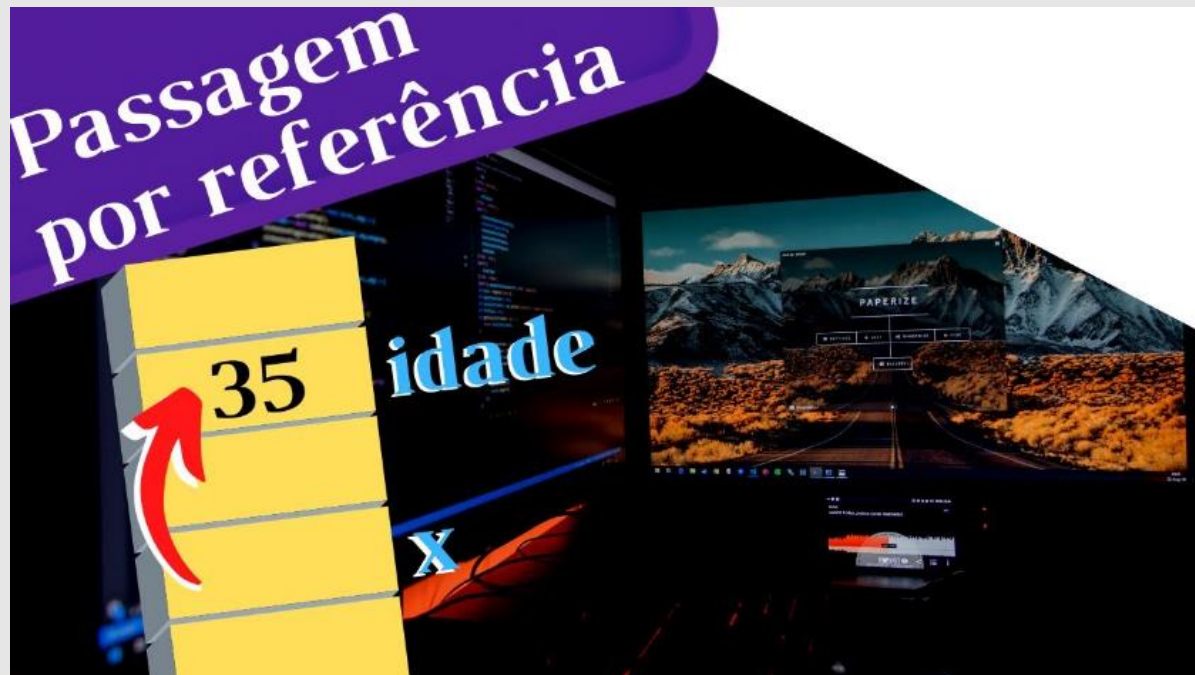
Passagem de parâmetros por Valor

- ❖ Corresponde ao método mais comum de passar parâmetros entre funções na linguagem **JavaScript**;
- ❖ Quando um parâmetro é passado por **valor**, seu valor é copiado para a função chamada. Se o valor do parâmetro for alterado na função chamada, a alteração **não** será refletida na função que a chamou.



Passagem de parâmetro por Referência

- ❖ Quando um parâmetro é passado por **referência**, a referência ao parâmetro é passada para a função chamada;
- ❖ Isso significa que a função chamada pode acessar o valor original do parâmetro e **alterá-lo**.



Passagem de parâmetro por Referência

Em **JavaScript**, todos os parâmetros são passados para as funções por **valor**. No entanto, quando você passa um **objeto para uma função**, na verdade está passando uma **referência** ao **objeto**, e não o próprio objeto. Portanto, se você alterar as propriedades do objeto dentro da função, essas mudanças serão refletidas fora da função.



Passagem de parâmetro por Referência

```
function updateName(obj) {  
    obj.name = "Novo Nome";  
}  
  
let person = {  
    name: "Nome Original"  
};  
  
console.log(person.name); // Saída: Nome Original  
updateName(person);  
console.log(person.name); // Saída: Novo Nome
```

Tentando modificar um parâmetro primitivo

javascript

```
function updateNumber(num) {  
    num = 100;  
}  
  
let myNumber = 5;  
console.log(myNumber); // Saída: 5  
updateNumber(myNumber);  
console.log(myNumber); // Saída: 5
```

Neste exemplo, `myNumber` é um número (um tipo de dado primitivo). Quando passamos `myNumber` para `updateNumber`, o JavaScript passa o valor de `myNumber`, não uma referência a ele. Portanto, qualquer modificação em `num` dentro da função não afeta `myNumber`.

Alterando elementos de um Array

javascript

```
function updateArray(arr) {  
    arr[0] = "Alterado!";  
}  
  
let myArray = ["Original", "Valor"];  
console.log(myArray[0]); // Saída: Original  
updateArray(myArray);  
console.log(myArray[0]); // Saída: Alterado!
```

Arrays em JavaScript são objetos e, portanto, são passados por referência. No exemplo acima, quando passamos `myArray` para `updateArray`, modificamos diretamente o array original.

