Javascript Avançado

1. Eventos do Javascript

Os eventos são as ações que o JavaScript pode detectar. Um bom exemplo é um evento ***onmouseover***, que o JavaScript detecta quando o mouse é movido sobre algum objeto. O evento ***onload*** é acionado assim que uma página da web é carregada. Em JavaScript, eventos e funções são usados juntos. Uma função é acionada quando um evento ocorre. Outros exemplos de eventos incluem fechar uma janela, pressionar uma tecla, redimensionar uma janela e outros.

1.1. Evento *onclick*

Este é um evento comum em JavaScript. Ocorre quando o usuário clica com o botão esquerdo do mouse. Pode-se adicionar um aviso, validação etc. contra o evento. Exemplo:

index.html

```
<body>

    <p>Clique no botão abaixo</p>

    <form>

        <input type="button" onclick="helloFunc()" value="Clique
aqui" />

    </form>

</body>
```

script.js

```
function helloFunc() {

  alert("Alô Mundo!");
```

```
}
```

No exemplo, um formulário simples foi criado.

Clique no botão “Clique aqui” e a função *helloFunc* será invocada. Uma vez clicado, você verá um pop-up

1.2. Evento *onsubmit*

Este evento é chamado quando alguém submeteu um formulário. A validação do formulário pode ser adicionada ao evento. Isso ajudará você a validar os dados antes que eles possam ser enviados para um servidor da web.

index.html

```
<body>

    <form action="http://httpbin.org/post" method="POST"
onsubmit="return formValidation()" >

        <input type="submit" value="Enviar" />

    </form>

</body>
```

script.js

```
function formValidation() {

    // código de validação deve ser adicionado aqui

    return false; //ou true ou false

}
```

1.3. Evento *onchange*

Este método é usado principalmente durante a validação dos campos do formulário. O exemplo a seguir demonstra como usar esse método:

index.html

```
<body>
    Nome: <input type="text" id="name" onchange="testFunc()">
    <p>Saia do campo de entrada e veja seu nome mudar para letras maiúsculas</p>
</body>
```

script.js

```
function testFunc() {
  var a = document.getElementById("name");
  a.value = a.value.toUpperCase();
}
```

Basta digitar seu nome em letras minúsculas e mover o cursor do mouse do campo de entrada clicando fora. Você notará que seu nome será transformado em letras maiúsculas.

O método ***upperCase()*** é invocado depois que o usuário altera o conteúdo do campo de entrada.

1.4. Eventos onmouseover e onmouseout

Esses eventos ajudam você a adicionar recursos atraentes ao texto e às imagens. O evento onmouseover é acionado quando o mouse é movido sobre um objeto. O evento onmouseout é acionado assim que o mouse é movido para fora do elemento.

index.html

```
<body>

    <p>Passe o mouse na divisão e observe:</p>

    <div onmouseover="overObject()" onmouseout="outOfObject()">

        <h2>Divisão</h2>

    </div>

</body>
```

script.js

```
function overObject() {

  alert("Seu Mouse Está Sobre");

}

function outOfObject() {

  alert("Seu Mouse Está Fora");

}
```

Basta mover o mouse conforme as instruções e ver o que acontece. Veja que o texto vai mudar.

Definimos duas funções, ou seja, overObject e outOfObject. Essas duas funções foram então chamadas na seguinte linha:

```
<div onmouseover="overObject()" onmouseout="outOfObject()">
```

Uma vez que o mouse é movido sobre a divisão, o método ***overObject*** será chamado. Assim que o mouse for movido para fora do objeto, o método ***outOfObject*** será chamado. O ***onmouseover="overObject()"*** instrui o JavaScript a chamar a função ***overObject*** assim

que o mouse for movido sobre a divisão, enquanto o ***onmouseout="outOfObject()"***> instrui o interpretador JavaScript a chamar a função ***outOfObject*** assim que o mouse for movido para fora da divisão.

