MÓDULO VI

Lógica de Programação utilizando a Linguagem JavaScript

A Educação é o primeiro passo para um futuro melhor...

MÓDULO VI

Laços em JavaScript explicados: laços for, while, do...while e mais

Os laços são usados em **JavaScript** para realizar tarefas repetidas com base em uma **condição**. As **condições** tipicamente retornam **true** ou **false**.

Um laço continuará em execução até que a **condição** definida retorne **false**.

MÓDULO VI

Laço `for`

Sintaxe

```
for (inicialização; condição; expressãoFinal) {
  // instruções a serem executadas repetidamente
}
```

MÓDULO VI

O laço for consiste em três expressões e uma sequência de instruções:

- **inicialização** - esta expressão é executada antes da execução do primeiro laço, sendo geralmente usada para criar um contador.
  A variável criada aqui fica no escopo do laço. Tão logo o laço se encerre, ela é destruída.
- **condição** - esta expressão é verificada a cada iteração antes da execução do laço.
  Se ela for **true**, as instruções ou o código do laço é executado.
  Se ela for **false**, o laço é interrompido. Se essa expressão for omitida, será sempre **true**.
- **expressãoFinal** - esta expressão é executada após cada iteração do laço.
  Ela é usada normalmente para **incrementar** um contador, mas também pode ser usada para **decrementá-lo**.
- **instruções** - o código dentro do bloco a ser repetido no laço.
  Pode ser uma única ou várias linhas de código.

MÓDULO VI

Qualquer uma das três expressões ou mesmo as próprias instruções – o código dentro do bloco – podem ser omitidas.

Laços for são normalmente usados para repetir instruções/executar código por um número definido de vezes. Além disso, é possível usar a instrução **break** para sair do laço precocemente, antes de a expressão de condição ser **false**.

MÓDULO VI

Exemplos

1. Iterar por números inteiros de 0 a 8:

```
for (let i = 0; i < 9; i++) {
  console.log(i);
}

// Resultado:
// 0
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
```

MÓDULO VI

2. Use break para sair de um laço for antes de a expressão da condição ser false:

```
for (let i = 1; i < 10; i += 2) {
  if (i === 7) {
    break;
  }
  console.log('Número total de elefantes: ' + i);
}

// Resultado:
// Número total de elefantes: 1
// Número total de elefantes: 3
// Número total de elefantes: 5
```

MÓDULO VI

Armadilha comum: exceder os limites de um array

Ao iterar por um array, é fácil exceder os limites dele acidentalmente.
Por exemplo, seu laço pode tentar fazer referência ao 4º elemento de um array que tem apenas 3 elementos:

MÓDULO VI

Há duas maneiras de consertar esse código: definir a expressão de condição como **i < arr.length ou i <= arr.length - 1**.

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

for (let i = 0; i <= arr.length; i++) {
  console.log(arr[i]);
}

// Resultado:
// 1
// 2
// 3
// undefined
```

MÓDULO VI

Laço `for...in`

Sintaxe

```
for (variável in objeto) {
  // instruções
}
```

MÓDULO VI

O laço **for...in** itera pelas propriedades de um objeto.
Para cada propriedade, as instruções dentro do laço são executadas.

Exemplos

1. Iterar pelas propriedades de um objeto e imprimir suas chaves e valores no console:

MÓDULO VI

```
const capitais = {
  a: "Atenas",
  b: "Belgrado",
  c: "Cairo"
};

for (let chave in capitais) {
  console.log(chave + ": " + capitais[chave]);
}

// Resultado:
// a: Atenas
// b: Belgrado
// c: Cairo
```

MÓDULO VI

Armadilha comum: comportamento inesperado ao iterar por um array

Embora você possa usar um laço **for...in** para iterar por um **array**, é recomendado usar um laço **for** ou **for...of** nesse caso.

O laço **for...in** pode iterar por **arrays** e **objetos** semelhantes a um **array**, mas nem sempre pode acessar os índices de um **array** em ordem.

Além disso, o laço **for...in** retorna todas as propriedades e propriedades herdadas de um **array** ou de um **objeto** semelhante a um **array**, o que pode levar a um comportamento inesperado.

MÓDULO VI

Por exemplo, este laço simples funciona de acordo com o que é esperado:

```
const array = [1, 2, 3];

for (const i in array) {
  console.log(i);
}

// 0
// 1
// 2
```

MÓDULO VI

Porém, se algo como uma biblioteca do **JS** que você está usando modificar o protótipo de **Array** diretamente, um laço **for...in** também fará a iteração por ele:

```
const array = [1, 2, 3];

Array.prototype.umMetodoQualquer = true;

for (const i in array) {
  console.log(i);
}

// 0
// 1
// 2
// umMetodoQualquer
```

MÓDULO VI

Embora modificar protótipos somente leitura, como **Array** ou **Object**, diretamente não seja uma boa prática, isso pode ser o problema de algumas bibliotecas ou bases de código.

Além disso, como **for...in** é destinado a **objetos**, ele é muito mais lento com **arrays** do que os outros laços.

Em resumo, lembre-se de usar os laços **for...in** somente para iterar por **objetos**, não por **arrays**.

MÓDULO VI

Laço `for...of`

Sintaxe

```
for (variável of objeto) {
  // instruções
}
```

MÓDULO VI

O laço **for...of** itera pelos valores de muitos tipos de iteráveis, incluindo **arrays** e tipos especiais de coleções, como **Set** e **Map**. Para cada valor no objeto iterável, as instruções dentro do laço são executadas.

Exemplos

1. Iterar por um array:

```
const arr = [ "Fred", "Tom", "Bob" ];

for (let i of arr) {
  console.log(i);
}

// Resultado:
// Fred
// Tom
// Bob
```

MÓDULO VI

2. Iterar por um Map:

```
const m = new Map();
m.set(1, "preto");
m.set(2, "vermelho");

for (let n of m) {
  console.log(n);
}

// Resultado:
// [1, preto]
// [2, vermelho]
```

MÓDULO VI

3. Iterar por um Set:

```
const s = new Set();
s.add(1);
s.add("vermelho");

for (let n of s) {
  console.log(n);
}

// Resultado:
// 1
// vermelho
```

MÓDULO VI

Laço `while`

Sintaxe

```
while (condição) {
  // instruções
}
```

MÓDULO VI

O laço **while** começa avaliando a condição. Se a conditição for **true**, o código dentro do bloco será executado. Se a condição for **false**, o código dentro do bloco não será executado e o laço se encerra.

Exemplos:

1. Quando uma variável for inferior a **10**, imprima-a no console e some a ela mais **1**:

MÓDULO VI

```
let i = 1;

while (i < 10) {
  console.log(i);
  i++;
}

// Resultado:
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

MÓDULO VI

Laço `do...while`

Sintaxe:

```
do {
  // instruções
} while (condição);
```

MÓDULO VI

O laço **do...while** é fortemente associado ao laço **while**. Em um laço **do...while**, a expressão da condição é verificada ao final de cada iteração do laço, em vez de no começo, antes de o laço ser executado.

Isso significa que o código em um laço **do...while** com certeza será executado pelo menos uma vez, mesmo se a expressão da condição já for avaliada como **false**.

MÓDULO VI

Exemplo:

1. Enquanto uma variável for inferior a 10, imprima-a no console e some a ela mais 1:

```
let i = 1;

do {
  console.log(i);
  i++;
} while (i < 10);

// Resultado:
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

MÓDULO VI

2. Enviar para um array, mesmo se a condição for true:

```
const myArray = [];
let i = 10;

do {
  myArray.push(i);
  i++;
} while (i < 10);

console.log(myArray);