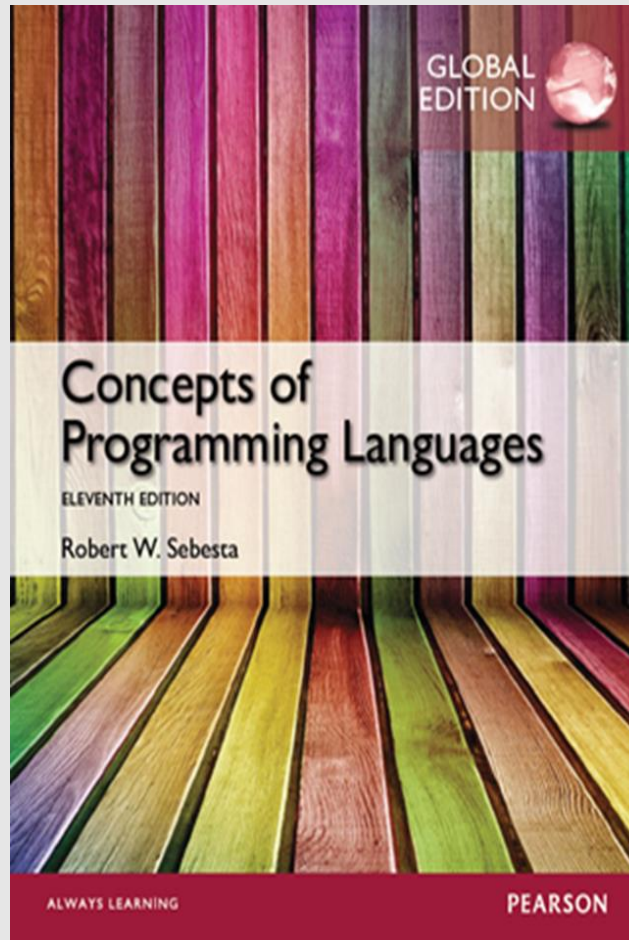
Paradigma Funcional de Programação



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@maua.br
aparecidovfreitas@gmail.com

Bibliografia

■ R. Sebesta – Concepts Of Programming Languages

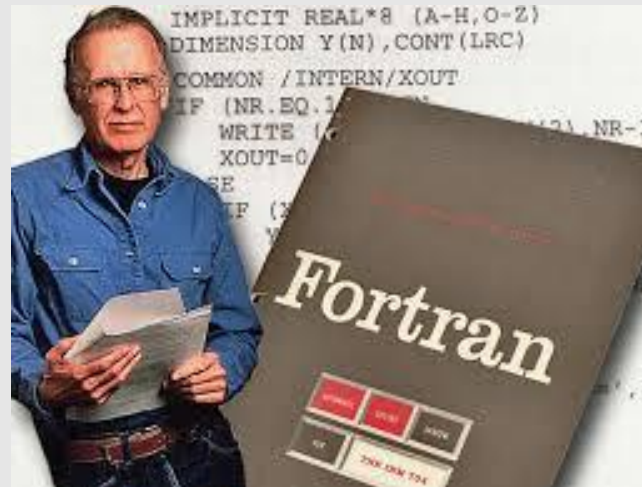


O Paradigma Funcional

- ✓ Fortran foi a primeira linguagem de programação;
- ✓ O primeiro compilador Fortran foi desenvolvido por uma equipe da IBM sendo chefiada por **John Backus**, na década de 50. Linguagem fortemente **aderente** ao paradigma **imperativo** e seu projeto tinha como grande objetivo o uso eficiente de máquina. A partir da Linguagem Fortran, diversas outras linguagens foram desenvolvidas;
- ✓ Mas, em 1977 na palestra ministrada por John Backus quando ganhou o prêmio ACM Turing, ele argumentou que **linguagens funcionais** são melhores que as **imperativas**, pois podem apresentar mais **confiabilidade**, **legibilidade** e com maior probabilidade de estarem **corretas**.



Em que se baseou Backus para afirmar que Linguagens Funcionais apresentam maior legibilidade e confiabilidade ?

