



PUC Minas

Linguagem JavaScript

Prof. Marcos André S. Kutova

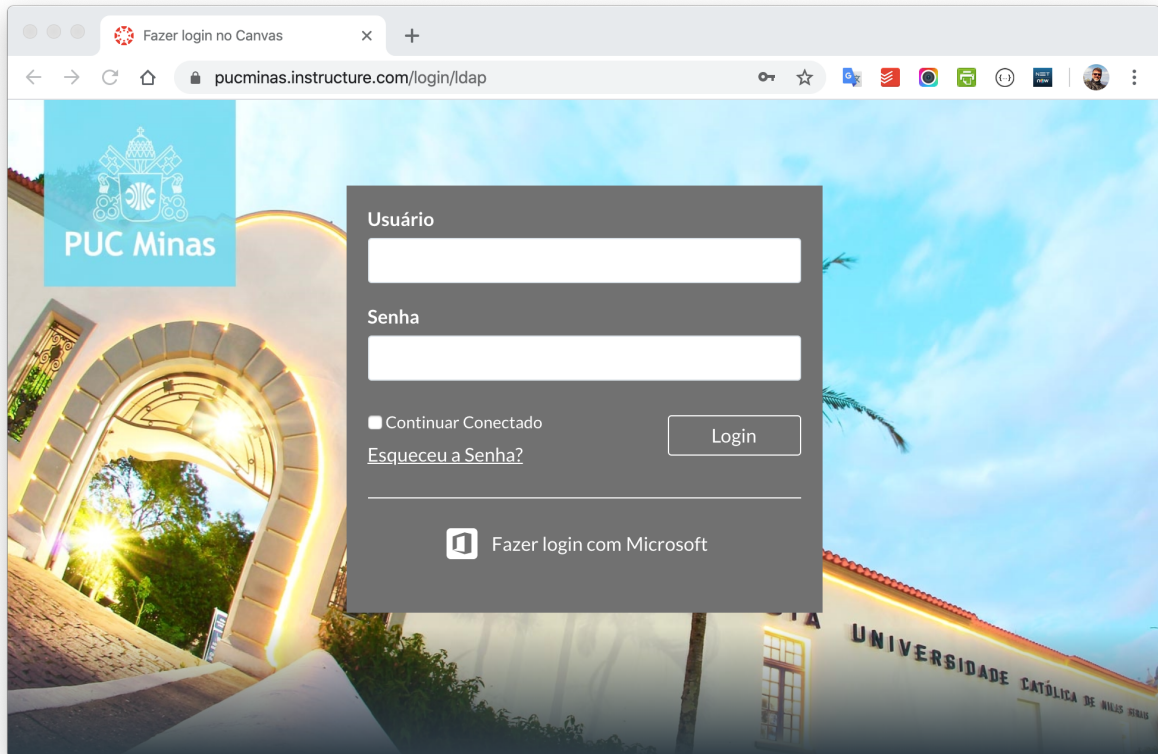
O que é JavaScript?

JavaScript é uma linguagem de programação usada para tornar as páginas web interativas.



Que tipo de interatividade pode ser criada?

Uso mais básico:
Controle de formulários nas páginas web



Que tipo de interatividade pode ser criada?

Alterações de estrutura, conteúdo ou apresentação de qualquer componente de uma página web



Controle de exibição de filmes



Menus e formulários



Envio de fotos e aplicação de filtros



Manipulação das cartas e interação com outros jogadores



Manipulação dos mapas e desenho de rotas



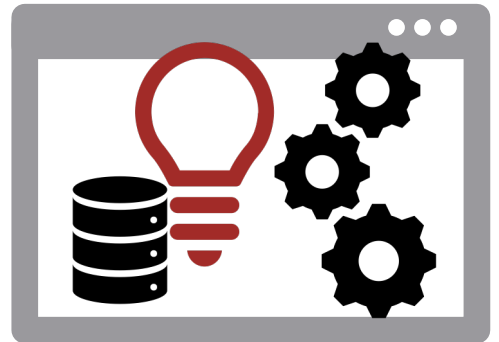
Gerenciamento de mensagens e formatação de texto

Um pouco mais sobre a JavaScript

JavaScript é uma linguagem de *scripts*

Linguagens de *scripts* geralmente são usadas para manipulação, personalização ou automação de recursos já existentes (ao invés de serem usadas para a construção de sistemas completos).

