SEMINÁRIO JAVASCRIPT

Elciney Júnior
Fernando Teodoro
Mateus Antonio
Felippe Mozer
Luan Correa

INTRODUÇÃO

Introdução

- A Netscape foi fundada em 1994 para explorar a Web que estava surgindo. Foi então criado o Netscape Navigator;
- A Netscape chegou à conclusão que a web teria que se tornar mais dinâmica;
- JavaScript desenvolvido por Brendan Eich em 1995 (apenas 10 dias);
- Primeiramente nomeada de Mocha, depois de LiveScript, para só então se tornar JavaScript;

Introdução

- Microsoft criou, em Agosto de 1996, uma linguagem idêntica para ser usada no Internet Explorer 3: JScript;
- Netscape decidiu normatizar a linguagem através da organização ECMA International, companhia que era especializada em padrões e normativas;
- O nome JavaScript já era patenteado pela Sun Microsystems (hoje Oracle) e não poderia ser usado. Portanto, o nome composto por ECMA e JavaScript foi usado, resultando em ECMAScript;
- ECMAScript é apenas usado para se referir as versões da linguagem;
- JavaScript normatizada no padrão ECMA-262 e ISO/IEC 16262;

Por que JavaScript?

- **Multi Plataforma:** A linguagem não se limita mais em rodar apenas no Browser, mas, hoje roda como aplicativos desktop, como aplicações mobile Híbridas, e claro, no servidor com Node.js;

- **Ganhando Ainda Mais Popularidade:**

gráfico que mostra uma crescente na interação da comunidade através de perguntas no site Stack Overflow

Por que JavaScript?

- **Muitos Recursos Disponíveis:** Com o aumento de profissionais que trabalham com a linguagem o que não faltam são pacotes com funcionalidades prontas para ajudar a criar aplicações de forma mais produtiva;
- Node.js conta com uma das maiores comunidades ativas;
- NPM(Node Package Manager);
- Desenvolvimento JavaScript Full-Stack;

CARACTERÍSTICAS BÁSICAS

Características Básicas

- Multi-paradigma;
- Case sensitive;
- Possui mecanismos de tratamento de exceções;
- Suporta operações bit a bit;
- Interpretada;
- Funções como cidadãs de primeira classe;
- Memória gerenciada pela linguagem;

Declaração e Definição

- JavaScript não restringe o momento de declaração de variáveis;
- Variáveis podem ser declaradas pelas palavras chave:
  - var
  - let
  - const
- Variáveis declaradas como constantes não podem ser alteradas;

```
const a = 10

a = a * 3 //Uncaught TypeError: Assignment to constant variable
```

Declaração e Definição

- Entretanto, atributos de objetos não estão protegidos;

```
// const também funciona com objetos
const myObject = {'key':'value'};

// Sobrescrever o objeto falha
myObject = {"otherKey": "value"};

// a seguinte instrução é executada sem problemas
myObject.key = "otherValue"; //Utilize Object.freeze() se quiser tornar um objeto imutável
```

Escopo

- Diferenças no escopo da variável feito pela palavra chave;

```
if(1){
    let it = "go"
    var i = "abel"

    console.log(it)
    console.log(i)
}

console.log(i)
console.log(it) //Uncaught ReferenceError: it is not defined
```

Elevação

- Hoisting;

```
var i = "abel" //variavel global

var func = function() {
   console.log(i)

   var i  = 10    //variavel local

   console.log(i)
}

console.log(i)
func()
```

Qual o resultado esperado?

