



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Fundamentos de JavaScript

QXD0020 - Desenvolvimento de Software para Web

Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Agenda

- Introdução
- Visão geral
- Objetos e Funções

Introdução



Introdução

O que é JavaScript?

- Uma linguagem de scripts e interpretada criada em meados da década de 90 pela Netscape Communications
- Multiparadigma, da suporte a programação funcional e imperativa
- Possui tipagem dinâmica
- Uma das três principais tecnologias da World Wide Web
- Desde 2013 é a linguagem mais popular de acordo com o [StackOverflow Survey](#)

Introdução

Javascript

- Inicialmente foi criada atender à demanda crescente por sites mais interativos e dinâmicos
 - Inserir texto dinamicamente no código HTML
 - Reagir a eventos (ex: carregamento da página, cliques do usuário)
 - Pegar informações sobre o computador do usuário(ex: navegador)
 - Realizar cálculos no computador do usuário (Ex: validação de formulário)

Introdução

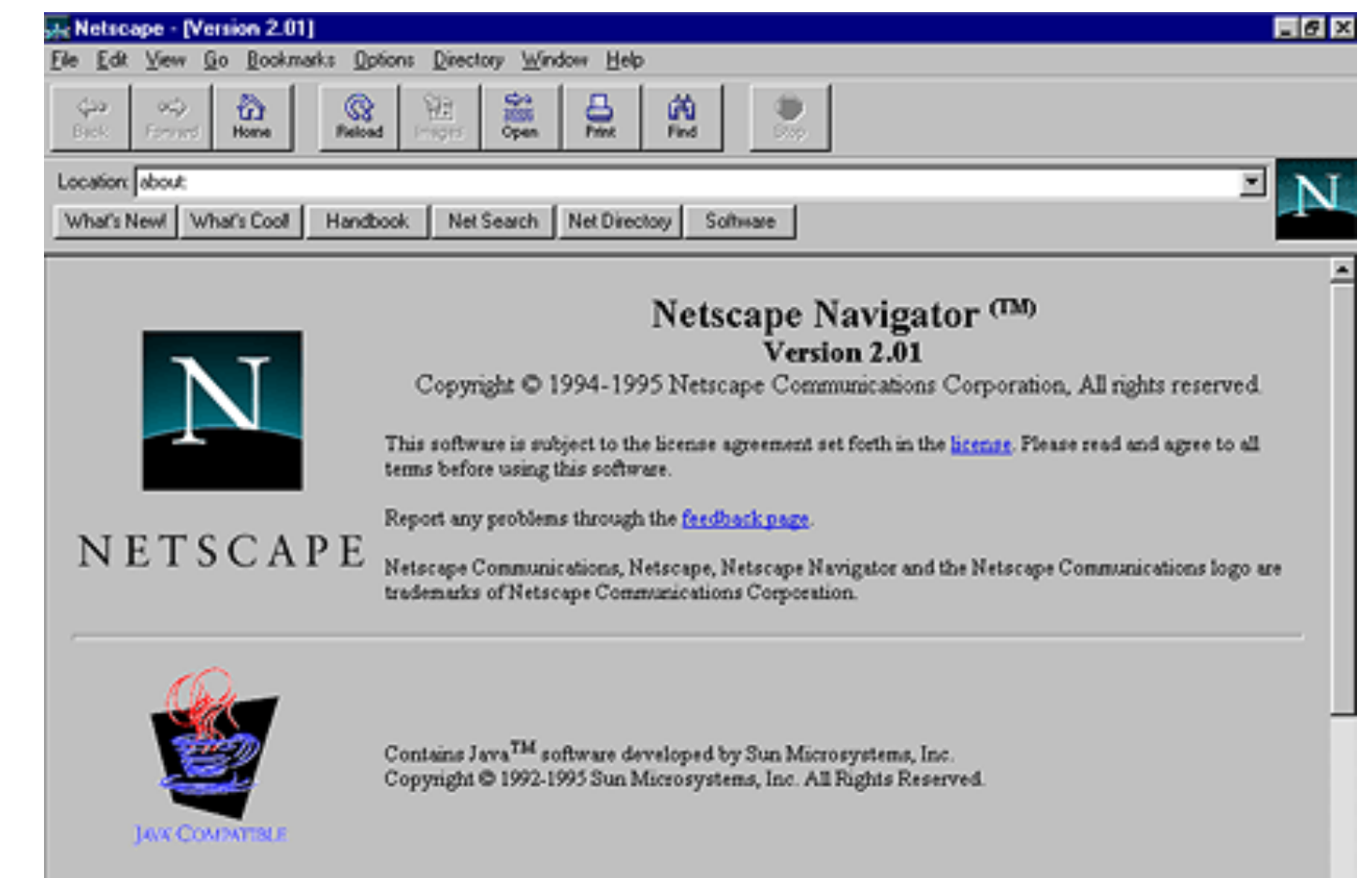
Javascript

- Atualmente pode ser utilizado no lado do servidor “back-end”
 - Ganhou popularidade em 2019 graças ao NodeJs
- Também pode ser utilizado para o desenvolvimento de aplicativos móveis