Aqui está outro exemplo que muda a cor do texto após passar o mouse:

```
<body>

    <h1 onmouseover="style.color='yellow'"
onmouseout="style.color='black'">

        Mova o cursor aqui</h1>

</body>
```

1.5. Eventos *onload* e *onunload*

Esses dois eventos são invocados quando um usuário entra ou sai de uma página. Podemos usar o evento ***onload*** para verificar o tipo e a versão do navegador do visitante. É a partir daí que podemos carregar a página da web certa, dependendo da versão. Ambas são boas funções para lidar com cookies:

index.html

```
<body onload="inspectCookies()">

    <p id="example"></p>

</body>
```

script.js

```
function inspectCookies() {

  var txt = "";
```

```
  if (navigator.cookieEnabled == true) {

    txt = "Seu navegador tem cookies ativados.";

  } else {

    txt = "seu navegador está com os cookies desabilitados.";

  }

  document.getElementById("example").innerHTML = txt;

}
```

1.6. Evento *onload*

Quando usado, exibe uma caixa de alerta após o carregamento de uma página. Por exemplo:

index.html

```
<body onload="displayMessage()">
</body>
```

script.js

```
function displayMessage() {
  alert("Uma mensagem pelo evento onload");
}
```

1.7. Evento *onfucus*

Este método faz com que um campo de entrada mude de cor depois de receber o foco.

index.html

```
<body>
```

```
    Nome: <input type="text" onfocus="myFunc(this)">
    <p>Clique dentro do campo de entrada e veja-o mudar de
cor.</p>
</body>
```

script.js

```
function myFunc(col) {
  col.style.background = "green";
}
```

2. Manipulação do DOM (Document Object Model)

O objeto do documento (DOM) refere-se a toda a sua página HTML. Depois de carregar um objeto no navegador da Web, ele imediatamente se torna um objeto de documento, que é o elemento raiz que representa o documento html. Ele vem com propriedades e métodos. O objeto de documento nos ajuda a adicionar conteúdo às páginas da web.

Métodos de DOM

Os métodos DOM são as ações que você pode executar nos elementos html. As propriedades DOM são os valores dos elementos HTML que podem ser definidos ou alterados. A seguir estão os métodos de objeto de documento:

1. ***write("string")***- escreve uma string em um documento.
2. ***writeln("string")***- escreve uma string em um documento com um novo caractere de linha.
3. ***getElementById()***- fornece o elemento com o id especificado.
4. ***getElementsByName()***-fornece todos os elementos com o nome especificado.
5. ***getElementsByTagName()***-fornece todos os elementos com o nome de tag especificado.
6. ***getElementsByClassName()***-fornece todos os elementos com o nome de classe especificado.

Vamos apresentar neste tópico um dos métodos mais utilizados no dia a dia, o ***getElementById()*** para saber mais detalhes dos outros, favor consultar este post.

getElementById()

Além do nome, também podemos obter o elemento por seu id. Isso pode ser feito usando o método ***document.getElementById()***. No entanto, o campo de texto de entrada deve receber um id.

index.html

```
<body>
    <form>
        Digite um número inteiro:<input type="text" id="integer"
name="myNumber"/>
        <br/>
        <input type="button" value="Calcula Raiz Quadrada"
        onclick="computeSquare()"/>
    </form>
</body>
```

script.js

```
function computeSquare(){
  var x=document.getElementById("integer").value;
  alert(x * x);
}
```

No exemplo, definimos o método computeSquare() que nos ajuda a obter o quadrado de um número inserido no campo de texto de entrada. Considere a seguinte linha:

```
  var x=document.getElementById("integer").value;
```

Na linha, usamos o método ***getElementById()*** que usa o id do campo de texto de entrada como argumento. O método nos ajuda a obter o valor digitado no campo de texto de entrada usando seu id.

innerHTML

Esta propriedade pode ser usada para adicionar um conteúdo dinâmico a uma página html. É utilizado em páginas html quando há a necessidade de gerar um conteúdo dinâmico como formulário de comentários, formulário de cadastro, etc.

index.html

```
    <form name="form1" onsubmit="displayform()">
        <input type="button" value="Entre em contato"
onclick="displayform()">
        <div id="area"></div>
    </form>
```

script.js

```
function displayform() {
  let data = "Username:<br><input type='text'
name='name'><br>Comment:";
  data +="<br><textarea rows='6' cols='45'></textarea><br>";
  data +="<input type='submit' value='Contact us'>";
  document.getElementById('area').innerHTML=data;
}
```

O que fizemos foi criar um formulário de contato depois que o usuário clicar em um botão. Observe que o formulário html foi gerado dentro de um div que criamos e demos a ele o nome area. Para identificar a posição, chamamos o método document.getElementById().