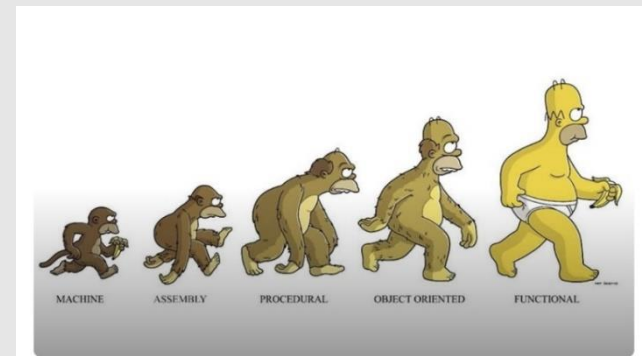


Linguagens Funcionais

- ✓ O argumento de John Backus teve como base que em **Linguagens Puramente Funcionais**, o significado das expressões são independentes de contexto;
- ✓ Em **Linguagens Puramente Funcionais** nem expressões nem funções apresentam Efeitos Colaterais (**Side Effects**);
- ✓ **Backus** propôs na época uma nova Linguagem Funcional chamada **FP** (Functional Programming) para embasar seu argumento;
- ✓ A linguagem não vingou, mas abriu espaço para pesquisa do Paradigma Funcional de Programação;

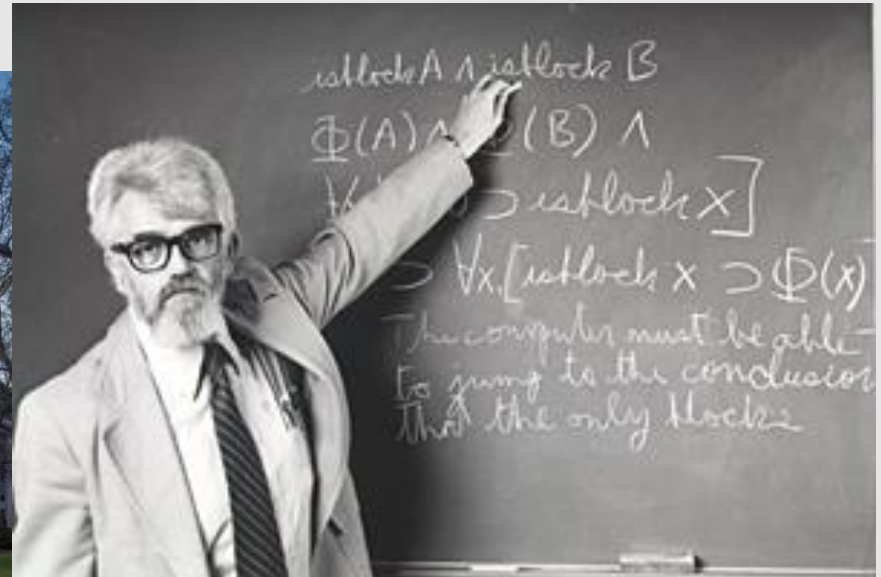
Programas Funcionais x Imperativos

- ✓ Uma das principais características dos programas escritos em Linguagens Imperativas é que eles possuem ESTADO;
- ✓ Ou seja, a execução do programa corresponde à sucessivas mudanças de estado em variáveis, no qual a resposta do programa será representada pelos estados finais de suas **variáveis** (Transformação de Estado);
- ✓ Para grandes programas, esta tarefa pode ser difícil;
- ✓ Em programas escritos com **Linguagens Puramente Funcionais** estes problemas não ocorrem, pois programas funcionais **NÃO** possuem Estado nem tão pouco Variáveis;



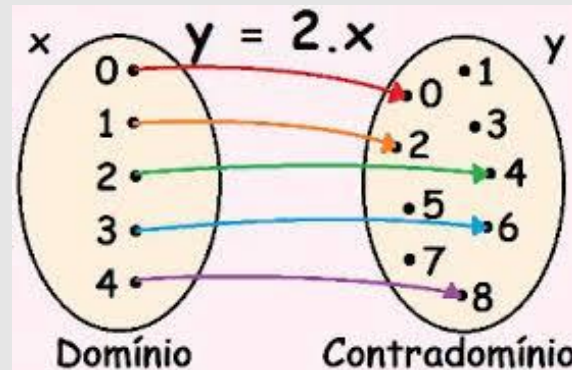
Linguagem de Programação Lisp

- ✓ Concebida por John McCarthy em 1959, no MIT;
- ✓ Focada no uso exclusivo de funções matemáticas como estrutura de dados;
- ✓ Tem como base formal o Cálculo Lambda de Alonzo Church;
- ✓ É ainda a mais importante linguagem representativa do Paradigma Funcional.