Elevação

- Hoisting;

```
var i = "abel" //variavel global

var func = function() {
   console.log(i)

   var i  = 10    //variavel local

   console.log(i)
}

console.log(i)
func()
```

abel
undefined
10

VALORES E TIPOS

Valores e Tipos de Dados

Valores padrão:

- NULL
- Undefined
- NaN (Not-a-Number)

exemplo.js

```
1   var exemplo;
2   console.log(exemplo);
3   // undefined
4
5   var foo = null;
6   console.log(foo);
7   // null
8
9   console.log(Math.sqrt(-1));
10  // NaN
```

Valores e Tipos de Dados

- Tipagem fraca
- Tipagem dinâmica

ex1.html | exemplo.js

```
1  console.log('1' + 1);
2  // 11
3
4  console.log('1' + true);
5  // 1true
6
7  console.log(1 + true);
8  // 2
```

exemplo.js

```
1  var ex = 10;
2  console.log(ex);
3  //10
4
5  ex = "JavaScript";
6  console.log(ex);
7  //JavaScript
```

Valores e Tipos de Dados

Tipos primitivos:

- String (UTF-8)
- Number (double-precision 64-bit floating point format (IEEE 754))
- Boolean
- Null
- undefined
- Symbol
  - A única utilização sensata para para essa construção é armazená-la em uma variável que será utilizada como chave para uma propriedade de objeto cujo objetivo é torná-lo anônimo (https://developer.mozilla.org/pt-BR/docs/Glossario/Symbol)

JS exemplo.js

```
1   var ex = 10;
2   console.log(typeof(ex));
3   // number
4
5   ex = "JavaScript";
6   console.log(typeof(ex));
7   // string
8
9   ex = null;
10  console.log(typeof(ex));
11  // object (bug no ECMAScript, deveria ser null)
12
13  ex = undefined;
14  console.log(typeof(ex));
15  // undefined
16
17  ex = Symbol();
18  console.log(typeof(ex));
19  // symbol
```

Valores e Tipos de Dados

- Tipo enumerado:
  - Declarado como objeto

exemplo.js

```
var SizeEnum = {
    SMALL: 1,
    MEDIUM: 2,
    LARGE: 3,
    properties: {
      1: {name: "small", value: 1, code: "S"},
      2: {name: "medium", value: 2, code: "M"},
      3: {name: "large", value: 3, code: "L"}
    }
  };

console.log(SizeEnum.properties[SizeEnum.MEDIUM].code);
// M
```

Valores e Tipos de Dados

- Classes
- Objetos
- Vetores

exemplo.js

```js
var pessoa = { nome: "Felippe", idade: 18 };
console.log(pessoa.nome);
// Felippe

class Sapato {
    constructor(marca, modelo, tamanho) {
        this.marca = marca;
        this.modelo = modelo;
        this.tamanho = tamanho;
    }
}

var rafarillo = new Sapato("Rafarillo", "Modelo X", 42);
console.log(rafarillo.tamanho);
// 42
```

exemplo.js

```js
var pessoas = ["Felippe", "Elciney", "Luan", "Fernando", "Mateus"];

pessoas.forEach(function(item) {
    console.log(item);
});

// Felippe
// Elciney
// Luan
// Fernando
// Mateus
```

Valores e Tipos de Dados

- Não possui uniões
- Vetores multidimensionais não existem nativamente (é possível criar arrays dentro de arrays)
- Possui funções

exemplo.js

```
1   var array1 = new Array(4);
2
3   for(var i = 0; i < 4; i++) {
4       array1[i] = new Array(4);
5
6       for(var j = 0; j < 4; j++) {
7           array1[i][j] = i+j;
8           console.log(array1[i][j]);
9       }
10  }
11
12  // 0 1 2 3 1 2 3 4 2 3 4 5 3 4 5 6
```

exemplo.js

```
1   function quadrado(numero) {
2       return numero * numero;
3   }
4
5   console.log(quadrado(5));
6   // 25
```

Valores e Tipos de Dados

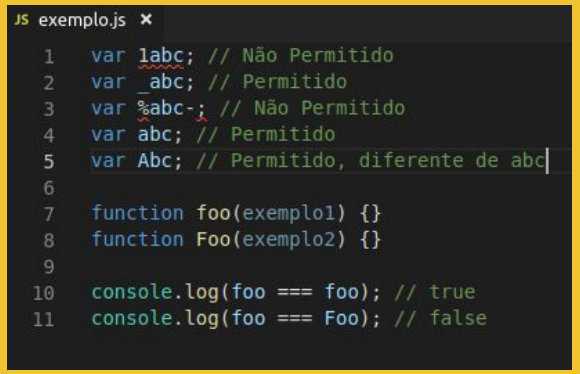
- Funções podem ser tratadas como cidadãs de primeira classe