3. Objetos

Além dos tipos de dados primitivos como número e *string*, existe outro tipo de dados importante em JavaScript chamado objeto. Objetos são coleções de pares chave-valor. Como tal, eles podem ser usados como array ou dicionários em outras linguagens.

Em outras linguagens, todos os valores em um array geralmente precisam ter o mesmo tipo de dados. Em JavaScript, apenas o tipo da chave é restrito: tem que ser uma string.

Os valores dentro de um objeto podem ter tipos diferentes. Eles podem ser tipos primitivos como números, mas também arrays, outros objetos ou até mesmo funções. Isso torna os objetos muito versáteis, de modo que também são entidades-chave para a programação orientada a objetos (OOP) em JavaScript.

3.1. Criando um objeto

Você cria um objeto usando chaves. Você também pode incluir diretamente algumas entradas. Para isso, indique a chave primeiro, seguida de dois pontos e o valor.

```
const emptyObject = {};



const obj = {

  favoriteNumber: 42,

  greeting: 'Hello World',

  useGreeting: true,

  address: {

    street: 'Trincomalee Highway',

    city: 'Batticaloa',

  },
```

```
  fruits: ['melon', 'papaya'],

  addNumbers: function (a, b) {

    return a + b;

  },

};
```

A vírgula após a última entrada é opcional em JavaScript.

3.2. Recuperando um valor

Existem duas maneiras de recuperar o valor de uma determinada chave, notação de ponto e notação de colchete.

```
const obj = { greeting: 'hello world' };

obj.greeting;
// => hello world

obj['greeting'];
// => hello world

// Bracket notation also works with variables.
const key = 'greeting';
```

```
obj[key];

// => hello world
```

3.3. Adicionar ou alterar um valor

Você pode adicionar ou alterar um valor usando o operador de atribuição =. Novamente, existem notações de ponto e colchetes disponíveis.

```
const obj = { greeting: 'hello world' };


obj.greeting = 'Hi there!';

obj['greeting'] = 'Welcome.';
```

3.4. Excluindo uma entrada

Você pode excluir um par chave-valor de um objeto usando a palavra-chave *delete*.

```
const obj = {
  key1: 'value1',
  key2: 'value2',
};


delete obj.key1;

delete obj['key2'];
```

3.5. Verificando se existe uma chave

Você pode verificar se uma determinada chave existe em um objeto com o método hasOwnProperty.

```
const obj = { greeting: 'hello world' };


obj.hasOwnProperty('greeting');
// => true


obj.hasOwnProperty('age');
// => false
```

3.6. Percorrendo um objeto

Existe um loop *for...in* especial para iterar sobre todas as chaves de um objeto.

```
const obj = {
  name: 'Ali',
  age: 65,
};


for (let key in obj) {
  console.log(key, obj[key]);
}
// name Ali
// age 65
```

4. Função de seta (Arrow Function)

Além das declarações e expressões de função, o JavaScript também possui outra sintaxe muito concisa para definir uma função. Essas funções são chamadas de arrow functions.

Neste conceito, vamos nos concentrar na sintaxe usada para escrever uma função de seta. Existem diferenças na maneira como uma função de seta funciona, como essa associação, que serão abordadas em outros conceitos.

Aqui está uma comparação entre uma declaração de função e uma função de seta.

```
function addUpTwoNumbers(num1, num2) {
  return num1 + num2;
}

// palavra-chave de função removida e => adicionada
const addUpTwoNumbers = (num1, num2) => {
  return num1 + num2;
};
```

Acima, você verá que a sintaxe da função de seta:

1. remove a palavra-chave `function`
2. declarou o identificador `addUpTwoNumbers` como `const`
3. adiciona uma seta grossa `=>`

4.1. Se o corpo da função contiver **apenas uma instrução** `return`, como no exemplo acima, o `{}` e a palavra-chave `return` poderão ser omitidos.

```
const addUpTwoNumbers = (num1, num2) => { return num1 + num2 };

// pode ser encurtado para
const addUpTwoNumbers = (num1, num2) => num1 + num2;
// chaves {} e return removidos
```

4.2. No caso especial de apenas retornar **um objeto** de uma função de seta, **são necessários parênteses** ao redor do objeto para poder omitir a instrução de `return`.

```
// retorno explícito de objeto
const addUpTwoNumbers = (num1, num2) => {
  return {
    num1,
    num2,
  };
};

// retorno implícito de objeto
const addUpTwoNumbers = (num1, num2) => ({ num1, num2 });
```