Introdução

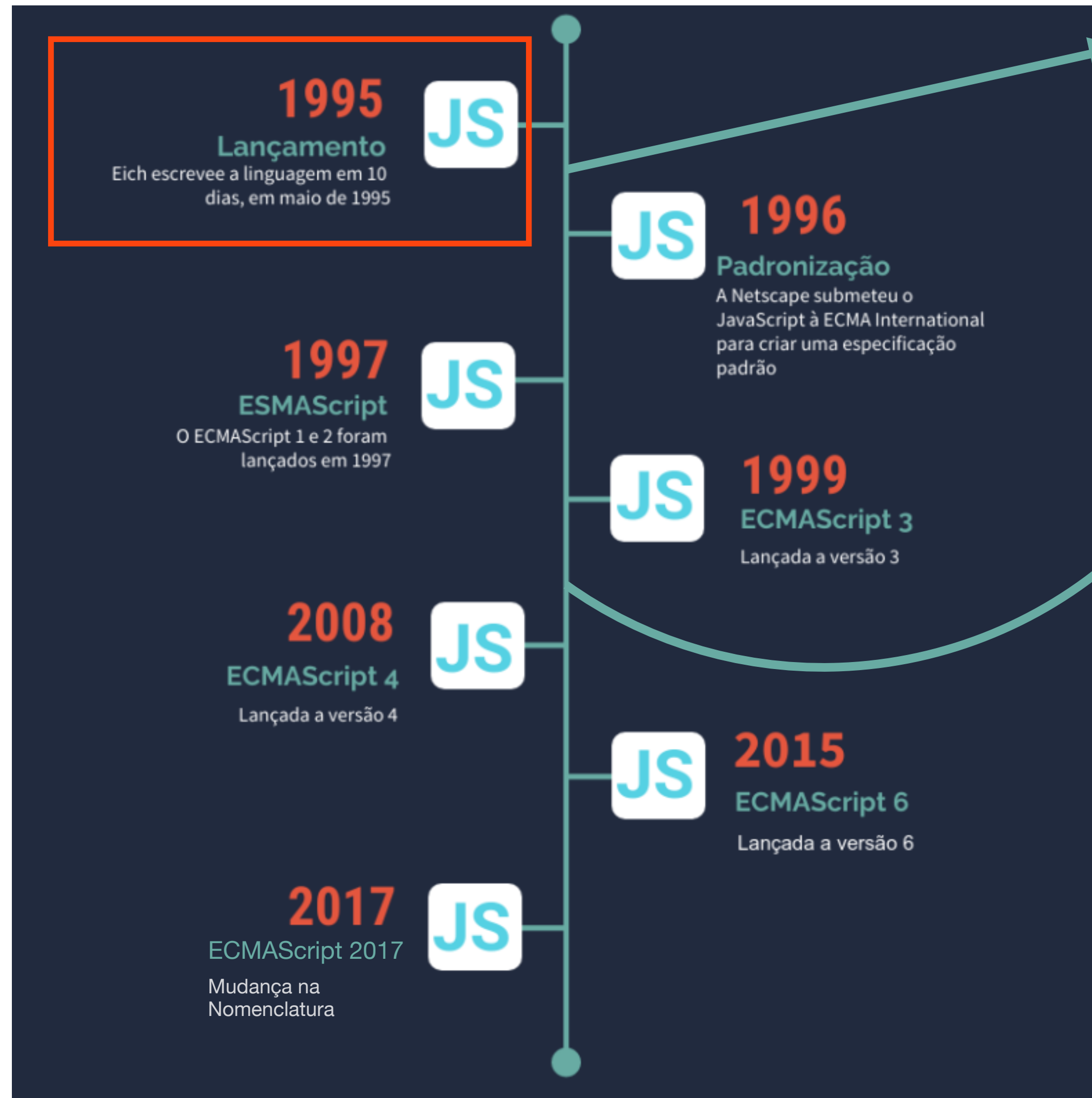
Origem

- Criada por Brendan Eich em 1995 para a **Netscape**
 - Originalmente chamado **LiveScript**
 - Mudou de nome por decisão de *marketing* apoiada na popularidade da linguagem **Java**
 - Foi lançada no **Netscape Navigator 2.0 beta 3**
- Originalmente implementada como parte dos navegadores
 - Permitir a execução de scripts do lado do cliente
 - Permitia a interação com o usuário sem a intervenção do servidor