No nosso caso, esse recurso é a página web.



Um pouco mais sobre a JavaScript

JavaScript é executada pelo navegador

A execução dos *scripts* pelo navegador, alivia a sobrecarga do servidor e acelera a interação do usuário com a página.

Mas há algumas limitações de segurança, como o impedimento de acessar arquivos ou programas no seu computador.



Um pouco mais sobre a JavaScript

JavaScript é um produto ECMAScript

<https://www.ecma-international.org/ecma-262/>

ECMAScript é uma especificação de linguagens de *scripts*.



JavaScript é uma implementação dessa linguagem, planejada para rodar nos navegadores. Podem existir diferenças entre as implementações dos navegadores.

O nome JavaScript foi uma jogada de marketing e a linguagem não tem nada a ver com Java.

Como a JavaScript funciona

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Olá Mundo</title>
    <meta charset="utf-8" />
    <script>
      function ola() {
        alert( "Olá mundo!" );
      }
    </script>
  </head>
  <body onload="ola()">
    <p>Olá mundo em JavaScript</p>
  </body>
</html>
```




PUC Minas

Variáveis

Prof. Marcos André S. Kutova

Variáveis em JavaScript

Uma variável é um espaço temporário para armazenamento de alguma informação, durante a execução de algum *script*.

Essa informação poderá ser recuperada a qualquer momento, por meio do nome da variável.

Variáveis em JavaScript

```
let nome;  
nome = "Marcos";  
  
console.log(nome);
```



Por convenção, usamos a forma camelCase nos nomes das variáveis

Uma variável simples só pode conter um valor de cada vez.

Variáveis em JavaScript

```
let nome = "Marcos";  
console.log(nome);
```

A declaração e a atribuição de valor podem ser feitas em um único comando.

Variáveis em JavaScript

As variáveis podem armazenar valores de diversos tipos.

Cada tipo de dado é representado de uma forma diferente nos computadores e isso determina quais valores podem ser armazenados.

```
let nome = "Marcos";  
let idade = 50;  
let casado = true;
```

Declaração e Escopo das Variáveis

As variáveis podem ser declaradas de quatro formas diferentes:

1. Por meio da instrução **let**
Essa instrução declara uma variável que só é visualizada dentro do bloco em que foi declarada

```
{  
    let nome = "Marcos";  
    console.log(nome); // Marcos  
}  
console.log(nome); // Erro: variável não  
                    existe
```

Declaração e Escopo das Variáveis

As variáveis podem ser declaradas de quatro formas diferentes:

2. Por meio da instrução **var**
Essa instrução declara uma variável cujo escopo é de uma função ou global

```
{  
    var nome = "Marcos";  
    console.log(nome); // Marcos  
}  
console.log(nome);      // Marcos
```

Declaração e Escopo das Variáveis

As variáveis podem ser declaradas de quatro formas diferentes:

3. Sem nenhum termo
A variável terá escopo global (mesmo que declarada em uma função).

Essa prática não é recomendável!

```
{  
    nome = "Marcos";  
    console.log(nome);    // Marcos  
}  
console.log(nome);        // Marcos
```


Declaração e Escopo das Variáveis

As variáveis podem ser declaradas de quatro formas diferentes:

4. Por meio da instrução **const**

Essa instrução declara uma constante - o seu valor não pode ser alterado.

O escopo da constante é o bloco em que foi declarada.

Por convenção, os nomes das constantes são em letras maiúsculas.

```
{  
    const PI = 3.14159;  
    console.log(PI);  
}
```



PUC Minas

Tipos de Dados

Prof. Marcos André S. Kutova

Tipos de dados

Uma variável em JavaScript pode ser de um entre vários tipos: números, *strings*, lógicos, objetos, vetores, etc.

JavaScript é uma linguagem **fracamente tipada**. Isso significa que não precisamos informar o tipo da variável na sua declaração e que ele pode ser alterado conforme a necessidade.