4.3. O uso de parênteses em torno de parâmetros depende do número de parâmetros.

```
// um parâmetro não precisa de parênteses
const square = num => num * num;

// dois ou mais parâmetros precisam ser colocados entre parênteses
const addUpTwoNumbers = (num1, num2) => num1 + num2;
```

5. Operador ternário

O operador ternário condicional é usado para escrever uma expressão condensada que retorna um dos dois valores alternativos com base em alguma condição. Muitas vezes, é referido como o "operador ternário". O nome decorre do fato do operador possuir três operandos:
`predicado ? expressão-conseqüente : expressão-alternativa`
Pode ser usado como um substituto para instruções if-else curtas.

Semelhante às instruções if, o JavaScript executará a conversão de tipo implícita para avaliar a condição. Se a condição for verdadeira, o operando do lado esquerdo dos dois pontos será retornado. Caso contrário, o resultado da
expressão ternária é o operando no lado direito dos dois pontos.

```
const year = 2020;

year > 2000 ? 'nos últimos anos' : 'a muito tempo atrás';
// => 'nos últimos anos'
```

6. Transformação de Array

Em JavaScript, a classe `Array` tem muitos métodos integrados poderosos para transformar arrays. Esses métodos tornam muito mais fácil converter um array em outro do que seria usando um loop for simples ou uma manipulação mais direta.

Alguns métodos são **puros**, o que significa que não modificam o array original. Em vez disso, eles retornam um novo. Outros métodos, entretanto, manipulam o array em que são chamados e não retornam o array modificado.

Alguns dos métodos mais comumente usados para transformar arrays são apresentados a seguir.

6.1. map() - pura

Cria um novo array transformando cada elemento de acordo com uma função passada como argumento. Essas funções de retorno de chamada (callback) geralmente são escritas como funções de seta (arrow functions).

script.js

```
let arr = [1, 2, 3, 4];

const newArr = arr.map((value) => value - 1);
console.log(newArr);
// => [0, 1, 2, 3]
console.log(arr);
```

```
// => [1, 2, 3, 4]
```

Vale a pena notar que o array resultante sempre terá o mesmo tamanho do original.

6.2. filter() - pura

Cria um array filtrando o atual, dada uma função de filtragem (que retorna true se o elemento deve ser mantido e false se deve ser removido).

script.js

```
let arr = [1, 2, 3, 4];

arr.filter((value) => value % 2 === 0);
// => [2, 4]
```

6.3. reduce() - pura

Reduz a array a um único valor usando uma função que usa um acumulador e o elemento atual da array como parâmetros. Esta função instrui como o elemento atual deve ser mesclado no acumulador e retorna o acumulador que será usado na próxima iteração.

script.js

```
// Obter a soma dos elementos
const numbers = [1, 2, 3, 4, 5]
const sum = numbers.reduce((accumulator, currentValue) =>
accumulator + currentValue, 0)

console.log(sum)
```

script.js

```
// Classifique os números se são ímpares ou não
const arr = [1, 2, 3, 4]

newArray = arr.reduce(
  (accumulator, currentValue) => {
    if (currentValue % 2 === 0) {
      accumulator.even.push(currentValue);
    } else {
      accumulator.odd.push(currentValue);
    }

    return accumulator;
  },
  { even: [], odd: [] }
);

console.log(newArray);

// => { even: [2, 4], odd: [1, 3] }
```

index.js

```
const videoGames = [
    {
      name: 'The Binding of Isaac',
      category: 'Roguelike'
    },
    {
      name: 'Overwatch 2',
      category: 'FPS'
    },
    {
      name: 'Vampire Survivors',
      category: 'Roguelike'
    },
    {
      name: 'Valorant',
```

```
      category: 'FPS'
    },
    {
      name: 'The Legend Of Zelda: Tears of the Kingdom',
      category: 'Adventure'
    }
  ];

  const categories = videoGames.reduce((acc, val) =>
acc.includes(val.category) ? acc : acc.concat(val.category), []);

  console.log(categories) // [ "Roguelike", "FPS", "Adventure" ]
```

index.js

```
const items = [
    { id: '🍔', name: 'Super Burger', price: 399 },
    { id: '🍟', name: 'Jumbo Fries', price: 199 },
    { id: '🥤', name: 'Big Slurp', price: 299 }
  ];

  const reduced = items
  .map(item => item.price)
  .reduce((prev, next) => prev + next);

// Total: 8.97
console.log(reduced);
```

6.4. reverse()

Inverte os elementos de uma array.

script.js

```
const arr = [1, 2, 3, 4];
```

```
const newArray = arr.reverse();

console.log(newArray);
// => [4, 3, 2, 1]
```

Este método modifica o array no qual é chamado.