Introdução

Evolução



- Implementado como **JScript** pela Microsoft por meio de eng. reversa
 - Internet Explorer 3
 - Tornou difícil o funcionamento de um site em diferentes navegadores
- ECMA Script 4 começa a ser desenvolvido em 2000
 - Apesar de implementar parte da especificação, a Microsoft não tinha intenção de cooperar
 - Lançado em 2008 após o trabalho conjunto entre a Mozilla, Macromedia (ActionScript) e Brendan Eich
- ECMA Script 5 lançado em 2009 selou a paz entre as empresas
- Passou a contar com uma versão anual

Introdução

Principais features recentemente adicionadas

- Let/const
- Arrow Functions
- Template literal
- Funções novas em Arrays
- Default Parameters
- Property Shorthand

Visão geral



Visão geral

Tipos de dados

- Number
- Null
- Undefined
- String
- Boolean
- Object
- BigInt
- Symbol

Visão geral

O tipo number

- Não há separação entre números inteiros e números reais
 - Ambos são do mesmo tipo, **number**
 - São sempre pontos flutuante de 64 bits
 - Números inteiros são precisos até 15 dígitos
 - A precisão de ponto flutuantes é suficiente para operações básicas
- “Números” especiais
 - **NaN** - Not a Number
 - **Infinity**

```
console.log(typeof 3) // "number"
console.log(typeof 3.13) // "number"
console.log(typeof 5e9) // "number"
console.log(typeof 5e-5) // "number"
console.log(9999999999999999) // 9999999999999999
console.log(9999999999999999) // 100000000000000000
console.log(0.2 + 0.1) // 0.30000000000000004
console.log((0.2 * 10 + 0.1 * 10) / 10) // 0.3
console.log("x"/2)
console.log(10/0)
```

Visão geral

O tipo number

- Operadores aritméticos:
 - `+, -, *, /, %, **, ++, --, +=, -=, *=, /=, %=, **=`
 - Mesma precedência do Java
 - **Cuidado:** Muitos operadores realização conversão automática de tipos

Visão geral

O tipo BigInt

- Números acrescidos de *n* ao final
- Usado para representar números maiores de 2^{23}
- Operadores:
 - $+$, $-$, $*$, $/$ e $**$
- Ao comparar com um number a conversão é necessária

```
console.log(typeof 1n)
console.log(typeof 9007199254740991n)
console.log(typeof new BigInt(900719925474091))
```

Visão geral

Declarando variáveis

- Variáveis podem ser declaradas usando três palavras chaves
 - `var`, `let` (ECMA6) e `const`
- O tipo da variável não é especificado
- A função `typeof` retorna o tipo de dados de objeto

Visão geral

var vs let vs const

Declarações	Escopo	Pode ser alterada
<code>var</code>	Global (hoisting) / Função	Sim
<code>let</code>	Bloco	Sim
<code>const</code>	Bloco	Não

Visão geral

var vs let vs const

```
console.log(mensagem); // undefined  
var mensagem = "declaracao";  
var mensagem = "redeclaracao";  
console.log(mensagem); // redeclaracao
```

Usando var é possível redeclarar uma variável

```
mensagem2 = 'Msg';  
console.log(mensagem); // Msg  
var mensagem2;
```

Hoisting / Içamento

```
var saudacao = "oi";  
var n = 4;  
if (n > 3) {  
  var saudacao = "ola";  
}  
console.log(saudacao); // "ola"
```

Escopo global
Hoisting + Redeclaração

Visão geral

var vs let vs const

```
function exibirMensagem() {  
  var msgForaDoIf = 'Msg 1';  
  if(true) {  
    var msgDentroDoIf = 'Msg 2';  
    console.log(msgDentroDoIf); // Msg 2  
  }  
  console.log(msgForaDoIf); // Msg 1  
  console.log(msgDentroDoIf); // Ms2  
}
```

```
function exibirMensagem() {  
  var msgForaDoIf = 'Msg 1';  
  if(true) {  
    var msgDentroDoIf = 'Msg 2';  
    let letMsg = 'let';  
    console.log(msgDentroDoIf); // Msg 2  
  }  
  console.log(msgForaDoIf); // Msg 1  
  console.log(letMsg); // Erro  
}
```

Visão geral

null e Undefined

- null
 - Existe, mas foi atribuído com vazio (null)
 - Deliberadamente sem valor
- undefined
 - Variáveis declaradas, mas não inicializadas
 - Membros objeto/Array que não existem

```
typeof null; // "object"  
typeof undefined; // "undefined"
```

Visão geral

O tipo String

```
const string1 = "Primeira string";
const string2 = 'Segunda string';
const string3 = `Mais uma string`;
const string4 = new String("A String como objeto");

const a = 5, b = 10;
console.log(` ${a + b} é igual a 15 não  ${2 * a + b}.`);
//15 é igual a 15 e não 20
```