```
let v;  
v = 'JavaScript';  
v = 31;
```

Teste do tipo de uma variável

Você pode descobrir o tipo de uma variável usando o operador **typeof**

```
let nome    = 'Marcos';  
let casado  = true;  
let filhos  = 3;
```

```
console.log( typeof nome );    // string  
console.log( typeof casado );  // boolean  
console.log( typeof filhos );  // number
```

Tipos básicos de dados

Números

O tipo `number` é usado para representar tanto números inteiros quanto números reais. Variáveis desse tipo podem ser usadas em expressões matemáticas.

```
let i = 15;  
let r = 3.14159;
```

Valores especiais:

- `Infinity`
- `-Infinity`
- `NaN`

Tipos básicos de dados

Números inteiros muuuiiiitto grandes

O tipo **BigInt** é usado em algumas operações especiais, como as de criptografia. Esses números são representados com um **n** no fim. Nem todos os navegadores oferecem suporte a BigInts.

```
let giga = 1234567890123456789012345678901234567890n;
```

Tipos básicos de dados

Strings

Uma **string** é qualquer tipo de texto, inclusive de um único carácter (não existe o tipo `char` em JavaScript). Existem três formas de delimitação de uma string:

let nome1 = "Joana";	→ Forma mais simples
let nome2 = 'Joana';	→ Forma para combinar HTML e JS
let nome3 = `Joana`;	→ Forma de <i>template string</i>

Tipos básicos de dados

Template Strings

Uma *template string* tem duas vantagens. A primeira é a possibilidade de criação de uma *string* que tem várias linhas.

```
let poema = `O Poeta é um fingidor.  
Finge tão completamente  
Que chega a fingir que é dor  
A dor que deveras sente.`;
```

Fernando Pessoa

Tipos básicos de dados

Template Strings

Isso é muito útil quando a *string* contém código HTML que fica mais claro quando organizado em várias linhas.

```
let conteudo =  
  `

<p>Um parágrafo</p>  
  </div>`;


```

Tipos básicos de dados

Template Strings

A segunda vantagem da *template string* é permitir a incorporação de outra *string* ou expressão, por meio dos delimitadores `${}` e ```.

```
let nome = 'Marcos';  
let mensagem = `Olá ${nome}!`;   
console.log(mensagem);  
  
let a = 3, b = 5;  
console.log(`A soma de ${a} e ${b} é ${a+b}.`);
```

Tipos básicos de dados

Lógicos

O tipo `boolean` possui apenas dois valores: `true` e `false`.

```
let casado = true;  
let temFilhos = false;  
let adulto = idade >= 18;
```

Valores especiais

As variáveis em JavaScript podem assumir dois valores especiais:

null

O valor **null** significa sem valor ou vazio. Pode ser usado como forma de se limpar ou apagar o valor de uma variável.

```
let idade = null;
```

undefined

Variáveis ainda não inicializadas possuem o valor **undefined**.

```
let idade; //undefined
```



PUC Minas

Operadores

Prof. Marcos André S. Kutova

Tipos de operadores

Os operadores podem ser unários ou binários.
Operadores unários possuem apenas um operando e
operadores binários possuem dois operandos.

```
let d = -a;  
let e = b+c;
```

Atribuição

Atribui um valor a uma variável ou constante

```
let a = 5;  
let nome = 'José';
```

Operadores unários

Inverter o valor de uma variável ou número

```
let a = -c;  
let b = -3;
```

Assegurar um valor numérico

```
let n = +"12"; // 12
```

Incremento e decremento

```
b = --a * 2; // pré-decremento: a=a-1, b=a*2  
b = a-- * 2; // pós-decremento: b=a*2, a=a-1  
b = ++a * 2; // pré-incremento: a=a+1, b=a*2  
b = a++ * 2; // pós-incremento: b=a*2, a=a+1
```


Operadores matemáticos

Operação	Operador	Exemplo
Adição	+	let a = 5 + 2; // 4
Subtração	-	let a = 5 - 2; // 2
Multiplicação	*	let a = 5 * 2; // 10
Divisão	/	let a = 5 / 2; // 2.5
Resto da divisão	%	let a = 5 % 2; // 1
Exponenciação	**	let a = 5 ** 2; // 25

Concatenação de strings

A concatenação de *strings* é a junção de duas *strings* em uma nova *string*.