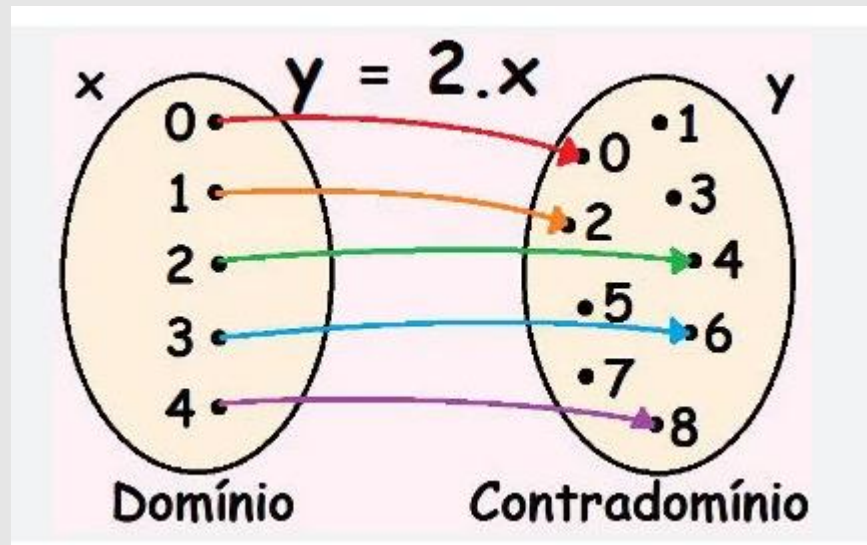
Funções Matemáticas

- ✓ Uma **função** matemática é um **mapeamento** de elementos de um conjunto, chamado conjunto **Domínio**, para outro conjunto, chamado **Contra-Domínio** (range set);
- ✓ A definição de uma função especifica o domínio e o range set, de forma explícita ou implícita.
- ✓ O mapeamento é descrito por uma expressão. Funções são geralmente aplicadas a um elemento específico do **Domínio**, passado como parâmetro para a função;
- ✓ Ao se aplicar um argumento à função obtém-se um valor do **Contra-Domínio**.



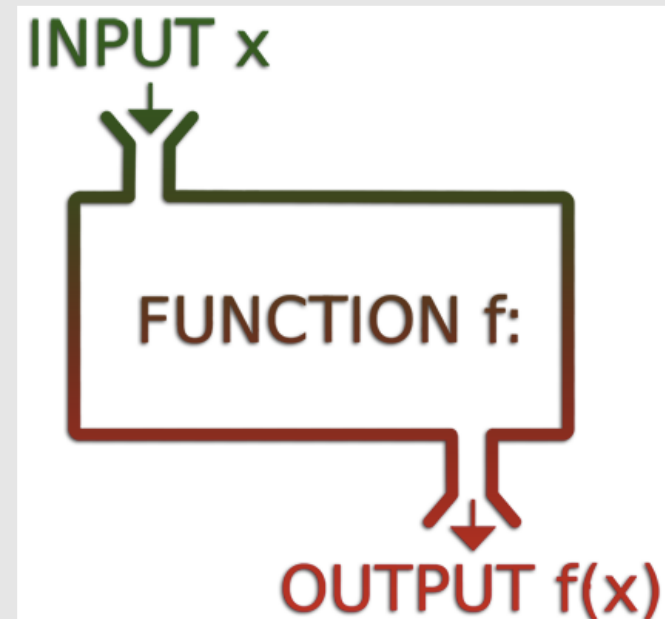
Funções Matemáticas

- ✓ A ordem de avaliação da expressão de mapeamento é controlada por Recursão e expressões condicionais. Diferentemente do Paradigma Imperativo no qual a avaliação é feita por sequenciamento e repetição iterativa;
- ✓ Outra importante característica das funções matemáticas é que elas sempre mapeiam o **mesmo** valor do Contra-Domínio para um valor do Domínio.



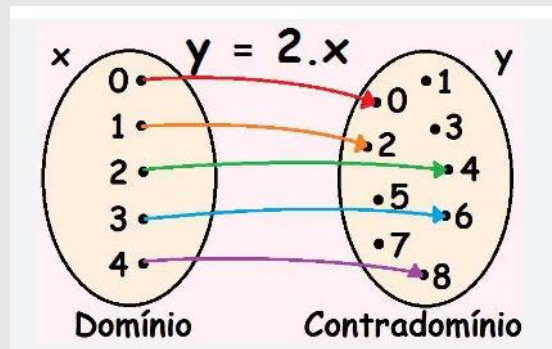
Funções Matemáticas

- ✓ Essa característica ocorre pois com funções matemáticas **não** há dependência de **valores externos** e **não** há **side effects**;
- ✓ Essa particularidade das funções matemáticas **não** ocorre nas linguagens imperativas, pois nestas linguagens um **subprograma** pode depender dos valores **correntes** de diversas **variáveis não-locais** ou **globais**, causando assim **side effects**.



Funções Matemáticas

- ✓ Na Matemática, não existe algo como uma variável que é usada para modelar uma localização de memória;
- ✓ Variáveis locais nas linguagens imperativas mantém o **estado** da função. Nessas linguagens, a computação é realizada pela avaliação das expressões que **modificam** o **estado** do programa, por meio de **comandos** de atribuição. Na Matemática, por outro lado, não existe o conceito de **estado de uma função**;
- ✓ Uma função matemática **sempre** mapeia seu parâmetro (ou parâmetros) para um valor (ou valores) ao invés de se especificar uma sequência de operações em valores de memória para computar um resultado.