Visão geral

O tipo String e os seus principais métodos

Método / Propriedade	Descrição
<code>length</code>	Propriedade que contém o tamanho da string
<code>concat()</code>	Concatena um ou mais strings
<code>indexOf()</code>	Retorna a primeira ocorrência de um caractere na string
<code>lastIndexOf()</code>	Retorna a última ocorrência de um caractere na string
<code>match()</code>	Verifica a ocorrência de uma expressão regular na string
<code>replace()</code>	Substitui alguns caracteres na string
<code>slice()</code>	Extrai em uma nova string, parte da string original
<code>split()</code>	Quebra a string em um array de strings
<code>toLowerCase()</code>	Mostra a string em letras minúsculas
<code>toUpperCase()</code>	Mostra a string em letras maiúsculas

Visão geral

Conversão entre String e Number

- Convertendo String em números

```
let count = 10;
let s1 = "" + count; // "10"
let s2 = count + " bananas, ah ah!"; // "10 bananas, ah ah!"
let n1 = parseInt("42 is the answer"); // 42
let n2 = parseFloat("booyah"); // NaN
```

- Acessando caracteres

```
let firstLetter = s[0]; // fails in IE
let firstLetter = s.charAt(0); // does work in IE
let lastLetter = s.charAt(s.length - 1);
```

Visão geral

Comentários

- Idêntico ao do C

```
// comentário de uma linha
/* comentário de
Múltiplas
linhas      */
```

Visão geral

if/else

- Idêntico ao java
- Praticamente qualquer coisa pode ser usada como condição (*booleano*)

```
if (i == j) {  
  if (j == k) {  
    console.log("i igual a k");  
  } else {  
    console.log("i não é igual j");  
  }  
}
```

```
if (n == 1) {  
  // Executa o bloco de código #1  
} else if (n == 2) {  
  // Executa o bloco de código #2 }  
} else if (n == 3) {  
  // Executa o bloco de código #3  
}
```


Visão geral

O tipo booleano

- Qualquer valor pode ser usado como **Boolean**
- Valores considerados **false**:
 - 0, 0.0, NaN, "", null, e undefined
- Valores para verdadeiro
 - Todo o resto
- Ainda é possível converter um valor para **boolean** explicitamente

Visão geral

O tipo booleano

```
let boolValue = Boolean(outroValor); // Convertendo para Boolean
let iLike190M = true;
let ieIsGood = "IE6" > 0; /* false */
if ("web dev is great") { /* true */
  if (0) { /* false */ }
}
```

Visão geral

Operadores relacionais

Operadores	Descrição
>	Maior que
>=	Maior que ou igual a
<	Menor que
<=	Menor que ou igual a
==	Igualdade
!=	Diferente
===	Igualdade sem coerção
!==	Igualdade com coerção

A maioria dos operadores convertem os tipos automaticamente

Visão geral

Operadores relacionais

- `===` e `!==` não realizam a conversão de tipo

```
5 < "7" // true
42 == 42.0 // true
"5.0" == 5 // true
"5.0" === 5 // false
null === undefined; // false
null == undefined; // true
null === null; // true
null == null; // true
!null; // true
```

Visão geral

Operadores lógicos

Operadores	Descrição
&&	E
	Ou
!	Negação

Visão geral

Avaliação Curto Circuito Lógico

- Nem sempre o segundo **operando** é válido ao usar os operadores **&&** e **||**
 - Útil para checagem de objetos antes de acessar seus atributos
 - Atribuição de valor default

```
let name = o && o.getName();  
let name = otherName || "default";
```

Visão geral

Switch/case

- Comparação usa o operador `===`

```
switch (n) {  
  case 1: // Começa aqui se n === 1  
    // Executa o bloco de código #1. break;  
    break; // Para aqui  
  case 2:  
    // Executa o bloco de código #2.  
    break; // Para aqui  
  case 3: // Começa aqui se n === 3 // Executa o bloco de código #3.  
    break; // Para aqui  
  default: // Se tudo falhar... // Executa o bloco de código #4.  
    break; // Para aqui  
}
```

Visão geral

Operador ternário

- Utilizado como alternativa ao **if/else**
- Usem com moderação

```
const result = condition ? trueExpression : falseExpression
const nota = 80
let conceito
if (nota > 70) {
  conceito = "Excelente"
} else {
  conceito = "Bom"
}
console.log(conceito)
```

```
const conceito =
  nota > 70 ? "Excelente" : "Bom"
console.log(conceito)
```


Visão geral

Operador ternário

- O operador ternário é associativo a direita
 - É possível fazer encadeá-lo
 - Alternativa a if/else aninhados
 - Não recomendo

```
const nota = 45

const conceito =
  nota > 70
    ? "Excelente"
    : nota > 50
    ? "Médio"
    : score > 40
    ? "Ruim"
    : "Péssimo"

console.log(conceito)
```

Visão geral

Vetores

- Conjunto **ordenado** de dados
 - Em JavaScript eles não são **tipados**
 - **Dinâmicos**, crescem e diminuem conforme a necessidade
- Cada valor é chamado de *elemento*
- Cada elemento tem uma posição numérica no vetor, conhecida como *índice*