```
let nome = 'José';  
let sobrenome = 'Pereira';  
let nomeCompleto = nome + ' ' + sobrenome;  
                        // 'José Pereira'
```

Coerção

Os resultados de uma expressão em que números estão entre aspas (*strings*) serão diferentes de acordo com a combinação de operandos e operadores:

```
let a = "5" + 2;           // 52
let b = +"5" + 2;          // 7
let c = "5" - 2;           // 3
let d = 5 + "2";           // 52
let e = 5 - "2";           // 3
let f = 5 + 2 + "2";       // 72
let g = 5 + "2" + 2;       // 522
let h = +(5 + "2") + 2;    // 54
let i = "5" / "2";         // 2.5
```

Comparações

Comparações são operações que sempre retornam um valor lógico: falso (**false**) ou verdadeiro (**true**).

```
if( i>0 ) {  
    // sequência de  
    comandos  
}
```

Operadores relacionais

Operação	Operador	Exemplo
Maior que	>	<code>5 > 2; // true</code>
Menor que	<	<code>5 < 2; // false</code>
Maior ou igual a	>=	<code>2 >= 2; // true</code>
Menor ou igual a	<=	<code>1 <= 2; // true</code>
Igual	==	<code>5 == 2; // false</code>
Diferente	!=	<code>5 != 2; // true</code>

Comparações de strings

A comparação entre *strings* nos permite testar se uma delas é igual, menor ou maior que a outra. Para isso, considera a ordem das letras no sistema Unicode.

```
console.log( 'Z' > 'A' );           // true
console.log( 'Pato' > 'Pata' );     // true
console.log( 'Dezena' > 'Dez' );    // true
console.log( 'a' > 'A' );           // true
```


Comparações de tipos diferentes

O operador `===` (*estritamente igual*) compara valores e tipos.

```
console.log( "1" == 1 );           // true
console.log( true == 1 );           // true
console.log( ' ' == false );        // true
```

```
console.log( "1" === 1 );           // false
console.log( true === 1 );           // false
console.log( ' ' === false );        // false
```


null e undefined

É muito comum fazermos testarmos variáveis e objetos comparando-os com **null** e **undefined**, mas é importante saber que eles são valores de tipos diferentes.

```
console.log(null == undefined); // true
console.log(null === undefined); // false
```

```
let a; // variável sem valor ou undefined
console.log(a == undefined); // true
console.log(a == null); // true
console.log(a === undefined); // true
console.log(a === null); // false
```

Operadores lógicos

Operação	Operador	Exemplo
Conjunção (AND)	&&	<code>a > b && a != 0</code>
Disjunção (OR)		<code>a > b a != 0</code>
Negação (NOT)	!	<code>!(a > b)</code>



PUC Minas

Controle do fluxo

Prof. Marcos André S. Kutova

Controle do fluxo de execução

O controle do fluxo de execução é a determinação de qual instrução deve ser executada a seguir. Um fluxo de execução geralmente é linear, sequencial. Isso significa que uma instrução será executada logo após a outra:

```
let i = 15;  
let r = 2;  
console.log(i*r);
```




*Nesse exemplo, cada
instrução será executada
na ordem em que
aparece no código.*

Declaração condicional

É possível estabelecer condições para a execução de determinadas instruções, por meio da declaração **if**.

As chaves, { e }, são usadas para delimitar um bloco de instruções que só será executado se a condição for verdadeira.

```
let acessoPermitido = false;  
let idade = 21;  
if( idade >= 18 ) {  
    console.log( "acesso permitido" );  
    acessoPermitido = true;  
}
```



Declaração condicional

É possível estabelecer condições para a execução de determinadas instruções, por meio da declaração **if**.