exemplo.js

```
1   var myVar = function myFunc() {};
2   console.log(myVar);
3   // f myFunc() {}
4
5   function myFunc1() {};
6   function myFunc2 (myFunc1) {  console.log(myFunc1); };
7   myFunc2(myFunc1);
8   //  f myFunc1() {}
9
10  function myFunc4() { return function myFunc5() {} };
11  console.log(myFunc4());
12  // f myFunc5() {}
13
14  var obj = { myFunc6 : function myFunc6() {} };
15  console.log(obj.myFunc6);
16  //  f myFunc6() {}
```

Variáveis e constantes

- Restrições quanto a identificadores de variáveis e funções:
  - Não é possível começar com números;
  - Versões antigas do JS não aceitam o uso de acentos e cedilhas (evitar por questão de portabilidade);
  - Não é permitido o uso de símbolos especiais, como ‘&’ e ‘-’, por exemplo;
  - Não há limite para tamanho dos identificadores;
- JS é Case Sensitive para nomenclatura de funções e variáveis



Palavras reservadas em JS

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Variáveis e Constantes

- Em JS, tudo é objeto

- JS possui coletor de lixo (*Garbage Collector*)
  - Utiliza a estratégia de coleta *Mark and Sweep* (Marcar-Varrer);

EXPRESSÕES E OPERADORES

Operadores:

- O JavaScript possui tanto operadores **binários** quanto **unários** e um operador **ternário (condicional)**.

- Um operador **binário** exige dois operandos, um antes do operador e outro depois:

  Por exemplo, 3+4 ou x*y.

- Um operador **unário** exige um único operando, seja antes ou depois do operador:

  Por exemplo, x++ ou ++x.

Operadores de atribuição:

- Um operador de atribuição atribui um valor ao operando à sua esquerda baseado no valor do operando à direita. O operador de atribuição básico é o igual (=), mas existem vários outros:

(+=) (-=) (*=) (/=) (%=) (=)**

Operadores de comparação:

- Um operador de comparação compara seus operandos e retorna um valor lógico baseado em se a comparação é verdadeira. Os operandos podem ser numéricos, strings, lógicos ou objetos. São eles:

(==) (!=) (>) (>=) (<) (<=) (===) (!==)

*Obs: **(=>)** não é um operador, mas a notação para Arrow Function*

Operadores aritméticos:

- Operadores aritméticos tomam valores numéricos (sejam literais ou variáveis) como seus operandos e retornam um único valor numérico.

Modulo (%), **Incremento (++)**, **Decremento (--)**
Negação (-), **Adição (+)** e **Exponencial (**)**

Operadores bit a bit:

- Operadores bit a bit tratam seus operandos como um conjunto de 32 bits (zeros e uns). Operadores bit a bit realizam suas operações nestas representações, mas retornam valores numéricos padrões do JavaScript. Eles se dividem em operadores lógicos (AND, OR, XOR e NOT) e operadores de deslocamento (<<, >> e >>>)

Operadores lógicos:

- Operadores lógicos são utilizados tipicamente com valores booleanos (lógicos); neste caso, retornam um valor booleano.

AND (&&), OR (||) e **NOT (!)**

Operadores de string:

- Além dos operadores de comparação, que podem ser utilizados em valores string, o operador de concatenação (+) concatena dois valores string, retornando outra string que é a união dos dois operandos.

Operador condicional (ternário):

- O operador condicional é o único operador JavaScript que utiliza três operandos.