Visão geral

Vetores

- Existem duas maneira de inicializar um vetor

```
let vazio = []; //vetor vazio criado com a forma literal
let preenchido = [1, "bola", ..., true]; //vetor pré preenchido

let novo = new Array(); //vetor vazio criado usando operador new
let cabemDez = new Array(10); // vetor de tamanho 10
let preenchido2 = new Array(1, "bola", ..., true); //vetor pré preenchido

console.log(preenchido[1]); // bola
preenchido[100] = "novo elemento";
```

Visão geral

Vetores e os seus principais métodos

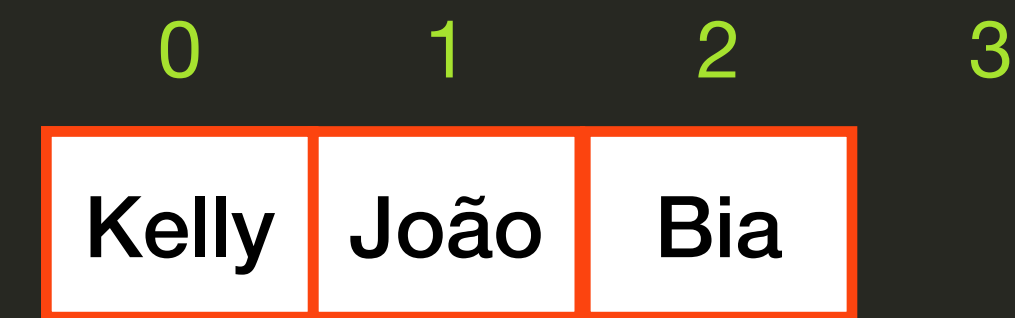
Método / Propriedade	Descrição
<code>length</code>	Propriedade que contém o tamanho de um vetor
<code>concat()</code>	Cria e retorna um novo array contendo os elementos do array original
<code>reverse()</code>	Inverte a ordem dos elementos de um array e retorna o array invertido
<code>slice()</code>	Extrai em um novo vetor, parte do vetor original
<code>sort()</code>	Ordena os elementos de um array e retorna o array ordenado
<code>splice()</code>	Método de uso geral para inserir ou remover elementos de um array
<code>join()</code>	Converte os elementos de um array em strings e as concatena
<code>push()</code>	Anexa um ou mais elementos no final de um array. Retorna o novo tamanho.
<code>pop()</code>	Exclui o último elemento de um array. Retorno o elemento removido.
<code>unshift()</code>	Anexa um ou mais elementos no início de um array. Retorna o novo tamanho.
<code>shift()</code>	Exclui o primeiro elemento de um array. Retorno o elemento removido.

Visão geral

Usando vetores como estrutura de dados

- Pilhas: **push** e **pop**, adicionam e removem respectivamente
- Fila: **unshift** e **shift**, adicionam e e removem respectivamente

```
let a = ["Stef", "João"];  
a.push("Bia");  
a.unshift("Kelly");  
a.pop();  
a.shift();  
a.sort();
```



Visão geral

Vetores

- **split**

- Quebra a string em partes utilizando um delimitador
- Pode ser utilizado com expressões regulares

- **join**

Transforma um vetor em uma string, utilizando um delimitador entre os elementos

```
let s = "Js é bem legal";  
let a = s.split(" ");  
a.reverse();  
s = a.join("!");  
console.log(s); // "legal!bem!é!Js"
```

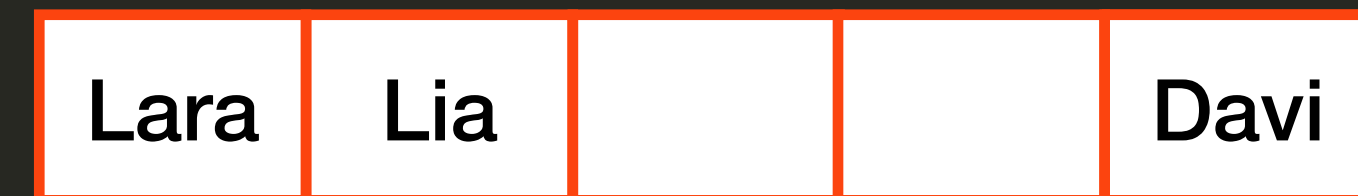
Js	é	bem	legal
----	---	-----	-------

Visão geral

Vetores esparsos

- É um vetor que não possui índices contínuos
 - A propriedade length não representa a quantidade de elementos

```
let nomes = [];  
nomes[0] = "Lara";  
nomes[1] = "Lia";  
nomes[4] = "Caio";  
nomes[4] = "Davi";
```



nomes.length = 5

Visão geral

Estruturas de Repetição

- while
- do / while
- for
 - for/in e for/of (ECMA 2015 ou ES6)

