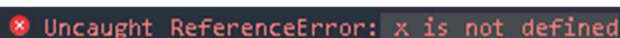


JS STRICT MODE	2
POR QUE USAR STRICT MODE	2
NÃO PERMITIDO NO STRICT MODE	2
ARROW FUNCTION.....	3
ARROW FUNCTION THIS	3
JS ASYNC	4
CALLBACK	4
ASYNCHRONOUS JAVASCRIPT	5
PROMISES	7
ASYNC/AWAIT.....	8
ARRAY ITERATION.....	9
FILTER()	9
MAP()	9
REDUCE().....	9
ARRAY REFERÊNCIA METHODS AND PROPERTIES	10
SPREAD OPERATOR.....	11
JAVASCRIPT ERRORS.....	12
TRY CATCH	12
THROW	12
FINALLY	13
FETCH	14

JS STRICT MODE

O propósito da expressão **"use strict"**; é definir que o JavaScript deve ser executado em "strict mode". Com o strict mode não é possível utilizar variáveis não declaradas, tornando o código mais limpo. A expressão deve ser declarada no começo do script ou da função, declarar no começo para escopo global e no começo de uma função para escopo local.

```
<script>  
"use strict";  
x = 3.14;  
</script>
```

A screenshot of a browser console showing a red error message: "Uncaught ReferenceError: x is not defined". The message is preceded by a red 'x' icon.

POR QUE USAR STRICT MODE

O modo estrito torna mais fácil escrever um código JavaScript "seguro". No JavaScript normal, digitar incorretamente um nome de variável cria uma nova variável global. No modo estrito, isso gerará um erro, impossibilitando a criação acidental de uma variável global. No JavaScript normal, um desenvolvedor não receberá nenhum feedback de erro atribuindo valores a propriedades non-writable. No modo estrito, qualquer atribuição a uma propriedade non-writable, uma propriedade getter-only, uma propriedade non-existing, uma variável non-existing ou um objeto non-existing gerará um erro.

NÃO PERMITIDO NO STRICT MODE

- Usar uma variável, sem declará-la, não é permitido
- Usar um objeto, sem declará-lo, não é permitido
- A exclusão de uma variável (ou objeto) não é permitida
- A exclusão de uma função não é permitida
- Não é permitido duplicar um nome de parâmetro
- Literais numéricos octais não são permitidos
- Caracteres de escape octais não são permitidos
- Não é permitido gravar em uma propriedade somente leitura
- Não é permitido gravar em uma propriedade get-only
- A exclusão de uma propriedade que não pode ser excluída não é permitida
- A palavra eval não pode ser usada como variável
- A palavra arguments não pode ser usada como variável
- A declaração with não é permitida
- Por questões de segurança, eval() não é permitido criar variáveis no escopo do qual foi chamado
- A palavra-chave this em funções se comporta de maneira diferente no modo estrito. A palavra-chave this refere-se ao objeto que chamou a função.
- Palavras-chave reservadas para futuras versões do JavaScript NÃO podem ser usadas como nomes de variáveis no modo estrito.

ARROW FUNCTION

As arrow functions foram introduzidas no ES6 elas permitem escrever uma sintaxe de função mais curta.

```
let myFunction = (a, b) => a * b;
```

Se a função tiver apenas uma instrução e a instrução for retornar um valor, você poderá remover os colchetes e a palavra-chave return:

```
hello = function() {  
  return "Hello World!";  
}  
hello = () => "Hello World!";
```

ARROW FUNCTION THIS

O manuseio do **this** é diferente nas arrow functions em comparação com as funções normais pois não há vinculação de this. Em funções regulares, a palavra-chave this representa o objeto que chama a função, que poderia ser a janela, o documento, um botão ou qualquer outra coisa. Com arrow functions, a palavra-chave this sempre representa o objeto que definiu a função de seta. No seguinte exemplo com uma **função regular**, **this** representa o **objeto que chama** a função:

```
<body>  
<button id="btn">Click Me!</button>  
<p id="demo"></p>  
<script>  
var hello;  
hello = function() {  
  document.getElementById("demo").innerHTML += this;  
}  
window.addEventListener("load", hello);  
document.getElementById("btn").addEventListener("click", hello);  
</script>  
</body>
```