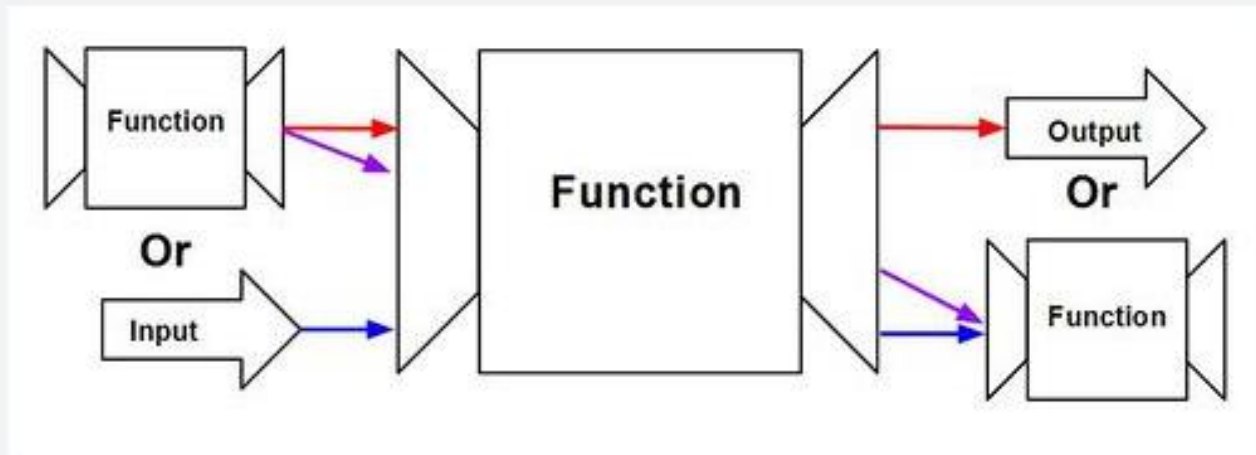
Cálculo Lambda

- ✓ Church (1941) definiu um modelo formal de computação (um sistema formal para definição de funções, aplicação de funções e recursão) com o emprego de expressões lambda;
- ✓ O **Cálculo Lambda** pode ser tipado ou não tipado;
- ✓ As linguagens puramente funcionais baseiam-se no **Cálculo Lambda** não tipado;



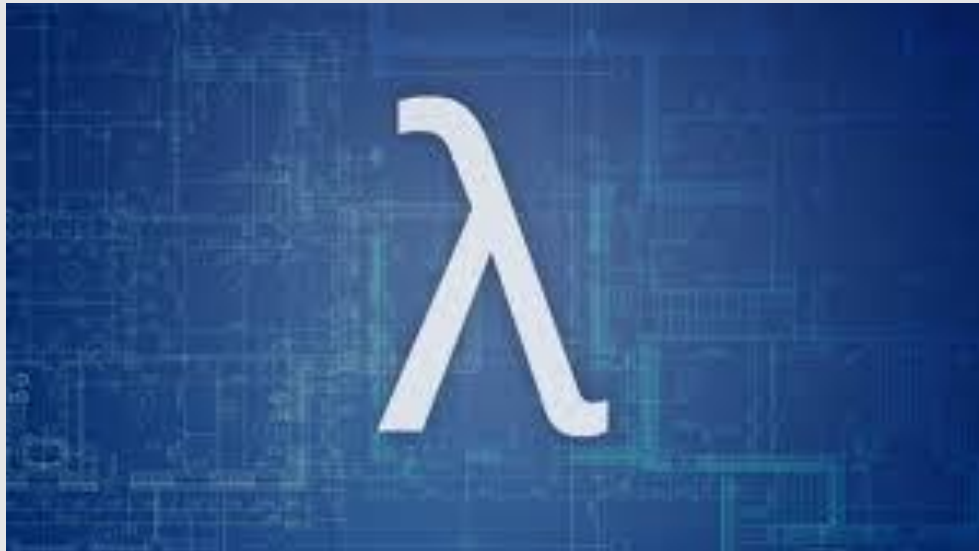
Higher Order Function

- ✓ Uma **higher-order-function**, ou forma funcional (functional form), é uma função **que pode receber como parâmetro uma ou mais funções**;
- ✓ **Higher-order-function** podem também **produzir funções** como resultado;