Visão geral

Estruturas de Repetição - while

- A condição é testada antes de iniciar a execução do bloco
- O bloco de código é executado enquanto condição for verdadeira

```
while (condição) {  
    Bloco de código que será executado  
}  
  
let i = 0;  
while (i < 10) {  
    console.log(i + ","); //0,1,2,3,4,5,6,7,8,9  
    i++;  
}
```

Visão geral

Estruturas de Repetição – do/while

- A condição é testada após a execução do bloco
 - O laço é executado **pelo menos uma vez**
- O bloco de código é executado enquanto condição for verdadeira

```
do {  
    Bloco de código pode ser executado  
}  
while (condição);  
  
let i = 1;  
do {  
    if (i % 2 === 0) console.log(i + "é divisível por 2");  
    else console.log(i + " não é divisível por 2");  
    i++;  
} while (i < 1);
```

Visão geral

Estruturas de Repetição – for

- Instrução 1 - Executada antes de iniciar o bloco
- Instrução 2 - Executadas antes de cada iteração do laço
- Instrução 3 - Executadas após a iteração do laço

```
for (instrução 1; instrução 2; instrução 3) {  
  code block to be executed  
}
```

```
for (let i = 0; i < 10; i++) {  
  console.log(`${i}, "); //0,1,2,3,4,5,6,7,8,9  
}
```

Visão geral

Estruturas de Repetição – for/in

- Utilizada para percorrer propriedades enumeráveis
 - Ex: vetores, strings, objetos

```
const v = [1, 2, 3, 4, 5]
for (let i in v) {
  console.log(`v[${i}] = ${v[i]}`)
}
```

Visão geral

Usos comuns

- Filtrar elementos

```
const todasCompras = [10, 33, 120, 27, 50, 200, 500];  
const compraCaras = [];  
for (let i in todasCompras) {  
  if (todasCompras[i] >= 100) {  
    compraCaras.push(todasCompras[i]);  
  }  
}
```

120	200	500
-----	-----	-----

Visão geral

Usos comuns

- Transformar o vetor

```
const todasCompras = [120, 50, 200, 500];  
const DESCONTO = 0.10; // 10%  
for (let i in todasCompras) {  
  todasCompras[i] -= todasCompras[i] * DESCONTO  
}
```

108

45

150

450

Visão geral

Usos comuns

- Processar e resumir o vetor em um resultado

```
const todasCompras = [100, 50, 200, 500];  
let total = 0;  
for (let i in todasCompras) {  
  total += todasCompras[i]  
}  
console.log(total); // 850
```

Visão geral

Alterando o fluxo normal de um laço

- É possível alterar o fluxo por meio de algumas instruções
 - *break*
 - *continue*
 - *return*
 - *throw*

Visão geral

Alterando o fluxo normal de um laço

- *break*
 - Faz com que o laço ou switch mais interno seja abando

```
const v = [1, 4, 2, 4, ...] //Suponha que esse vetor seja muito grande
const buscandoPor = 4
let achou = false
for (let i in v) {
  achou = v[i] === buscandoPor;
  if(achou) {
    break;
  }
}
```

Visão geral

Alterando o fluxo normal de um laço

- *return*
 - Só pode aparecer dentro do corpo de uma função
 - Retorna o valor da expressão como resultado da execução da função

```
function qualquer() {  
  const v = [1, 4, 2, 4, 3] //Suponha que esse vetor seja muito grande  
  const buscandoPor = 4  
  for (let i in v) {  
    if(v[i] === buscandoPor) {  
      return true;  
    }  
  }  
  return false;  
}
```

Visão geral

Alterando o fluxo normal de um laço

- *continue*
 - Quando a instrução `continue` é executada, a iteração atual do laço circundante é terminada e a próxima iteração começa

```
function processaVetor() {  
  const v = [1, 4, 2, 4, 3]  
  for (let i in v) {  
    if (v[i] === condicao) continue;  
    processamentoPesado(v[i])  
  }  
}
```

Visão geral

Exceções

- São um sinal indicando que ocorreu algum tipo de condição excepcional ou erro
 - São lançadas quando ocorre um erro em tempo de execução
 - O programa pode disparar uma explicitamente usando a instrução *throw*
- Capturar uma exceção
 - É tratar-lá, ou seja, executar as ações necessárias para se recuperar da exceção
- As exceções são capturadas com as instruções *try/catch/finally*

Visão geral

Exceções

```
try {  
    }  
catch (err) {  
    }  
finally {  
    }
```

- Uso obrigatório.
- Delimita o bloco que pode gerar a exceção

- É executado caso a exceção ocorra
- Interrompe a propagação do erro
- Tem acesso a exceção lançada

- Opcional
- O seu bloco de código é **SEMPRE** executado
 - Independentemente da exceção ser lançada
 - Mesmo que o bloco *try* possua um *return*