Exemplo: *var status = (idade >= 18) ? "adulto" : "menor de idade";*

Operadores unarios:

- **delete**

O operador **delete** apaga um objeto, uma propriedade de um objeto ou um elemento no índice especificado de uma matriz.

delete nomeObjeto;
delete nomeObjeto.propriedade;

- **typeof**

O operador **typeof** retorna uma string indicando o tipo do operando sem avaliação. Operando é uma string, variável, palavra-chave ou objeto cujo tipo deve ser retornado.

```
typeof meuLazer;  // retorna "function"
typeof forma;     // retorna "string"
typeof tamanho;   // retorna "number"
```

Operadores relacionais:

- **in**

O operador **in** retorna verdadeiro se a propriedade especificada estiver no objeto especificado.

```
nomePropriedadeOuNumero in nomeObjeto
```

- **instanceof**

O operador **instanceof** retorna verdadeiro se o objeto especificado for do tipo de objeto especificado.

nomeObjeto instanceof tipoObjeto

Expressões:

Existem dois tipos de expressões: aquelas que atribuem um valor a uma variável e aquelas que simplesmente possuem um valor.

- **this**

Utilize a palavra reservada **this** para se referir ao objeto atual.

- **new**

Você pode utilizar o operador **new** para criar uma instância de um tipo de objeto definido pelo usuário ou de um dos tipos de objeto predefinidos.

- **super**

A palavra reservada **super** é utilizada para chamar a função pai de um objeto. É útil para nas classes para a chamada do construtor.

MODULARIZAÇÃO

- Encapsulamento é um dos fundamentos da programação orientada a objetos tradicional. Se entendermos encapsulamento como uma forma de restringir acesso à informação, concluímos que a definição de escopo é o caminho para alcançá-lo.

Módulos:

- Existem alguns módulos que podem ser seguidos em JavaScript

- **Revealing Module Pattern:**

Neste padrão, todas as funções e valores do módulo são acessíveis no escopo local e apenas referências são retornadas na forma de objeto.

```
var counter = (function () {
 var current = 0;
 function next() {
   return current + 1;
 }
 function isFirst() {
   return current == 0;
 }

 return {
   next: next,
   isFirst: isFirst
 };
})();
```

- **Namespace:**

Os padrões que vimos até então poluem o escopo global da aplicação com a definição de uma série de variáveis. Uma solução é a criação de um *namespace* de uso específico para os módulos.

```
window.App = {
   modules: {}
 };
  App.modules.counter = (function () {
   /* ... /
 })();
  App.modules.slider = (function () {
   / ... */
 })();
```

- **Asynchronous Module Definition (AMD):**

Módulos AMD podem ser requisitados, definidos e utilizados a medida que necessários. Nosso contador, se reescrito em AMD, ficaria da seguinte maneira:

```
define('counter', function () {
   var current = 0;
   function next() {
     return current + 1;
   }
   function isFirst() {
     return current == 0;
   }
   return {
     next: next,
     isFirst: isFirst
   };
 });
```

POLIMORFISMO E EXCEÇÕES

Sistemas de Tipos:

- Oferece inferência de tipos:

```
var arbitro = 1 // linguagem infere tipo number
var arbitro = "arbitro de video" // linguagem infere tipo string
```

- Isenta de especificações de tipos

- Tipagem fraca e dinâmica

Coerção:

- JS realiza **conversão implícita**

- Como numeros sao tipo Number (implicitamente float) , conversões ocorrem entre outros tipos

Ordem de coerção para o operador soma ‘+’ :

- Numeros e strings -> converte para string
- Booleanos e strings -> converte para string
- Numeros e booleanos -> converte para numeros

Coerção:

- Exemplos

String > Number > Boolean

- Ampliação

```
String and number

var a = 5;
var b = "Word";

a = a + b; // Saida: 5Word

String and boolean

var a = false;
var b = "Word";

a = a + b; // Saida: falseWord

Number and boolean

var a = 5;
var b = true;

a = a + b; // Saida: 51
```

Coerção:

- Conversão implícita de strings para números em operações numéricas

```
var x = "100";
var y = "10";
var z = x / y;   // 10

var x = "100";
var y = "10";
var z = x * y;   // 1000

var x = "100";
var y = "10";
var z = x - y;   // 90
```

Exceto: operador ‘+’

```
var x = "100";
var y = "10";
var z = x + y;   // 10010
```

Coerção:

Funções principais usadas para conversão implícita:

- Number();
- String();
- Boolean();
- Object();

Sobrecarga:

- Permite sobrecarga de operadores

- Permite sobrecarga de métodos
    - Dependente de contexto
    - Independente de contexto

Sobrecarga:

- Sobrecarga de operadores

```
var x = 10;
var y = 20;
var z = x + y;   // 30
```

```
var x = "Ola";
var y = " Mundo";
var z = x + y = "Ola Mundo";
```

Sobrecarga:

- Independente de contexto

```
function set(arg1, arg2) {
    var a = arg1;
    if(typeof arg2 !== "undefined") {
        x+= arg2;
    }
    return a;
};

set("Hello"); // retorno : Hello
set("Hello ", 10); // retorno : Hello10
```

Sobrecarga:

- Dependente de contexto

```
var Cliente =
{    Class:function()
     {
          this.tipo = function() {
               return this.tipo[args[0].constructor].apply(this, args)
          }
          this.tipo[String] = function() {
               alert("tipo String");
          }
          this.tipo[Number] = function() {
               alert("tipo Number");
          }
     }
}
```

Sobrecarga:

- Dependente de contexto

```
Var Cli = new Cliente.Class;
  Cli.tipo("Hello");  // alert tipo String
  Cli.tipo(50);       // alert tipo Number
```

Paramétrico:

- Em JavaScript, o polimorfismo paramétrico se expressa através da declaração de tipos.
- Identificador “var” declara desde primitivos até objetos
- Uma única variável pode abrigar diferentes tipos

Paramétrico:

- Exemplos:

```
var cliente = {
    nome: "Joilson",
    idade: 46,
};

var i = 4.76;
var time = "Vasco";
var lista = [i, time, cliente];
```

Inclusão e Herança:

- Poliformismo clássico da Orientação a Objetos
- Relacionamento de hierarquias entre subtipos e supertipos
- Conceito de heranças aparece
- Em javascript: **Modelo de protótipos**.

Inclusão e herança:

```
var Empresa = function(nome, endereco) {
    this.nome = nome;
    this.endereco = endereco;
};
Empresa.prototype.MeuEndereco = function() {
    console.log("Meu endereco e +" + this.endereco);
};
var EmpresaRegistrada = function(nome, endereco, cnpj) {
    Empresa.call(this, nome, endereco);
    this.cnpj = cpnj;
}
```

Inclusão e herança :

- A fim de se herdar métodos de empresa, necessita-se fazer o protótipo de EmpresaRegistrada ser uma instância de Empresa

```
EmpresaRegistrada.prototype = new Empresa();
```

- Métodos de EmpresaRegistrada devem ser declarados após sobrescrever o protótipo

Inclusão e Herança:

- Exemplo de função de ECMAScript

```
if(typeof Object.create !== 'function') {
    Object.create = function(o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

Inclusão e Herança:

- Permite sobrescrita de métodos

```
function adiciona(produto) {
    adiciona(produto, 1);
}
function adiciona(produto, categoria) {
    // essa funcao "sobrescreve" a anterior
}
```

- Possui amarração tardia
- Não possui herança múltipla

Classes

- Palavra-chave class e introduzida ao JavaScript em 2015

```
class Retangulo {
    constructor(altura, largura) {
        this.altura = altura;
        this.largura = largura;
    }
}
```

- Caráter sintático, continua fazendo uso de protótipos

Tipos de exceções:

- Basicamente existem três tipos de exceções em Javascript:
  - Throw
  - Try...Catch
  - Finally

- E basicamente, três tipos de erros:
  - Sintaxe
  - Tempo de execução
  - Lógicos

- Em geral, exceções tratam de **erros lógicos**.

Throw:

- Uso de exceções personalizadas
- Permite lançamento de uma expressão de vários tipos diferentes

```
throw "Error2";   // tipo string
throw 404;         // tipo numérico
throw true;       // tipo booleano
throw {toString: function() { return "Objeto na area!"; } };
```

Try...Catch:

- Tipo de exceção onde há uma tentativa de execução, mas caso haja algum erro na execução, existe um mecanismo pronto para “pegar” a expressão que trata este erro e assim tratá-lo
- Podem existir vários blocos catch para diferentes erros