6.5. slice() - pura

Dado um índice inicial e um final, cria um sub-array a partir do array passado como parâmetro.

O elemento no índice final não será incluído. Além disso, todos os parâmetros são opcionais: o índice inicial é padronizado como `0` e o índice final é padronizado com o tamanho do array.

```
const arr = [1, 2, 3, 4];

const arr1 = arr.slice(1, 2); // [2]
console.log(arr1);

const arr2 = arr.slice(1); // [2, 3, 4]
console.log(arr2);

// You can also use negative numbers, that represent the indexes
// starting from the end of the array
const arr3 = arr.slice(-2); // [3, 4]
console.log(arr3);

console.log(arr);
```

6.6. splice()

Remove ou substitui e/ou adiciona novos elementos em um array.

Leva os seguintes parâmetros:

- o índice do elemento onde começar a modificar o array
- o número de elementos a serem excluídos
- os elementos a inserir no array (opcional)

splice retorna os elementos que foram removidos.

script.js

```
const arr = ['1', '2', '5', '6'];

// Inserir um elemento no índice 2
arr.splice(2, 0, '3');
console.log(arr);
// => ['1', '2', '3', '5', '6']

// Remova 2 elementos, começando no índice 3 e insira 2 elementos
const removed = arr.splice(3, 2, '4', '5');
console.log(removed);
// => ['5', '6']
console.log(arr);
// => ['1', '2', '3', '4', '5']

// Remova 1 elemento no índice 1
arr.splice(1, 1);
console.log(arr);
// => ['1', '3', '4', '5']
```

6.7. sort ()

Por padrão, `sort` classifica os elementos de uma array convertendo-os primeiro em strings e, em seguida, aplicando a comparação de strings. A classificação ocorre no local,

o que significa que a array original é modificada. `sort` também retorna aquele array modificado que é conveniente se você quiser encadear outros métodos a ele.

script.js

```
const arr = ['c', 'a', 'z', 'b'];
const result = arr.sort();
console.log(result);
// => ['a', 'b', 'c', 'z']
console.log(arr);
// => ['a', 'b', 'c', 'z']
```

- **Classificação numérica**

Para personalizar o comportamento de classificação (crescente ou decrescente), você pode passar uma função de comparação como argumento.

```
const arr = [3, 1, 2, 10];
// Classificação crescente
const arrCresc = arr.sort((a, b) => a - b);
console.log(arrCresc);
// => [1, 2, 3, 10]

// Classificação decrescente
const arrDecresc = arr.sort((a, b) => b - a);
console.log(arrDecresc);
// => [10, 3, 2, 1]
```

7. Desestruturação de Array

A sintaxe de desestruturação de array do JavaScript é uma maneira concisa de extrair valores de um array e atribuí-los a variáveis distintas.

Neste exemplo, cada valor na array `numberOfMoons` é atribuído ao seu planeta correspondente:

script.js

```
const numberOfMoons = [0, 2, 14];
const [venus, mars, neptune] = numberOfMoons;

console.log(neptune);
// => 14
```

7.1. Desestruturando durante uma interação

script.js

```
const countries = [['Finland', 'Helsinki'], ['Sweden',
'Stockholm'], ['Norway', 'Oslo']]

for (const [country, city] of countries) {
console.log(country, city)
}
```

7.2. Desestruturando objeto

Quando desestruturamos o nome da variável que usamos para desestruturar deve ser exatamente o mesmo que a chave ou propriedade do objeto. Veja o exemplo abaixo.

```
const rectangle = {
  width: 20,
  height: 10,
  area: 200,
};
let { width, height, area, perimeter } = rectangle;
```

```
console.log(width, height, area, perimeter);
```