```
let idade = 21;  
if( idade >= 18 )  
    console.log( "acesso permitido" );
```

*Quando o bloco tem
apenas 1 única instrução,
as chaves são
desnecessárias.*



Declaração condicional

A declaração **else** permite delimitar instruções que só serão executadas quando a condição for falsa.

```
let acessoPermitido;  
let idade = 21;  
if( idade >= 18 )  
    acessoPermitido = true;  
else  
    acessoPermitido = false;
```

Condições nas expressões

O operador **?** permite que criemos expressões com condições embutidas, sem a necessidade da estrutura condição completa do **if**.

O operador ? separa a condição dos valores de retorno

```
let idade = 21;
```

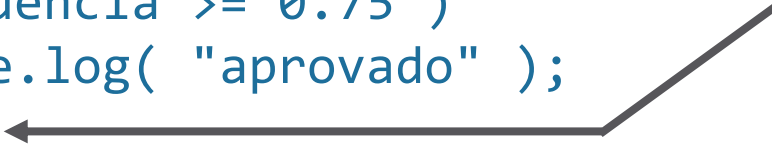
```
let acessoPermitido = (idade >= 18) ? true : false;
```

O operador : separa o valor a ser retornado quando a expressão for verdadeira e o valor a ser retornado quando a expressão for falsa.

Estruturas condicionais encadeadas

É comum precisarmos combinar várias estruturas **if** para testarmos diversas possibilidades de uma só vez.

```
if( nota >= 60 ) {  
    if( frequência >= 0.75 )  
        console.log( "aprovado" );  
    else  
        console.log( "reprovado por frequência" );  
}  
else  
    console.log( "reprovado" );
```



*O uso das chaves elimina o risco de não se saber a que **if** esse **else** pertence.*

Teste de vários valores

Eventualmente, precisamos testar vários possíveis valores de variáveis ou resultados de expressões.

```
let a = 2, b = 1, operador = '+';  
if( operador == '+' )  
    console.log( a+b );  
else if( operador == '-' )  
    console.log( a-b );  
else if( operador == '*' )  
    console.log( a*b );  
else if( operador == '/' )  
    console.log( a/b );  
else  
    console.log( "operador inválido" );
```

Declaração switch

A declaração **switch** permite o teste de vários valores de variáveis ou expressões.


```
let a = 2, b = 1, operador = '+';
switch( operador ) {
  case '+':
    console.log( a+b );
    break;
  case '-':
    console.log( a-b );
    break;
  case '*':
  case '/':
    console.log( "operador não implementado" );
    break;
  default:
    console.log( "operador inválido" );
}
```

Declaração de repetição

Algumas vezes, precisamos repetir a execução de determinado conjunto de instruções. Podemos criar um **laço de repetição** isso usando a declaração **while**.

```
let i = 0;  
while( i<3 ) {  
  console.log( i );  
  i++;  
}
```

*O bloco será executado
se e enquanto a
condição for verdadeira.*



Declaração de repetição

Se precisarmos executar o bloco pelo menos uma vez, podemos ver a condição para o fim do bloco com a declaração **do...while**.

```
let i = 0;  
do {  
  console.log( i );  
  i++;  
} while( i<3 );
```

Declaração de repetição

A declaração **for** permite a criação de laços que serão repetidos um determinado número de vezes

início *condição* *passo*
↓ ↓ ↓

```
for( let i=0; i<10; i++ ) {  
    if( i%2 == 0 )  
        console.log( i + " é um número par" );  
}
```

Quebra do fluxo normal do laço

Há duas instruções que permitem suspender uma repetição ou encerrar todo o laço.

A instrução **continue** suspende a repetição atual.

```
for(let i=0;i<10;i++) {  
  if(i%7 == 0)  
    continue;  
  console.log(i);  
}
```

A instrução **break** suspende todo o laço.

```
for(let i=0;i<10;i++) {  
  if(i%7 == 0)  
    break;  
  console.log(i);  
}
```



PUC Minas

Funções

Prof. Marcos André S. Kutova

Funções

Quando temos um trecho de código que deverá ser executado várias vezes, podemos “encapsulá-lo” em um bloco de instruções. Esse bloco recebe um nome e pode ser executado por meio desse nome.