```
try {
    if ( !usuario.permissao ) {
        throw new BusinessError("Acesso não autorizado");
    }
} catch (err) {

    alert(err.message);
}
```

Try...Catch

- Bloco catch nativamente dispõe de um “CatchID” para obtenção de informações a respeito da exceção
- Disponível enquanto o bloco está em execução

```
try {
    throw "myException";
}
catch (e) {
    // declarações
    logMyErrors(e); // passar a exceção para o manipulador de erro
}
```

Finally

- Bloco que deve ser executado após os blocos try...catch, porém antes da execução posterior aos mesmos
- Bloco que sempre será executado ao detectar-se uma exceção

```
openMyFile();
try {
    writeMyFile(theData);
} catch(e) {
    handleError(e);
} finally {
    closeMyFile();
}
```

Manipulador de eventos Onerror:

- Objetivo de facilitar o tratamento de erros em páginas web
- Oferece parâmetros para determinar com precisão a área do erro

```
<script>
    window.onerror = function () {
        alert("An error occurred.");
    }
</script>
```

CONCORRÊNCIA

Motivação:

- Navegadores (exceto Chrome) executam todas as abas como um único processo
- Tarefas demoradas podem tornar péssima a experiência do usuário
- Considerando o event-loop, não parece bom fazer o usuário esperar que a página encerre algum processamento para liberar novamente a interação com a página

Event-loop:

- **“Run-To-Completion”** processa completamente uma mensagem antes de iniciar outra
- Diferentemente de C que pode parar a execução de uma thread e executar outra
- I/O realizado por eventos e callbacks
- Não-bloqueante

Pontos negativos:

- Se uma tarefa (Não I/O) toma muito tempo, interação fica indisponível (clique, rolagem, digitação)
- Navegador geralmente mitiga com a mensagem “um script está tomando muito tempo para executar

Problema:

- Parece normal
- Retorna name antes de name receber o valor esperado

```
// Every day on StackOverflow
function getUserName() {
    let name;
    $.get('/users/123', (user) => {
        name = user.name;
    });
    return name;
}
// Why doesn't it work?
console.log('User Name:',
getUserName());
```

Concorrência

- Callbacks são muito comuns em JS
- Callback Hell

```
a(function(resultsFromA) {
    b(resultsFromA, function(resultsFromB) {
        c(resultsFromB, function(resultsFromC) {
            d(resultsFromC, function(resultsFromD) {
                e(resultsFromD, function(resultsFromE) {
                    f(resultsFromE, function(resultsFromF) {
                        console.log(resultsFromF);
                    }
                }
            }
        }
    }
});
```

Promises:

- Promises viabiliza o encadeamento necessário
- Em .then() nós podemos inserir o tratador (ou callback) que será executado quando o dado estiver pronto.
- Bem mais claro comparado à callbacks

```
// Promise style
readFile('config.json')
    .then(…)
    .catch(…);
```

Promises:

```
// This would be very difficult with callbacks
fetchJSON('/user-profile')
    .then((user) => {
        return fetchJSON(`/users/${user.id}/friends`);
    })
    .then((friendIDs) => {
        let promises = friendIDs.map((id) => {
            return fetchJSON(`users/${id};`);
        });
        return Promise.all(promises);
    })
    .then((friends) => console.log(friends));
```

Concorrência

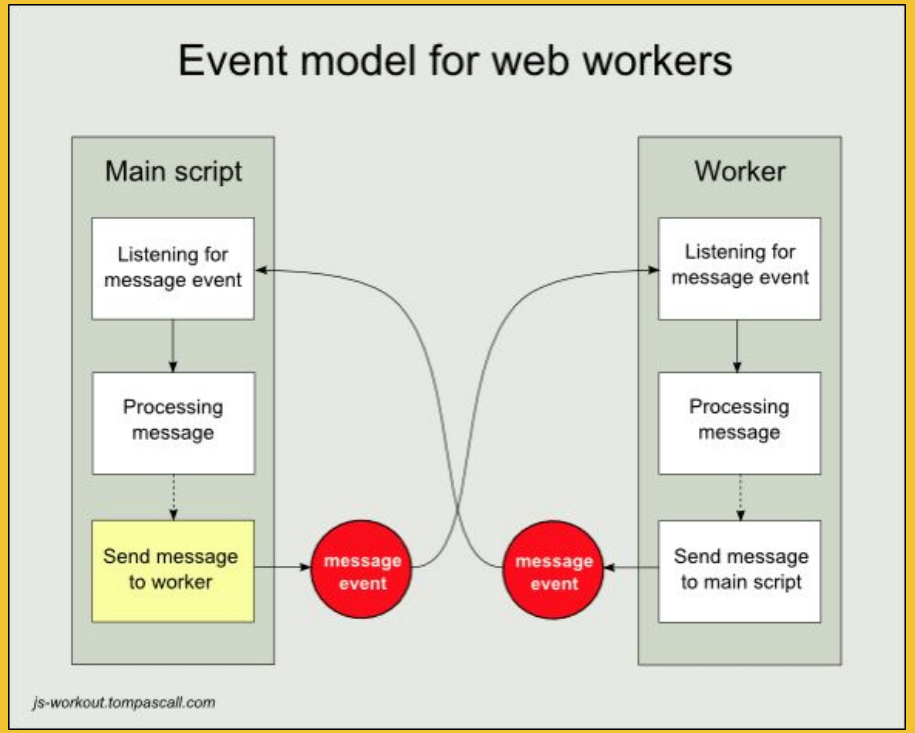
- Async/Await
- Feature mais importante do ES6
- Funciona como uma camada sobre promises
- Pausa execução da função
- Event loop continua trabalhando
- Retoma execução quando promessa for cumprida
- Continua fluxo normal

```
async function getUsers() {
   // Here's the magic
   let result = await
fetchJSON('/users');
   console.log(result);}
```

WEB-Worker:

- Ferramenta para fazer JS multithreading
- Utilizado para delegar tarefas para outro processo
- Libera a main thread para processar eventos UI (User Interface)
- Melhora eficiência do programa e UE (User Experience)
- Não tem acesso ao DOM (Document Object Model)
- Comunicação por mensagens

WEB-Worker:


Event model for web workers
Main script
Listening for message event
Processing message
Send message to worker
message event
message event
Worker
Listening for message event
Processing message
Send message to main script
js-workout.tompascall.com

WEB-Worker:

```
// main.js
var worker = new Worker('worker.js');
worker.onmessage(function(e) {
    console.log(e.data);
})
worker.postMessage('Happy Birthday');
```

```
//worker.js
self.onmessage(function(e) {
//received message e
    var message = e.data + 'to
myself!';
    self.postMessage(message);
    self.close();
})
```

AVALIAÇÃO DA LINGUAGEM

Avaliação da Linguagem

Critérios gerais	C	C++	JAVA	JavaScript
Aplicabilidade	Sim	Sim	Parcial	Sim
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Sim
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim
Método de Projeto	Estruturado	Estruturado e OO	OO	Multiparadigma
Evolutibilidade	Não	Parcial	Sim	Parcial

Avaliação da Linguagem

Critérios gerais	C	C++	JAVA	JavaScript
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim	Sim
Persistência dos dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Serialização (JSON)
Passagem de parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, por valor, e por cópia de referência

Avaliação da Linguagem

Critérios gerais	C	C++	JAVA	JavaScript
Gerenciamento de memória	Programador	Programador	Sistema	Sistema
Encapsulamento e proteção	Parcial	Sim	Sim	Sim
Sistema de tipos	Não	Parcial	Sim	Não
Verificação de tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Todos	Todos
Exceções	Não	Parcial	Sim	Sim
Concorrência	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim	Sim

Referências

- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide
- https://msdn.microsoft.com/pt-br/library/6974wx4d(v=vs.94).aspx
- https://blog.vanila.io/
- https://medium.com/techtrument/multithreading-javascript
- https://devdocs.io/javascript/
- https://developer.mozilla.org
- https://tableless.com.br/primeiros-passos-fullstack-javascript/
- https://stackoverflow.com