Renomeando durante a estruturação

```
const rectangle = {
  width: 20,
  height: 10,
  area: 200,
};
let { width: w, height: h, area: a, perimeter: p } = rectangle;

console.log(w, h, a, p);
```

Parâmetro do objeto sem desestruturação

```
const rect = {
  width: 20,
  height: 10,
};
const calculatePerimeter = (rectangle) => {
  return 2 * (rectangle.width + rectangle.height);
};

console.log(calculatePerimeter(rect));
```

Parâmetro do objeto com desestruturação

```
const rect = {
  width: 20,
  height: 10,
};
const calculatePerimeter = ({ width, height }) => {
  return 2 * (width + height);
};

console.log(calculatePerimeter(rect));
```

Desestruturando o objeto durante a iteração

```
const todoList = [
  {
    task: "Prepare JS Test",
    time: "4/1/2020 8:30",
    completed: true,
  },
  {
    task: "Give JS Test",
    time: "4/1/2020 10:00",
    completed: false,
  },
  {
    task: "Assess Test Result",
    time: "4/1/2020 1:00",
    completed: false,
  },
];

for (const { task, time, completed } of todoList) {
  console.log(task, time, completed);
}
```

8. Operadores rest e spread

O JavaScript possui um operador `...` embutido que torna mais fácil trabalhar com um número indefinido de elementos. Dependendo do contexto, ele é chamado de operador rest ou operador spread.

8.1. Operador rest

8.1.1. Elemento rest

Quando `...` aparece no lado **esquerdo** de uma atribuição, esses três pontos são conhecidos como operador rest. Os três pontos junto com um nome de variável são chamados de elemento `rest`. Ele coleta zero ou mais valores e os armazena em um único array.

script.js

```
const [a, b, ...everythingElse] = [0, 1, 1, 2, 3, 5, 8];

console.log(a);
// => 0

console.log(b);
// => 1

console.log(everythingElse);
// => [1, 2, 3, 5, 8]
```

Observe que em JavaScript, ao contrário de algumas outras linguagens, um elemento rest não pode ter uma vírgula à direita. Deve ser o último elemento em uma atribuição de desestruturação. O exemplo abaixo gera um SyntaxError:

```
const [...items, last] = [2, 4, 8, 16]
```

8.1.2. Propriedades rest

Da mesma forma que arrays, o operador rest também pode ser usado para coletar uma ou mais propriedades de objeto e armazená-las em um único objeto.

script.js

```
const { street, ...address } = {
  street: 'Platz der Republik 1',
  postalCode: '11011',
  city: 'Berlin',
};

console.log(street);
// => 'Platz der Republik 1'

console.log(address);
// => {postalCode: '11011', city: 'Berlin'}
```

8.1.3. Parâmetros rest

Quando `...` aparece em uma definição de função ao lado de seu último argumento, esse parâmetro é chamado de parâmetro rest. Ele permite que a função aceite um número indefinido de argumentos como uma array.

```
function concat(...strings) {
  return strings.join(' ');
}

concat('one');
// => 'one'

concat('one', 'two', 'three');
// => 'one two three'
```

8.2. spread

8.2.1. Elemento spread

Quando `...` aparece no **lado direito** de uma atribuição, é conhecido como operador de spread. Ele expande uma array em uma lista de elementos. Ao contrário do elemento `rest`, ele pode aparecer em qualquer lugar em uma expressão literal de array e pode haver mais de um.

script.js

```
const oneToFive = [1, 2, 3, 4, 5];
const oneToTen = [...oneToFive, 6, 7, 8, 9, 10];

console.log(oneToTen);
// => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const woow = ['A', ...oneToFive, 'B', 'C', 'D', 'E', ...oneToFive,
42];

console.log(woow);
// =>  ["A", 1, 2, 3, 4, 5, "B", "C", "D", "E", 1, 2, 3, 4, 5, 42]
```

8.2.2. Propriedades spread

Da mesma forma que arrays, o operador spread também pode ser usado para copiar propriedades de um objeto para outro.

```
let address = {
  postalCode: '11011',
  city: 'Berlin',
};

address = { ...address, country: 'Germany' };

console.log(address);
```

```
// => {
//   postalCode: '11011',
//   city: 'Berlin',
//   country: 'Germany',
// }
```

9. JSON

JSON significa JavaScript Object Notation.
JSON é um formato de texto para armazenar e transportar dados.