```
function mensagem() {  
  console.log('Olá mundo');  
}
```

*Declaração
da função*

```
mensagem();
```

*← Chamada ou
invocação*

Parâmetros

Uma função pode receber valores como parâmetros. Esses parâmetros podem alterar o processamento ou o resultado da função.

```
function mensagem(nome) {  
    console.log('Olá ' + nome);  
}
```

```
mensagem('Marcos');
```

```
function soma(a, b) {  
    console.log(a+b);  
}
```

```
soma(3, 4);
```

Valor de retorno

Uma função também pode retornar um resultado como resposta à sua chamada.

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

```
let total = soma(3, 4);  
console.log( total );
```

Valor padrão do parâmetro

Os parâmetros podem ter um valor padrão, caso não sejam informados.

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

```
console.log(soma(2,3)); // 5  
console.log(soma(2));   // NaN  
console.log(soma());    // NaN
```

```
function soma(a=0, b=0) {  
  return a+b;  
}
```

```
console.log(soma(2,3)); // 5  
console.log(soma(2));   // 2  
console.log(soma());    // 0
```

Vetor de parâmetros

Toda função tem um objeto interno `arguments`

```
function soma() {  
  console.log(arguments); // { '0': 2, '1': 3 }  
  return arguments[0]+arguments[1];  
}  
  
console.log(soma(2,3)); // 5
```

Variáveis locais

Uma função declarar variáveis para apoiar as suas operações. Essas variáveis só existirão enquanto a função estiver sendo executada.

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

```
console.log( soma(3, 4) ); // 7  
console.log( resultado ); // ReferenceError
```

Variáveis globais

Uma função pode acessar variáveis globais, isto é, variáveis declaradas fora do escopo de qualquer bloco.

```
let nome = 'Marcos';
```

```
function mensagem() {  
  console.log('Olá ' + nome);  
}
```

```
mensagem();
```

Variáveis globais

... desde que o nome da variável não seja sobreposto localmente na função.

```
let nome = 'Marcos';

function mensagem() {
  let nome='Ana';
  console.log('Olá ' +nome);
}

mensagem();
```

```
let nome = 'Marcos';

function mensagem(nome) {
  console.log('Olá ' +nome);
}

mensagem('Ana');
```


Expressão de função

Uma função pode ser considerada uma expressão e, assim, ser atribuída a uma variável ou usada como parâmetros de outras funções.

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

← Função anônima

```
console.log( soma(3, 4) ); // 7
```



*Podemos fazer a chamada
da função por meio da
variável que a referencia*

Expressão de função

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

```
let executa = function(fn, op1, op2) {  
  return fn(op1, op2);  
}
```

```
let resultado = executa(soma, 3, 4);  
console.log( resultado ); // 7
```

Expressão de função nomeada

```
let f = function fatorial(n) {  
  if(n<=1)  
    return 1;  
  return n*fatorial(n-1);  
}
```

*O nome é usado
internamente, nas
chamadas recursivas*

```
console.log( f(4) ); // 24
```

Arrow functions (funções seta)

Uma função seta é apenas uma forma simplificada de declaração de funções que retornam valores.

```
let soma = function(a, b) {  
  return a+b;  
}  
  
console.log(soma(2, 3));
```

```
let soma = (a,b) => a+b;  
  
console.log(soma(2, 3));
```



PUC Minas

Vetores

Prof. Marcos André S. Kutova

Vetores

Vetores são estruturas que nos permitem associar um conjunto de valores (e não apenas um) a variáveis. Podemos usar no nosso código tanto o vetor, como uma coleção de valores, quanto um valor individual, por meio da sua posição.

```
let frutas = ['laranja', 'maçã', 'banana'];
```

```
console.log(frutas); // ['laranja', 'maçã', 'banana']
```

```
console.log(frutas[0]); // laranja
```

```
console.log(frutas[1]); // maçã
```

```
console.log(frutas[2]); // banana
```

← A primeira
posição em um
vetor é a de
número 0.

Vetores

Os valores do vetor podem ser alterados, bem como novos valores podem ser acrescentados à coleção.