[object Window][object HTMLButtonElement]

Com uma **arrow function**, **this** representa o **proprietário** da função:

```
<body>  
<button id="btn">Click Me!</button>  
<p id="demo"></p>  
<script>  
var hello;  
hello = () => {  
  document.getElementById("demo").innerHTML += this;  
}  
window.addEventListener("load", hello);  
document.getElementById("btn").addEventListener("click", hello);  
</script>  
</body>
```

[object Window][object Window]

JS ASYNC

CALLBACK

Um callback é quando uma função passa um argumento para outra função, uma função pode ser executada apenas quando outra função for finalizada. Em JavaScript uma função é executada na sequência em que ela é chamada, não na sequência em que ela é definida.

No seguinte exemplo, é necessário estabelecer um controle melhor na execução de uma função. No exemplo é desejado realizar um cálculo simples e exibir o seu resultado.

No primeiro exemplo a função de cálculo é chamada (myCalculator), atribuído o resultado a uma variável, e então chamando novamente outra função (myDisplayer) para exibir o resultado.

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
let result = myCalculator(5, 5);
myDisplayer(result);
</script>
```

Do a calculation and then display the result.

10

No segundo exemplo a função é possível chamar somente uma função (myCalculator), e então essa função vai chamar a outra função (myDisplayer).

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}
myCalculator(5, 5);
</script>
```

Do a calculation and then display the result.

10

O problema no primeiro exemplo é que vai ser necessário chamar duas funções para solucionar o resultado e exibi-lo. O problema no segundo exemplo é que, não é possível prevenir a função “myCalculator” de exibir o resultado, mesmo esse não sendo o objetivo dessa função que era somente “calcular”.

É possível chamar uma função (myCalculator) com um callback (outra função como um argumento), e então a função vai realizar o callback depois de terminar o cálculo. Como no seguinte exemplo:

```
<p id="demo"></p>
<script>
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
</script>
```

Do a calculation and then display the result
10

Nesse exemplo, “myDisplayer” é o nome de uma função ela é passada como argumento para outra função “myCalculator()”. É possível observar que quando uma função é utilizada como argumento, não é utilizado os parenteses.

ASYNCHRONOUS JAVASCRIPT

O exemplo anterior foi um exemplo bem simples que tinha apenas o propósito de demonstrar a sintaxe e comportamento das “funções callback”. Geralmente **as funções callback são utilizadas em funções assíncronas**. Funções assíncronas são funções que estão sendo executadas em paralelo a outros funções.

Um exemplo típico é o uso de callback no método **setTimeout()**. Quando uma função setTimeout() é utilizada, é possível especificar uma função callback para ser executada quando terminar o tempo limite especificado. No seguinte exemplo acima “myFunction” é uma função callback que é passada como um argumento para a função “setTimeout()”, somente após 3000 milissegundos “myFunction()” é chamada.

```
<h1 id="demo"></h1>
<script>
setTimeout(myFunction, 3000);
function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World!";
}
</script>
```

Wait 3 seconds (3000 milliseconds) for this page to change.
Hello World!

No seguinte exemplo, “function(){ myFunction(“Hello World!”); }” é usada como callback, é uma função completa que é passada para **setTimeout()** como um argumento.

```
<h1 id="demo"></h1>
<script>
setTimeout(function() { myFunction("Hello World!"); }, 3000);
function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
</script>
```

Wait 3 seconds (3000 milliseconds) for this page to change.
Hello World!

No próximo exemplo é utilizada a função **setInterval()** especificando uma função callback para ser executada a cada intervalo de tempo.

```
<h1 id="demo"></h1>
<script>
setInterval(myFunction, 1000);
function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
    d.getHours() + ":" +
    d.getMinutes() + ":" +
    d.getSeconds();
}
</script>
```

Using setInterval() to display the time every second (1000 milliseconds).