Visão geral

Alterando o fluxo normal de um laço

- Exceções não tratadas alteram o fluxo de execução de um programa
 - Interrompe a execução de uma função imediatamente
 - Continua a ser propagada até que ela seja tratada ou que o programa encerre com erro

```
function processaVetor() {  
  const v = [1, 4, 2, 4, 3]  
  for (let i in v) {  
    if (v[i] === condicao) throw "Valor inválido";  
    processamentoPesado(v[i])  
  }  
}
```

Objetos



Function == Object

Objetos

Introdução

- É o tipo fundamental de dados em JavaScript é o objeto
- É um conjunto não ordenado de propriedades
- Mapeiam propriedades (strings) em valores
 - Esse mapeamento recebe vários nomes: “hash”, “tabela de hash”, “dicionário” ou “array associativo”
- São manipulados por referência e não por valor.
- Em JavaScript são dinâmicos
 - Propriedades podem ser **adicionadas** e **excluídas**

Objetos

Criando objetos

- A maneira mais fácil de criar um objeto é incluir um **objeto literal**
 - É uma lista separada com vírgulas de pares **nome:valor** separados por **dois-pontos**, colocados **entre chaves**
- O nome de uma propriedade é um identificador JavaScript ou uma string literal
- O valor de uma propriedade é qualquer expressão JavaScript
 - Pode ser um valor primitivo ou um valor de objeto

Objetos

Sintaxe de Objetos literais

```
let vazio = { };
let ponto = { x: 0, y: 0 }

let pikachu = {
  nome: "Pikachu",
  especie: "Pikachu",
  nivel: 1,
};

let livro = {
  titulo: "JavaScript",
  'sub-titulo': "O Guia Definitivo",
  autor: {
    nome: "David",
    sobrenome: "Flanagan"
  }
};
```

Objetos

Criação de Objetos

- É possível criar objetos usando o operador **new**
- Após a criação é possível adicionar métodos e atributos

```
let charmander = new Object();  
charmander.nome = "Charmander";  
charmander.especie = "Charmander";  
charmander.nivel = 5;
```

Objetos

Acessando propriedades

- É possível acessar os propriedade usando duas notações
 - Dot notation, por meio do operador .
 - Bracket notation, por meio do operador `[]` (vetor associativo)

```
let pikachu = {  
  nome: "Pikachu",  
  ...  
}  
let charmander = new Object();  
charmander.nome = "Charmander";  
  
console.log(pikachu.nome);  
console.log(charmander['nome']);
```

Objetos

Acessando atributos

- Bracket notation
 - Permite acessar propriedade cujo nome são calculados em execução
 - Não é preciso conhecer o nome previamente

```
let carteira = {  
  btc: 0.009,  
  eth: 0.195,  
  bnb: 0.18,  
  ...  
  ...  
}  
  
for (let cripto in carteira) {  
  console.log(`${cripto.toUpperCase()} = ${carteira[cripto]}`)  
}
```

Objetos

Verificando a existência de um propriedade

- Garante a existência de uma propriedade
- Lembrete: em JavaScript é possível remover e adicionar propriedades

```
let livro = {
  titulo: "JavaScript",
  autor: {
    nome: "David",
    sobrenome: "Flanagan"
  }
};

console.log(livro.subtitulo) // undefined
console.log(livro.subtitulo.length) // TypeError

let tamanho = undefined
if (livro) {
  if (livro.titulo) {
    tamanho = livro.titulo.length
  }
}
tamanho = livro &&
  livro.titulo &&
  livro.titulo.length
```

Objetos

Verificando a existência de um propriedade

```
let livro = {
  titulo: "JavaScript",
  subtítulo: undefined,
  autor: {
    nome: "David",
    sobrenome: "Flanagan"
  }
};

console.log("titulo" in livro)
console.log(livro.titulo !== undefined)
console.log("subtítulo" in livro)
console.log(livro.subtítulo !== undefined)
```

Objetos

Percorrendo um objeto

```
let carteira = {  
  btc: 0.009,  
  eth: 0.195,  
  bnb: 0.18,  
  ...  
}
```

```
for ( let attr in Object.keys(carteira) ) {  
  console.log(`${attr} = ${carteira[attr]}`);  
}
```

```
for ( let [attr, value] of Object.entries(carteira) ) {  
  console.log(`${attr} = ${value}`);  
}
```

```
for ( let value of Object.values(carteira) ) {  
  console.log(`${value}`);  
}
```

Adicionados no ECMA 6



Objetos

Funções construtoras

```
function Pokemon(nome, especie, nivel=1) {  
  this.nome = nome;  
  this.especie = especie;  
  this.nivel = nivel;  
  this.falar = () => `${this.nome} ${this.nome}`;  
}
```

```
let pikachu = new Pokemon("Pikachu", "Pikachu");  
let charmander = new Pokemon("Charmander", "Charmander", 10)
```