O formato JSON é sintaticamente idêntico ao código para criar objetos JavaScript. Devido a essa semelhança, um programa JavaScript pode facilmente converter dados JSON em objetos JavaScript nativos.

Exemplo de um Objeto JSON:

```
{
  "users":[
    {
      "firstName":"Asabeneh",
      "lastName":"Yetayeh",
      "age":250,
      "email":"asab@asb.com"
    },
    {
      "firstName":"Alex",
      "lastName":"James",
      "age":25,
      "email":"alex@alex.com"
    },
    {
    "firstName":"Lidiya",
    "lastName":"Tekle",
    "age":28,
    "email":"lidiya@lidiya.com"
```

```
    }
  ]
  }
```

Os dados JSON são gravados como pares de **nome/valor**, assim como as propriedades do objeto JavaScript.

Um par nome/valor consiste em um nome de campo (entre **aspas duplas**), seguido por dois pontos, seguido por um valor:

```
"firstName":"John"
```

O exemplo de um objeto JSON acima não é muito diferente de um objeto normal. Então, qual é a diferença? A diferença é que a **chave de um objeto JSON deve estar entre aspas duplas** ou deve ser uma string. JavaScript Object e JSON são muito semelhantes, podemos mudar JSON para Objeto e Objeto para JSON.A seguir apresentaremos as funções que fazem esta transformação.

JSON.parse()

Um uso comum de JSON é trocar dados de/para um servidor web.
Ao receber dados de um servidor web, os dados são sempre uma **string**.
Analise os dados com JSON.parse() e os dados se tornarão um objeto JavaScript.

Imagine que recebemos este texto de um servidor web:

```
'{"name":"John", "age":30, "city":"New York"}'
```

Use a função JavaScript ***JSON.parse()*** para converter texto em um objeto JavaScript:

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New
York"}');
```

JSON.stringify()

Um uso comum de JSON é trocar dados de/para um servidor web. Ao enviar dados para um servidor web, os dados devem ser uma ***string***. Converta um objeto JavaScript em uma string com ***JSON.stringify()***.

Imagine que temos este objeto em JavaScript:

```
const obj = {name: "John", age: 30, city: "New York"};
```

Use a função JavaScript ***JSON.stringify()*** para convertê-lo em uma string.

```
const myJSON = JSON.stringify(obj);
```

Esta é uma string JSON:

```
'{"name":"John", "age":30, "car":null}'
```

Dentro da string JSON existe um objeto JSON literal:

```
{"name":"John", "age":30, "car":null}
```

Os literais de objeto JSON são colocados entre chaves {}.
Os literais de objeto JSON contêm pares chave/valor.
Chaves e valores são separados por dois pontos.

As chaves devem ser strings e os valores devem ser um tipo de dados JSON válido:

- string
- numérico
- objeto
- array
- booleano
- null

Cada par chave/valor é separado por uma vírgula.

- É um erro comum chamar um objeto JSON literal de "um objeto JSON".
- JSON não pode ser um objeto.
- JSON é um formato de string.

- Os dados são apenas JSON quando estão em formato de string.
- Quando é convertido em uma variável JavaScript, torna-se um objeto JavaScript.

10. Função Callback

Funções de retorno de chamada (***callback***) são funções passadas como argumentos. Esse padrão de programação cria uma sequência de chamadas de função na programação síncrona e assíncrona. Escrever uma função de retorno de chamada não é diferente de escrever uma função; no entanto, a função de retorno de chamada deve corresponder à assinatura definida pela função de chamada.

index.html

```
    <h1>Funçoes Javascript</h1>
    <h2>Funções de Callback</h2>

    <p>O resultado do cálculo é:</p>
    <p id="demo"></p>
```

script.js

```
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

11. Promises

Uma promise (promessa) é um objeto que representa a eventual conclusão ou falha de uma operação assíncrona.
Essencialmente, uma promise é um objeto retornado ao qual você anexa callbacks, em vez de passar callbacks para uma função.