14:14:47

No exemplo anterior “myFunction” é uma função usada como callback, ou seja, um argumento para função “setInterval()”. A cada intervalo de 1000 milissegundos a função “myFunction()” é chamada.

É possível criar uma função que carregue um recurso externo (como um script ou arquivo) e não pode ser executada antes do conteúdo desse recurso ser completamente carregado.

No seguinte exemplo é carregado um arquivo HTML (mycar.html) e então exibido em uma web page somente após ele ser completamente carregado. A função “myDisplayer” é usada como callback pois é passada para “getFile()” como argumento.

```
mycar.html




<p>A car is a wheeled, self-powered motor vehicle used for transportation.
Most definitions of the term specify that cars are designed to run primarily on roads, to have seating
for one to eight people, to typically have four wheels.</p>

<p>(Wikipedia)</p>
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Callbacks</h2>
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}
getFile(myDisplayer);
</script>
</body>
</html>
```

JavaScript Callbacks



A car is a wheeled, self-powered motor vehicle used for transportation. Most definitions of the term specify that cars are designed to run primarily on roads, to have seating for one to eight people, to typically have four wheels.

(Wikipedia)

PROMISES

Uma **promises** é um **objeto** JavaScript que **vincula um código produtor e o código consumidor** ("utilizar no lugar do callback"). O "código produtor" é um código que pode levar algum tempo, já o "código consumidor" é o código que aguarda um resultado. Um objeto Promise contém ambos.

Sintaxe:

```
let myPromise = new Promise(function(myResolve, myReject) {
  // "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject(); // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

O exemplo a seguir demonstra a **utilização de do método Promise()** no **lugar de callback**. O objetivo do exemplo é inserir em um elemento HTML específico um texto após o timeout de 3 segundos.

No seguinte exemplo um arquivo HTML é carregado e depois ele é exibido, é possível fazer isso como callback ou com Promises.

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}
getFile(myDisplayer);
</script>
```

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>
```


ASYNC/AWAIT

Async faz uma função retornar uma promessa. Await faz uma função esperar por uma promessa. As palavras-chave `async` e `await`, implementadas a partir do ES2017, são uma sintaxe que simplifica a programação assíncrona, facilitando o fluxo de escrita e leitura do código; assim é possível escrever código que funciona de forma assíncrona, porém é lido e estruturado de forma síncrona. O `async/await` trabalha com o código baseado em Promises, porém esconde as promessas para que a leitura seja mais fluída e simples de entender. Definindo uma função como `async`, podemos utilizar a palavra-chave `await` antes de qualquer expressão que retorne uma promessa. Dessa forma, a execução da função externa (a função `async`) será pausada até que a Promise seja resolvida. A palavra-chave `await` recebe uma Promise e a transforma em um valor de retorno (ou lança uma exceção em caso de erro). Quando utilizamos `await`, o JavaScript vai aguardar até que a Promise finalize. Se for finalizada com sucesso (o termo utilizado é `fulfilled`), o valor obtido é retornado. Se a Promise for rejeitada (o termo utilizado é `rejected`), é retornado o erro lançado pela exceção.

```
<h1 id="demo"></h1>
<script>
const myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function(){ myResolve("I love You !!"); }, 3000);
});
myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
</script>
```

```
<h1 id="demo"></h1>
<script>
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {resolve("I love You !!");}, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}
myDisplay();
</script>
```

OBS: Só é possível usar `await` em funções declaradas com a palavra-chave `async`. Os dois argumentos (`resolve` e `reject`) são predefinidos pelo JavaScript, mas muitas vezes não é necessário utilizar o `reject`.

ARRAY ITERATION

FILTER()

O método **filter()** cria uma nova array com os elementos array que passam em um teste determinado por uma função callback. No seguinte exemplo temos uma array contendo idade de pessoas e objetivo é identificar quais são adultos (maior de 18 anos) então é utilizado o método **filter()** e passando uma **função callback** que vai ser o teste que será realizado em cada elemento da array.