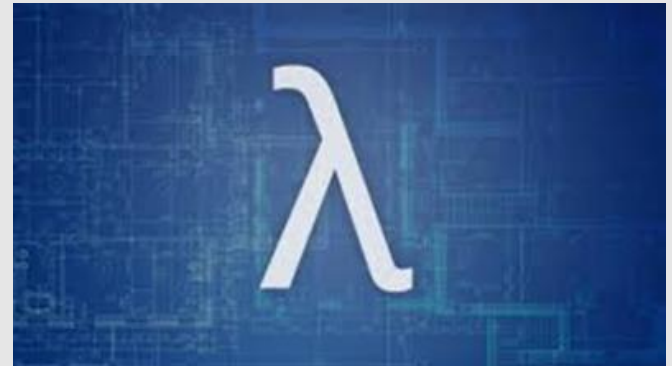
Objetivos da Programação Funcional

- ✓ Focar no emprego de funções matemáticas da forma mais intensa possível;
- ✓ Isso resulta numa abordagem totalmente diferente da Programação Imperativa;



Programação Funcional x Imperativa

- ✓ Programas em linguagens funcionais são **definições de funções e especificação de aplicação de funções**;
- ✓ A execução desses programas corresponde à **avaliação da aplicação das funções**;
- ✓ A execução de uma função **sempre produz o mesmo resultado** para os mesmos parâmetros de entrada.
- ✓ Essa característica dos programas funcionais é chamada **Transparência Referencial**.



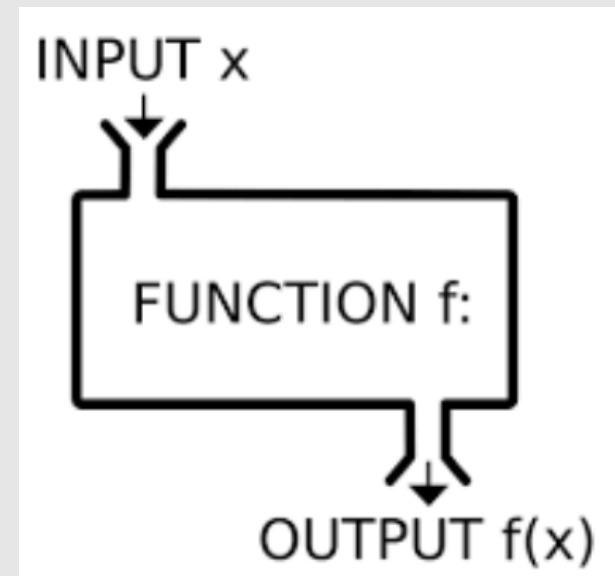
Linguagens Funcionais

- ✓ Provêem um **conjunto de funções primitivas**;
- ✓ Provêem um conjunto de **formas funcionais** (Higher Order Functions) para que funções complexas possam construídas a partir das primitivas;
- ✓ Provêem uma operação de **aplicação da função**;
- ✓ Provêem algumas **estruturas** para representar **dados**. Essas estruturas são usadas para representar parâmetros e valores computados pelas funções;



Programação Funcional em JavaScript

- ❖ **JavaScript** suporta, em parte, o Paradigma Funcional;
- ❖ **Não** é uma linguagem puramente funcional, mas possui muitas funcionalidades que a permitem atuar como se fosse. Esses recursos incluem:
 - ✓ Funções como objetos e primeira classe;
 - ✓ Recursão;
 - ✓ Closures;
 - ✓ Arrow Functions;
 - ✓ Spread.



Funções como objetos de primeira classe

- ❖ **Funções** podem ser tratadas da mesma forma que qualquer outro tipo de dado, como **strings**, **números** ou **objetos**;



Funções como objetos de primeira classe

1. Atribuindo funções a variáveis:

javascript

```
const saudacao = function(nome) {  
    return "Olá, " + nome + "!";  
};  
console.log(saudacao("Alice")); // "Olá, Alice!"
```


Funções como objetos de primeira classe

2. Passando funções como argumentos:

Array.map()

Transformando cada elemento de um array:

javascript

```
const numbers = [1, 2, 3, 4];  
const doubled = numbers.map(function(num) {  
    return num * 2;  
});  
console.log(doubled); // [2, 4, 6, 8]
```

Funções como objetos de primeira classe

3. Retorno de Funções: Uma função pode retornar outra função.

javascript

```
function criarFuncao() {
    return function() {
        console.log("Função interna.");
    };
}

const novaFuncao = criarFuncao();
novaFuncao(); // Exibe: "Função interna."
```

Funções como objetos de primeira classe

4. **Armazenar em Estruturas de Dados:** Funções podem ser armazenadas em arrays, objetos e outras estruturas de dados.

javascript

```
const funcoes = [  
  function() { console.log("Função A"); },  
  function() { console.log("Função B"); }  
];  
  
funcoes[1](); // Exibe: "Função B"
```

Funções como objetos de primeira classe

5. **Propriedades e Métodos:** Por serem objetos, funções também podem ter propriedades e métodos.

javascript

```
function contador() {  
    // Aumenta o valor da contagem toda vez que a função é chamada  
    contador.contagem++;  
    return contador.contagem;  
}  
  
// Inicialmente define a propriedade "contagem" como 0  
contador.contagem = 0;  
  
console.log(contador()); // 1  
console.log(contador()); // 2
```

Funções como objetos de primeira classe

5. **Propriedades e Métodos:** Por serem objetos, funções também podem ter propriedades e métodos.