Objetos

Classes - ECMA 6

- Nova sintaxe adicionada no ECMA 6
- JavaScript continua sendo baseada em prototype
- Facilita a implementação de herança
- Syntatic Sugar -> Parecido com outra linguagens

Objetos

Classes - ECMA 6

```
class Pokemon {
  constructor(nome, especie, nivel=1) {
    this.nome = nome;
    this.especie = especie;
    this._nivel = nivel;
  }
  falar = () => `${this.nome} ${this.nome}` ;

  get nivel() { return this._nivel }
  set nivel(valor) { this._nivel = valor > 0 ? valor : 1 }
}
let pikachu = new Pokemon("Pikachu", "Pikachu", -1);
console.log(`${pikachu.falar()} ${pikachu.nivel}`); // pikachu pikachu 1
```

Funções



Function == Object

Funções em JavaScript

- Em JavaScript funções são **objetos**, logo possuem **propriedades e métodos**
 - **Métodos:** apply() e call()
 - **Propriedades:** length e constructor
- Funções são **first-class citizen**. Podem ser:
 - Usada como um valor qualquer
 - Armazenadas em vetores, variáveis e objetos
 - Passadas como argumentos para outras funções
 - Retornadas por outras funções

Funções em JavaScript

Exemplo

```
function exemplo(a, b) {  
  return a * b;  
}
```

Declaração

```
exemplo.length // 2  
exemplo.constructor // Function()
```

Notação literal - Function expression

```
const s = function square(number) { return number * number };
```

```
let x = square(4) // x -> 16
```

```
let y = s(4) // x -> 16
```

Notação literal c/ função anônima

```
const double = function(number) { return 2 * number };
```

```
x = double(4) // x -> 8
```

Funções em JavaScript

Passagem de parâmetros

- É possível passar **qualquer quantidade de parâmetro para qualquer função**
 - Não resulta em erro
- Os parâmetros são armazenados em uma estrutura similar a um vetor chamada de **arguments**
- A propriedade **length** de uma função **armazena a quantidade de parâmetros da função**

Funções em JavaScript

O objeto arguments

```
let x = soma(1, 123, 500, 115, 44, 88);
```

```
function soma() {  
  let i, soma = 0;  
  for (i = 0; i < arguments.length; i++) {  
    soma += arguments[i];  
  }  
  return soma;  
}
```

Permite acessar todos os argumentos

Funções em JavaScript

Rest parameter - ECMA 6

```
let x = soma(1, 123, 500, 115, 44, 88);
```

```
function soma(...args) {
```

```
  let soma = 0;
```

```
  for (let arg of args) soma += arg;
```

```
  return soma;
```

```
}
```

Armazenas todos os argumentos em um vetor

```
let x = soma(4, 9, 16, 25, 29, 100, 66, 77);
```

Funções em JavaScript

Valor default - ECMA 6

```
function soma (x, y = 10) {  
  return x + y;  
}
```

```
soma (5) ; //15
```

Valor atribuído ao argumento caso nenhum valor seja passado

Funções em JavaScript

Callback

```
const cores = ['Marrom', 'Castanho', 'Vermelho', 'Roxo']
```

```
function contarCaracteres (palavra) {  
  return palavra.length  
}
```

Função enviada como argumento



```
console.log(cores.map(contarCaracteres)) // [6, 8, 8, 4]
```

```
console.log(cores.map(function (palavra) {  
  return palavra.length  
})) // [6, 8, 8, 4]
```

Arrow function com um único argumento



```
console.log(cores.map(palavra => palavra.length)) // [6, 8, 8, 4]
```

Funções em JavaScript

Arrow functions - ECMA 6

```
const numeros = [1, 8, 4, 9, -1]
console.log(numeros.filter( n => n % 2 == 0)) // [8, 4]
```

```
const maximo = (a, b) => {
  return Math.max(a, b)
}
```

Arrow function com dois argumentos

```
console.log(numeros.reduce(maximo)) // 9
```

```
console.log(numeros.filter( n => n % 2 == 0).reduce((a, b) => Math.max(a, b)))
// ]
```

Arrow function com uma única instrução

Funções em JavaScript

Funções Internas

```
function raizes(a, b, c) {  
  function delta() {  
    return b**2 - 4*a*c  
  }  
  function raiz1() {  
    return (-b + Math.sqrt(delta())) / (2*a)  
  }  
  function raiz2() {  
    return (-b - Math.sqrt(delta())) / (2*a)  
  }  
  return [raiz1(), raiz2()]  
}
```

- Funções internas tem acesso as variáveis do seu escopo e do seus parentes

Referências

- <https://blog.betrybe.com/javascript/>
- <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>
- <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript>
- <https://pt.wikipedia.org/wiki/JavaScript>
- <https://javascriptrefined.io/nan-and-typeof-36cd6e2a4e43>
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt
- <https://www.toptal.com/javascript/es6-class-chaos-keeps-js-developer-up>

Por hoje é só