```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age >= 18;
}
```

32,33,40

MAP()

O método **map()** cria uma nova array utilizando elementos de outra array realizando uma função em cada um. Esse método não altera array original. No seguinte exemplo uma array possui alguns valores numéricos e outra array é criada contendo esses valores multiplicados por 2.

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);
function myFunction(value) {
  return value * 2;
}
```

numbers2 90,8,18,32,50

REDUCE()

O método **reduce()** executa uma função em cada um dos elementos de uma array produzindo (reduzindo para) um único valor. Esse método não altera a array original os valores são reduzidos em uma nova array como no seguinte exemplo.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);
function myFunction(total, value, index, array) {
  return total + value;
}
```

99

Também é possível adicionar um valor inicial.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction, 100);
function myFunction(total, value) {
  return total + value;
}
```

199

ARRAY REFERÊNCIA METHODS AND PROPERTIES

Nome	Descrição
<u>concat()</u>	Combina arrays e retorna uma array com as arrays combinadas
<u>constructor</u>	Retorna a função que criou o protótipo objeto array
<u>copyWithin()</u>	Copia um elemento array em uma posição específica dentro da array
<u>entries()</u>	Retorna um Object Iteration Array (chave/valor)
<u>every()</u>	Checa se todos os elementos em uma array passaram em um teste
<u>fill()</u>	Preenche todos os elementos em uma array com um valor estático
<u>filter()</u>	Cria uma nova array com cada elemento que passou em um teste
<u>find()</u>	Retorna o valor do primeiro elemento que passou em um teste
<u>findIndex()</u>	Retorna o index do primeiro elemento que passou em um teste
<u>forEach()</u>	Chama uma função para cada elemento em uma array
<u>from()</u>	Cria uma array a partir de um objeto
<u>includes()</u>	Verifica se uma array contém um elemento específico
<u>indexOf()</u>	Procura na array por um elemento e retornando a sua posição.
<u>isArray()</u>	Verifica se um objeto é um array
<u>join()</u>	Combina todos os elementos de um array em uma string
<u>keys()</u>	Retorna um Array Iteration Object contendo as chaves da array original
<u>lastIndexOf()</u>	Procura na array por um elemento, começando do final, e retornando a sua posição.
<u>length</u>	Especifica e retorna o numero de elemento de uma array
<u>map()</u>	Cria uma nova array contendo o resultado de uma função chamada em cada elemento de uma array
<u>pop()</u>	Remove o ultimo elemento de uma array e retorna esse elemento
<u>prototype</u>	Permite adicionar propriedades e métodos a um objeto Array
<u>push()</u>	Adiciona novos elementos no final de um array e retorna uma nova largura
<u>reduce()</u>	Reduz os valores de uma array em um único valor (left-to-right)
<u>reduceRight()</u>	Reduz os valores de uma array em um único valor (right-to-left)
<u>reverse()</u>	Inverte a ordem dos elementos de uma array
<u>shift()</u>	Remove o primeiro elemento de uma array, e retorna esse elemento
<u>slice()</u>	Seleciona uma parte de uma array e retorna uma nova array
<u>some()</u>	Verifica se algum elemento de uma array passou em um teste
<u>sort()</u>	Organiza os elementos de uma array
<u>splice()</u>	Adiciona/remove elementos de uma array
<u>toString()</u>	Converte uma array em uma string, e retorna o resultado
<u>unshift()</u>	Adiciona novos elementos para começarem em uma array, e retorna uma nova largura
<u>valueOf()</u>	Retorna o valor primitivo de uma array

SPREAD OPERATOR

O operador spread (...) permite rapidamente criar uma copia de um objeto (ou array) ou de parte dele. No seguinte exemplo existem dois arrays com três valores cada, uma terceira array é criada contendo todos os valores dessas arrays combinados.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo]; 1,2,3,4,5,6
document.write(numbersCombined);
```

No seguinte exemplo o spread operator é utilizado com objetos, criando um terceiro objeto contendo todas as propriedades e valores de outros dois. É possível observar que os valores de algumas propriedades podem ser alterados, e esses valores não vão ser alterados no objeto de referência (as propriedades que não corresponderam foram combinadas, mas a propriedade que correspondeu, color, foi substituída pelo último objeto que foi passado).