No ES6, Promises são uma **forma de implementar programação assíncrona**. Uma promise se tornará um contêiner para valor futuro. Por exemplo, se você pedir qualquer comida em qualquer site para entregá-la em seu local, esse registro de pedido será a promessa e a comida será o valor dessa promessa.

```
let p = new Promise((resolve,reject)=>{
  // código pendente
let x = 5;
let y = 5;
let z = x + y;

if(z == 10)
{
    resolve('ok');
}
else{
    reject('error');
}
})

// usando o promise
console.log("start")
p.then((val)=>{ // Resolvido
console.log(val)
}).catch((err)=>{ //rejeitado
console.log(err)
})
```

Estado das promises

Em primeiro lugar, uma Promise é um objeto. Existem 3 estados do objeto Promise:

- **Pendente**: Estado inicial, antes que a promessa seja bem-sucedida ou falhe.
- **Resolvido**: Promessa Concluída
- **Rejeitado**: promessa com falha, lançar um erro

Exemplo:

```
const myFirstPromise = new Promise((resolve, reject) => {
  const condition = true;
  if(condition) {
       setTimeout(function(){
           resolve("Promise está resolvida!"); // realizado
      }, 300);
  } else {
      reject('Promise está rejeitada!');
  }
});
// Uso da promise acima da seguinte forma
myFirstPromise
.then((successMsg) => {
  console.log(successMsg);
})
.catch((errorMsg) => {
  console.log(errorMsg);
});
```

Exemplo um pouco complexo:

```
const demoPromise= function() {
  myFirstPromise
  .then((successMsg) => {
      console.log("Sucesso:" + successMsg);
  })
  .catch((errorMsg) => {
      console.log("Erro:" + errorMsg);
  })
}
```

```
demoPromise();
```

Encadeamento em promises:

Às vezes, precisamos chamar várias solicitações assíncronas e, depois que a primeira Promise for resolvida (ou rejeitada), um novo processo será iniciado ao qual podemos anexá-lo diretamente por um método chamado encadeamento.

Exemplo:

```
const helloPromise  = function() {
  return new Promise(function(resolve, reject) {
    const message = `Oi, Como você está!`;

    resolve(message)
  });
}
```

Encadeamos essa promessa à nossa operação “myFirstPromise” anterior da seguinte forma:

```
const demoPromise= function() {

  myFirstPromise
  .then(helloPromise)
  .then((successMsg) => {
      console.log("Sucesso:" + successMsg);
  })
  .catch((errorMsg) => {
      console.log("Erro:" + errorMsg);
  })
}

demoPromise();
```

12. Async/Await

Await é basicamente um adoçante para Promises. Isso faz com que seu código assíncrono pareça mais um código síncrono/procedural, sendo mais fácil o entendimento.

Sintaxe de Async e Await:

```
async function printMyAsync(){
  await printString("um")
  await printString("dois")
  await printString("três")
}
```

```
function printString(num) {
  let tempo;
  setTimeout(function () {
    console.log(num);
    switch (num) {
      case "um":
        tempo = 500;
        break;
      case "dois":
        tempo = 1000;
        break;
      case "três":
        tempo = 13500;
        console.log(tempo);
        break;
    }
  }, tempo);
  console.log(num);
}

printMyAsync();
```

Vamos dar um exemplo para entender o Async e o Await com nosso demoPromise:

```
async function demoPromise() {
  try {
    let message = await myFirstPromise;
    let message  = await helloPromise();
    console.log(message);

  }catch((error) => {
      console.log("Erro:" + error.message);
  })
}

//Uso da função async da seguinte forma:
(
async ()=>{
    await demoPromise()
}
)
```

13. Fetch API

A Fetch API fornece uma interface para buscar recursos (incluindo através da rede). Parecerá familiar para quem já usou XMLHttpRequest, mas a nova API fornece um conjunto de recursos mais poderoso e flexível. Neste desafio, usaremos fetch para solicitar url e APIS. Além disso, vamos ver a demonstração do caso de uso de promessas no acesso a recursos de rede usando a API de busca.

Com **promises**

```
const url = 'https://restcountries.com/v2/all' // api de países
fetch(url)
  .then(response => response.json()) // acessando a API de dados
como JSON
  .then(data => {
    // obtendo dados
    console.log(data)
  })
  .catch(error => console.error(error)) // tratando erro se der
algo de errado
```

Com **async e await**

```
const fetchData = async () => {
    try {
      const response = await fetch(url)
      const countries = await response.json()
      console.log(countries)
    } catch (err) {
      console.error(err)
    }
  }
  console.log('===== async and await')
  fetchData()
```

14. Manipulação de erros