// Resultado:
// [10]
```

MÓDULO VI

Métodos Úteis para Manipulação de Arrays em JavaScript

Alguns dos métodos mais comuns são:

- **push()**: Adiciona um ou mais elementos ao final do array.
- **pop()**: Remove o último elemento do array.
- **shift()**: Remove o primeiro elemento do array.
- **unshift()**: Adiciona um ou mais elementos no início do array.
- **slice()**: Retorna uma cópia superficial de uma parte do array.
- **splice()**: Modifica o conteúdo de um array, adicionando, removendo ou substituindo elementos.
- **concat()**: Combina dois ou mais arrays retornando um novo array resultante da concatenação.

MÓDULO VI

Como utilizar os métodos de array em JavaScript

Para utilizar os métodos de **array** em **JavaScript**, primeiro é necessário ter um **array** definido.

Por exemplo, podemos criar um **array** de números da seguinte forma:

```
let numeros = [1, 2, 3, 4, 5];
```

MÓDULO VI

A partir desse **array**, podemos utilizar os métodos de **array** para realizar diferentes operações. Por exemplo, se quisermos adicionar um novo número ao final do **array**, utilizamos o método **push()**:

```
numeros.push(6);
console.log(numeros);
```

Neste caso, o resultado impresso no console será: [1, 2, 3, 4, 5, 6].

MÓDULO VI

Outro exemplo seria utilizar o método **pop()** para remover o último elemento do **array**:

```
numeros.pop();
console.log(numeros);
```

O resultado impresso no console será: [1, 2, 3, 4, 5].

MÓDULO VI

Exemplos de aplicação dos métodos de array em JavaScript

Exemplo 1: Filtrar números pares de um array

Suponha que temos um **array** de números e queremos filtrar apenas os números pares.

Podemos utilizar o método **filter()** em conjunto com uma função de **callback** para realizar essa tarefa:

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let numerosPares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});

console.log(numerosPares);
```

O resultado impresso no console será: [2, 4, 6, 8, 10].

MÓDULO VI

Exemplo 2: Encontrar a soma total de um array de números

Vamos supor que temos um **array** de números e queremos calcular a soma total.

Podemos utilizar o método **reduce()** para obter esse resultado:

```
let numeros = [1, 2, 3, 4, 5];

let somaTotal = numeros.reduce(function(acumulador, numero) {
  return acumulador + numero;
}, 0);

console.log(somaTotal);
```

O resultado impresso no console será: **15**.

MÓDULO VI

Considerações finais sobre os métodos de array em JavaScript

Os métodos de **array** em **JavaScript** são ferramentas extremamente úteis para manipulação e transformação de **arrays**. Eles nos ajudam a escrever código mais legível, eficiente e com menos repetição.

Em resumo, os métodos de **array** em **JavaScript** são uma parte fundamental da linguagem e podem ser utilizados para facilitar o trabalho com **arrays** em seus projetos. Espero que este artigo tenha sido útil para entender a importância e o potencial desses métodos em suas aplicações **JavaScript**.

MÓDULO VI

Como utilizar os métodos de array em JavaScript

Agora que já conhecemos os métodos de **array** em **JavaScript** e sabemos quais são as principais funções que eles desempenham, vamos aprender como utilizá-los em nossos códigos.

A seguir, irei apresentar algumas dicas e exemplos práticos para ajudar você a utilizar esses métodos de forma eficiente e produtiva.

MÓDULO VI

1. Acessando elementos do array

Para acessar elementos específicos em um **array**, podemos utilizar o índice do elemento desejado dentro de colchetes (**[]**). O índice de um **array** começa em **0**, ou seja, o primeiro elemento está no índice **0**, o segundo no índice **1** e assim por diante. Veja o exemplo abaixo:

```
let numeros = [1, 2, 3, 4, 5];

console.log(numeros[0]); // imprime 1
console.log(numeros[2]); // imprime 3
```

MÓDULO VI

2. Adicionando elementos ao array

Para adicionar elementos a um **array**, podemos utilizar o método **push()**.