```
const myVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'red'
}
const updateMyVehicle = {
  type: 'car',
  year: 2021,
  color: 'yellow'
}
const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle} }
```

```
myUpdatedVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'yellow',
  type: 'car',
  year: 2021,
}
```

É possível ainda adicionar propriedades que não contém em nenhum outro objeto, somente adicionando-a após o sinal de vírgula.

```
const myUpdatedVehicle = {...myVehicle, fuel: 'gasoline', ...updateMyVehicle }

myUpdatedVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'yellow',
  fuel: 'gasoline',
  type: 'car',
  year: 2021,
}
```

JAVASCRIPT ERRORS

Ao executar o código JavaScript, diferentes erros podem ocorrer. Erros podem ser erros de codificação feitos pelo programador, erros devido a entrada errada e outras coisas imprevisíveis.

TRY CATCH

As instruções JavaScript try e catch vêm em pares. A instrução **try** permite que você defina um bloco de código para testar erros enquanto está sendo executado. A instrução **catch** permite definir um bloco de código a ser executado, caso ocorra um erro no bloco try.

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
```

No seguinte exemplo, o método "alert" está digitado incorretamente como "adddler" para produzir deliberadamente um erro:

```
<p id="demo"></p>
<script>
try {
  adddler("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>
```

How to use **catch** to display an error.
adddler is not defined

THROW

A instrução **throw** permite que criar um erro personalizado. Tecnicamente, você pode **throw(lançar) uma exceção (lançar um erro)**. A exceção pode ser um JavaScript String, um Number, um Boolean ou um Object. No seguinte exemplo, é examinado um input, se o valor for errado, uma exceção (err) é lançada (throw an exception). A exceção é capturada pelo catch statement e uma mensagem de erro é exibida.

```
<body>
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>
<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>
```

Please input a number between 5 and 10:

2 Test Input

Input is too low

FINALLY

A instrução **finally** executa o código após try e catch independente do resultado.

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

No seguinte exemplo finally executa o código de limpar os valores dentro do input após o tratamento de erros pelo try e catch.

```
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>
<script>
function myFunction() {
    const message = document.getElementById("p01");
    message.innerHTML = "";
    let x = document.getElementById("demo").value;
    try {
        if(x == "") throw "is empty";
        if(isNaN(x)) throw "is not a number";
        x = Number(x);
        if(x > 10) throw "is too high";
        if(x < 5) throw "is too low";
    }
    catch(err) {
        message.innerHTML = "Input " + err;
    }
    finally {
        document.getElementById("demo").value = "";
    }
}
</script>
```

FETCH

Quando são utilizados arquivos ou dados externos de um servidor ou fornecidos por uma API, isso deve ser feito através de callback ou promises já que serão solicitações assíncronas. E possível observar nos exemplos de callback o uso da função XMLHttpRequest(). Antes de o JSON dominar o mundo, o formato principal de troca de dados era o XML. A XMLHttpRequest() é uma função do JavaScript que tornou possível obter dados das APIs que retornavam dados em XML. **Fetch API é uma versão mais simples** e mais fácil de usar da XMLHttpRequest para consumir recursos de modo assíncrono. Fetch permite que você trabalhe com as APIs REST com opções adicionais como cache de dados, leitura de respostas em streaming e mais.

No seguinte exemplo é obtido um arquivo txt e o texto desse arquivo é introduzido em um elemento <p>.

```
<p id="demo">Fetch a file to change this text.</p>
<script>
let file = "fetch_info.txt"
fetch (file)
.then(x => x.text())
.then(y => document.getElementById("demo").innerHTML = y);
</script>
</body>
</html>
```

Fetch API

The Fetch API interface allows web browser to make HTTP requests to web servers.
If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way.

É possível ainda utilizar async/await para facilitar e melhorar o código.

```
<p id="demo">Fetch a file to change this text.</p>
<script>
getText("fetch_info.txt");
async function getText(file) {
  let x = await fetch(file);
  let y = await x.text();
  document.getElementById("demo").innerHTML = y;
}
</script>
```