```
let frutas = ['laranja', 'maçã', 'banana'];
```

```
frutas[2] = 'pêra';  
console.log(frutas[2]); // pêra
```

```
frutas[3] = 'limão';  
console.log(frutas[3]); // limão
```

```
console.log(frutas);  
// ['laranja', 'maçã', 'banana', 'limão']
```

Atenção

Não há controle interno para que o acréscimo de novos elementos seja feito de forma organizada.

```
let frutas = ['laranja', 'maçã', 'banana'];  
frutas[5] = 'limão';  
  
console.log(frutas);  
// ['laranja', 'maçã', 'banana',,,, 'limão']
```


Tamanho do vetor

Muitas vezes, precisamos saber qual é o número de elementos em um vetor. Usamos a propriedade **length** para isso.

```
let frutas = ['laranja', 'maçã', 'banana'];  
console.log(frutas.length); // 3
```

```
for(let i=0; i<frutas.length; i++)  
    console.log(frutas[i]);
```

Tamanho do vetor

Usando a propriedade **length**, podemos acrescentar elementos no fim do vetor sem risco de deixar elementos intermediários sem valor

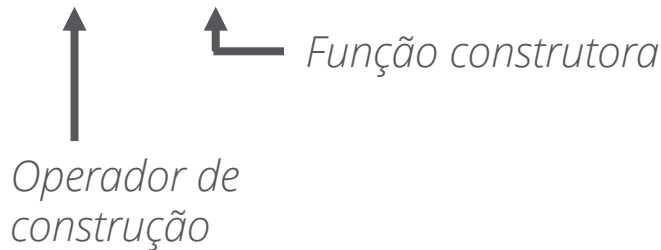
```
let frutas = ['laranja', 'maçã', 'banana'];  
frutas[frutas.length] = 'limão';  
  
console.log(frutas);  
// ['laranja', 'maçã', 'banana', 'limão']
```

Vetores vazios

Quando precisamos apenas declarar um vetor, ainda sem acrescentarmos valores a eles, podemos usar uma dessas duas formas:

```
let frutas = [];
```

```
let frutas = new Array();
```



Operador de construção

Função construtora

Métodos

A diferença entre variáveis e vetores não se limita a eles armazenarem mais valores. Eles também possuem algumas funções embutidas chamadas de métodos.

```
let frutas = ['laranja', 'maçã', 'banana', 'limão'];  
  
frutas.sort();  
console.log(frutas);  
// ['banana', 'laranja', 'limão', 'maçã']
```

Operações de inclusão e remoção

Remoção do fim do vetor

```
let frutas = ['laranja', 'maçã', 'banana'];  
console.log(frutas);    // ['laranja', 'maçã', 'banana']  
  
console.log(frutas.pop()); // banana  
console.log(frutas);    // ['laranja', 'maçã']
```

Operações de inclusão e remoção

Inclusão no fim do vetor

```
let frutas = ['laranja', 'maçã', 'banana'];  
console.log(frutas);    // ['laranja', 'maçã', 'banana']  
  
frutas.push('limão');  
console.log(frutas);  
                        // ['laranja', 'maçã', 'banana', 'limão']
```

Operações de inclusão e remoção

Remoção no início do vetor

```
let frutas = ['laranja', 'maçã', 'banana'];  
console.log(frutas);    // ['laranja', 'maçã', 'banana']  
  
console.log(frutas.shift()); // laranja  
console.log(frutas);    // ['maçã', 'banana']
```

Operações de inclusão e remoção

Inclusão no início do vetor.

```
let frutas = ['laranja', 'maçã', 'banana'];  
console.log(frutas);    // ['laranja', 'maçã', 'banana']  
  
frutas.unshift('limão');  
console.log(frutas);  
    // ['limão', 'laranja', 'maçã', 'banana']
```




PUC Minas

Objetos

Prof. Marcos André S. Kutova

Objetos

Objetos são estruturas similares aos vetores. No entanto, ao invés dos valores serem associados a uma posição no conjunto, eles são identificados por meio de um nome.