Assim como os objetos, as funções podem também ter métodos.

javascript

```
function cumprimentar() {  
    console.log("Olá!");  
}  
  
cumprimentar.despedir = function() {  
    console.log("Tchau!");  
}  
  
cumprimentar();          // Olá!  
cumprimentar.despedir(); // Tchau!
```

Funções como objetos de primeira classe

6. **Construtor `Function`:** Em JavaScript, funções são na verdade objetos criados pelo construtor ``Function``.

javascript

```
const funcaoConstruida = new Function('a', 'b', 'return a + b');  
console.log(funcaoConstruida(3, 4)); // Exibe: 7
```

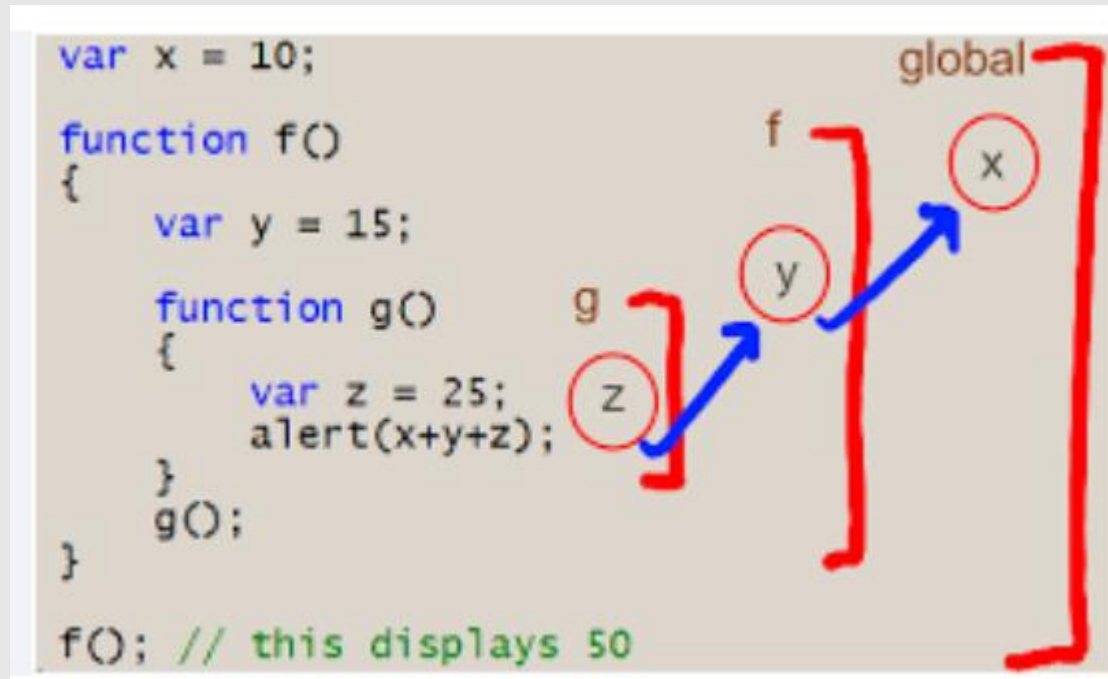
Suporte à Funções Recursivas

javascript

```
function fatorial(n) {  
    if (n <= 1) {  
        return 1;  
    } else {  
        return n * fatorial(n - 1);  
    }  
}  
  
console.log(fatorial(5)); // Isso imprimirá 120
```

Closures

- ❖ **Closures** em JavaScript são uma característica poderosa a qual permite que funções internas tenham acesso à variáveis do escopo da função externa, mesmo após a função externa ter completado sua execução;
- ❖ Em outras palavras, **closures** são funções que “lembram” o ambiente em que foram criadas.



Closures

javascript

```
function cumprimentar(saudacao) {  
    return function(nome) {  
        return saudacao + ', ' + nome + '!';  
    };  
}  
  
let ola = cumprimentar('Olá');  
console.log(ola('Maria')); // Imprime: Olá, Maria!
```

Closures

javascript

```
function cumprimentar(saudacao) {
  return function(nome) {
    return saudacao + ', ' + nome + '!';
  };
}

let ola = cumprimentar('Olá');
console.log(ola('Maria')); // Imprime: Olá, Maria!
```

O que está acontecendo aqui?

A função `cumprimentar` é uma função que aceita um argumento chamado `saudacao`. Quando chamamos essa função, ela retorna **outra função**. A função retornada, por sua vez, aceita um argumento chamado `nome` e retorna uma string que combina `saudacao` e `nome`.