Esse método adiciona um ou mais elementos ao final do **array**. Veja o exemplo abaixo:

```
let numeros = [1, 2, 3];

numeros.push(4);
console.log(numeros); // imprime [1, 2, 3, 4]

numeros.push(5, 6);
console.log(numeros); // imprime [1, 2, 3, 4, 5, 6]
```

MÓDULO VI

3. Removendo elementos do array

Existem várias maneiras de remover elementos de um **array** em **JavaScript**.

Alguns métodos comumente utilizados são:

- Método **pop()**: Remove o último elemento do **array** e o retorna.
- Método **shift()**: Remove o primeiro elemento do **array** e o retorna.
- Método **splice()**: Remove um ou mais elementos do **array** em uma posição específica.

MÓDULO VI

```
let numeros = [1, 2, 3, 4, 5];

let ultimoElemento = numeros.pop();
console.log(numeros); // imprime [1, 2, 3, 4]
console.log(ultimoElemento); // imprime 5

let primeiroElemento = numeros.shift();
console.log(numeros); // imprime [2, 3, 4]
console.log(primeiroElemento); // imprime 1

let elementosRemovidos = numeros.splice(1, 2);
console.log(numeros); // imprime [2]
console.log(elementosRemovidos); // imprime [3, 4]
```

MÓDULO VI

4. Percorrendo um array

Para percorrer os elementos de um **array**, podemos utilizar um **loop**, como o **for loop** ou o **forEach()**.

Veja o exemplo abaixo utilizando o **forEach()**:

```
let numeros = [1, 2, 3, 4, 5];

numeros.forEach(function(numero) {
  console.log(numero);
});
```

Este código irá imprimir cada elemento do array no console.

MÓDULO VI

5. Transformando um array

Existem vários métodos que nos permitem transformar os elementos de um array.

Alguns exemplos são:

- Método **map()**: Cria um novo **array** com os resultados de uma função aplicada a cada elemento do **array** original.
- Método **filter()**: Cria um novo **array** contendo apenas os elementos do **array** original que satisfazem a condição especificada em uma função.

MÓDULO VI

```
let numeros = [1, 2, 3, 4, 5];

let multiplicadosPorDois = numeros.map(function(numero) {
  return numero * 2;
});

console.log(multiplicadosPorDois); // imprime [2, 4, 6, 8, 10]

let pares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});

console.log(pares); // imprime [2, 4]
```

MÓDULO VI

Essas são apenas algumas das formas de utilizar os métodos de array em JavaScript.

Existem muitas outras possibilidades e combinações que você pode explorar para manipular e transformar seus dados de acordo com suas necessidades.

MÓDULO VI

Conclusão

Os métodos de **array** em **JavaScript** são ferramentas poderosas que nos permitem manipular e transformar **arrays** de maneira eficiente e produtiva. Com o conhecimento desses métodos e sua correta aplicação, podemos escrever códigos mais legíveis, eficientes e de fácil manutenção.

MÓDULO VI

Métodos de Arrays que você precisa conhecer

.forEach()

Caso você precise executar algum código para cada elemento do **Array**, executar um **forEach** é muito mais simples do que criar um **for** ou **while**, já que não precisamos declarar variáveis de controle.

```
let myHTML = '<ul>'
const numbersList = [1, 2, 3, 4, 5];

numbersList.forEach((number, index, array) => {
    myHTML += `<li>${number}</li>`;
});
myHTML += '</ul>';
```

MÓDULO VI

O **.forEach()** irá jogar cada um dos elementos do **Array** no primeiro parâmetro, o índice do elemento no segundo e o **Array** original no terceiro. Claro que se não for usar, pode colocar apenas o primeiro e deixar o código assim:

```
let myHTML = '<ul>'
const numbersList = [1, 2, 3, 4, 5];

numbersList.forEach(number => myHTML += `<li>${number}</li>` );
myHTML += '</ul>';
```

MÓDULO VI

.map()

Nós utilizamos o **.map()** quando queremos fazer alguma modificação nos elementos de um **Array**.

```
const usersList = [
    {name: 'João', credit: 500},
    {name: 'Maria', credit: 800}
];

const newUsersList = usersList.map((user, index, array) => {
    user.credit += 100;
    return user;
})
```

MÓDULO VI

No exemplo acima alteramos o valor de uma propriedade de cada um dos elementos do **Array**.
Na última linha nós precisamos executar o comando return para indicar o que será retornado para o **Array**. No final teremos um novo **Array**.