```
{  
  nome: 'João',  
  idade: 35  
}
```

↑ ↑
chave *valor*

```
{  
  música: 'I'm Eighteen',  
  artista: 'Alice Cooper',  
  segundos: 180  
}
```

Objetos

A declaração de um objeto é feita por uma *literal* de objeto (uma sequência de caracteres com os dados do objeto).

- O objeto é delimitado por chaves
- Os pares de chaves e valores são separados por vírgulas
- As chaves são strings
- Os valores podem ser de qualquer tipo, inclusive outros objetos, vetores, funções, ...

```
let usuário = {  
    nome: 'João',  
    idade: 35  
}
```

Objetos vazios

As chaves e valores não precisam ser criadas na declaração do objeto, mas podem ser acrescentadas mais tarde. Assim, podemos criar objetos vazios, que receberão seus valores posteriormente

```
let música = {};
```

```
let música = new Object();
```



Acréscimo de propriedades

Podemos acrescentar propriedades a objetos vazios (ou que já tenham outras propriedades) das seguintes formas:

```
let usuário = {};  
usuário.nome = 'João';  
usuário.idade = 35;
```

```
let usuário = {};  
usuário["nome"] = 'João';  
usuário["idade"] = 35;
```

Usando variáveis

Ao usarmos os colchetes, podemos alterar dinamicamente a propriedade a ser usada:

```
let usuário = {};  
  
let prop = 'nome';  
usuário[prop] = 'João';  
  
console.log(usuário[prop]);
```

Testando se uma propriedade existe

O operador **in** nos permite testar se uma propriedade existe no objeto:

```
let usuário = {  
  nome: "João",  
  idade: 35  
};  
  
if('idade' in usuário)  
  console.log(  
    `${usuário.nome} tem ${usuário.idade} anos.`  
  );  
  // João tem 35 anos.
```

Usando todas as propriedades

É possível percorrer um objeto, acessando todas as suas propriedades:

```
let usuário = {  
  nome: "João",  
  idade: 35  
};  
  
for(let chave in usuário)  
  console.log(`${chave} = ${usuário[chave]}`);
```


Cópia da referência do objeto

Quando copiamos um objeto, estamos apenas criando uma nova referência para ele (e não um novo objeto)

```
let usuário = {  
  nome: "João",  
  idade: 35  
};
```

```
let cliente = usuário;  
console.log(cliente.nome, cliente.idade); // João 35
```

```
cliente.nome = "Maria";  
console.log(cliente.nome); // Maria  
console.log(usuario.nome); // Maria
```

Clonagem de objetos

Uma forma de se criar um novo objeto a partir de um existente, é fazendo isso propriedade a propriedade:

```
let usuário = {  
  nome: "João",  
  idade: 35  
};
```

```
let cliente = {};  
for(let chave in usuário)  
  cliente[chave] = usuário[chave];
```

Função para criar um objeto

Outra técnica sofisticada é criar uma função que retorna um objeto:

```
function criaUsuário(n, i) {  
  let maior = i >= 18;  
  return {  
    nome: n,  
    idade: i,  
    maiorIdade: maior  
  };  
}  
  
let usuário = criaUsuário('João', 35);  
console.log(usuário);
```

Propriedades que são funções

```
function criaUsuário(n, i) {  
  return {  
    nome: n,  
    idade: i,  
    maiorIdade: function() {  
      return this.idade >= 18;  
    }  
  };  
}
```

↑
Referência ao próprio objeto

```
let usuário = criaUsuário('João', 35);  
console.log(usuario.maiorIdade());
```

Função construtora

```
function Usuário(n, i) {  
  this.nome = n;  
  this.idade = i;  
  this.maiorIdade = function() {  
    return this.idade >= 18;  
  }  
}  
  
let usuário = new Usuário('João', 35);  
console.log(usuario.maiorIdade());
```

Classes

```
class Usuário {  
  
    constructor(n, i) {  
        this.nome = n;  
        this.idade = i;  
    }  
  
    maiorIdade() {  
        return this.idade >= 18;  
    }  
}  
  
let usuário = new Usuário('João', 35);  
  
console.log(usuario);  
console.log(usuario.maiorIdade());
```



PUC Minas