Closures

javascript

```
function cumprimentar(saudacao) {  
    return function(nome) {  
        return saudacao + ', ' + nome + '!';  
    };  
}  
  
let ola = cumprimentar('Olá');  
console.log(ola('Maria')); // Imprime: Olá, Maria!
```

Como funciona o closure aqui?

A parte mágica deste código é que a função retornada ainda tem acesso à variável ``saudacao``, mesmo depois que a função ``cumprimentar`` já foi executada e retornou. Isso é possível por causa dos closures em JavaScript.

Para entender melhor, siga os passos do código:

1. Quando chamamos ``cumprimentar('Olá')``, passamos a string ``'Olá'`` como o argumento para ``saudacao``.
2. Dentro da função ``cumprimentar``, retornamos uma função anônima que aceita um argumento ``nome``.

Closures

javascript

```
function cumprimentar(saudacao) {  
  return function(nome) {  
    return saudacao + ', ' + nome + '!';  
  };  
}  
  
let ola = cumprimentar('Olá');  
console.log(ola('Maria')); // Imprime: Olá, Maria!
```

3. Atribuímos essa função anônima retornada à variável `ola`. Em outras palavras, agora a variável `ola` é uma função que podemos chamar.
4. Quando chamamos `ola('Maria')`, estamos efetivamente chamando a função anônima que retornamos em nosso passo anterior, passando `'Maria'` como o argumento `nome`.
5. Dentro dessa função anônima, ainda temos acesso à variável `saudacao`, que foi definida no escopo da função `cumprimentar`. Esse acesso é possível por causa do closure.
6. Portanto, a função anônima combina `saudacao` e `nome` para retornar a string `'Olá, Maria!'`.

Closures

javascript

```
function cumprimentar(saudacao) {  
    return function(nome) {  
        return saudacao + ', ' + nome + '!';  
    };  
}  
  
let ola = cumprimentar('Olá');  
console.log(ola('Maria')); // Imprime: Olá, Maria!
```

O fato de a função anônima "lembrar" do ambiente (escopo) em que foi criada e ter acesso às variáveis desse ambiente, mesmo depois que o ambiente original (a função `cumprimentar`) terminou de executar, é a essência do que são closures em JavaScript.

Arrow Functions

- ❖ **Arrow functions** foram introduzidas com o **ECMAScript 6** (também conhecido como **ES6** e **ES2015**);
- ❖ Oferecem uma forma mais moderna de se escrever funções mais concisas e têm algumas características particulares quando comparadas às funções tradicionais.

Exemplo de Função Tradicional:

javascript

```
function soma(a, b) {  
    return a + b;  
}
```

Exemplo de Arrow Function:

javascript

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

Arrow Functions

```
javascript
```

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

1. **const soma:** Aqui, estamos declarando uma constante chamada `soma`. Em JavaScript, uma função pode ser atribuída a uma variável, e isso é exatamente o que estamos fazendo. Usamos `const` porque não queremos que o valor dessa variável mude após a atribuição inicial.

Arrow Functions

```
javascript
```

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

2. **(a, b) =>:** Essa é a essência da arrow function. O que está à esquerda da "seta" `(=>)` são os parâmetros da função. Neste caso, temos dois parâmetros: `a` e `b`. O sinal `"=>"` é o que faz isso ser uma "arrow function".
3. **a + b:** Este é o corpo da função. Neste exemplo, temos um "corpo de expressão", o que significa que ele automaticamente retorna o valor da expressão, que é a soma de `a` e `b`. Portanto, não há necessidade de usar a palavra-chave `return`.

Arrow Functions

Exemplo de Função Tradicional:

javascript

```
function soma(a, b) {  
    return a + b;  
}
```

Exemplo de Arrow Function:

javascript

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

A principal diferença aqui é a sintaxe. A função tradicional usa a palavra-chave **'function'** para sua definição, enquanto a arrow function usa a seta **'=>'**.

Porém, o comportamento de ambas as funções é o mesmo nesse contexto: ambas aceitam dois números como argumentos, somam esses números e retornam o resultado.

Arrow Functions

- ❖ **Arrow functions** podem ter um “corpo de expressão”, que devolve o valor da expressão de forma automática;
- ❖ Portanto, nesse caso, **sem** a necessidade de “**return**”.

```
javascript
```

```
const quadrado = x => x * x;
```

Arrow Functions

- ❖ **Arrow functions** podem também ter um “**corpo de bloco**” que **não** retorna automaticamente um valor;
- ❖ Nesse caso, deve-se codificar o “**return**”.

javascript

```
const quadrado = x => {  
    let resultado = x * x;  
    return resultado;  
};
```

Arrow Functions

- ❖ **Arrow functions** são geralmente chamadas de **funções anônimas** devido à sua falta de um nome;
- ❖ Caso seja necessário se referir a uma **arrow function**, terá que se atribuí-la a uma variável ou atributo de um objeto, como feito até aqui; caso contrário, **não** se poderá usá-la.