Teremos o seguinte retorno:

```
[
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900}
];
```

MÓDULO VI

Também pode ser interessante se quisermos alterar a estrutura dos objetos do **Array**. Podemos pegar o exemplo de cima: suponha que a gente agora queira um **Array** com o nome de todos os usuários da nossa lista. Teríamos o seguinte código:

```
const newUsersList = usersList.map((user, index, array) => {
    return user.name;
})
/* Resultado:
[ 'João', 'Maria' ]
*/
```

MÓDULO VI

Outro exemplo bem simples, aproveitando a simplicidade das **Arrow Functions**: dobrar o valor dos números de um **Array**:

```
[1, 2, 3, 4, 5].map(number => number * 2);
/* Resultado:
[2, 4, 6, 8, 10]
*/
```

MÓDULO VI

.filter()

Como o próprio nome indica, serve para filtrarmos os elementos de um **Array**.
Passamos para ele uma função. Se essa função retornar **true**, o elemento será inserido no novo **Array** que será criado.
Se a função retornar **false**, o elemento será ignorado.

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const newUsersList = usersList.filter((user, index, array) => user.credit > 500);
/* Resultado:
[
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900}
]
*/
```

MÓDULO VI

.find()

Usamos esse método quando queremos encontrar algum elemento dentro no **Array**.
Para isso, passamos uma função que irá retornar **true** ou **false**.
O primeiro **true** que for retornado irá finalizar a função e retornar o elemento em que estamos.

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const carlos = usersList.find((user, index, array) => user.name === 'Carlos');
/* Resultado:
    {name: 'Carlos', credit: 300}
*/
```

MÓDULO VI

.findIndex()

Faz o mesmo que o **.find()**, mas retorna o índice do elemento encontrado ao invés de retornar o próprio elemento

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const carlos = usersList.findIndex((user, index, array) => user.name === 'Carlos');
/* Resultado:
    2
*/
```

MÓDULO VI

.every()

Serve para testarmos se todos os elementos do **Array** passam em uma condição.

Passamos uma função que retorna **true** ou **false**. Se todos os retornos forem **true**, significa que todos os elementos passaram no teste, e a função retornará **true**.

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const result1 = usersList.every((user, index, array) => user.credit < 1000);
const result2 = usersList.every(user => user.credit < 500);
```

MÓDULO VI

No primeiro nós testamos se todos os usuários possuem crédito menor que **1000**.

Como todos passaram no teste, o resultado de **result1** será **true**

No segundo nós testamos se todos os usuários possuem crédito menor que **500**.

Como não são todos que passam nesse teste, o resultado de **result2** será **false**.

MÓDULO VI

.some()

O **.some()** faz algo parecido com o **.every()**. A diferença é que o **.every()** só retorna **true** se todos os elementos passarem no teste. O **.some()** retorna true se pelo menos um elemento do **Array** passar no teste.

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const result = usersList.some((user, index, array) => user.credit === 300);
```

Verificamos se há pelo menos um usuário com crédito igual a **300**. O resultado será **true**.

MÓDULO VI

.sort()

O **.sort()** serve para ordenar os elementos de **Arrays**. Muitas pessoas utilizam este método de maneira errada, apenas executando **array.sort()**. Isso pode causar retornos inesperados, pois os elementos serão convertidos em texto.

O correto é passar uma função que compare dois elementos. Assim, podemos ordenar um **Array** com qualquer tipo de objeto, ordenando por qualquer propriedade.

MÓDULO VI

```
const numbersList = [4, 5, 7, 8, 2];
const orderedList = numbersList.sort((a, b) => {
    if(a < b){
       return -1;
    } else if(a > b){
        return 1;
    }
    return 0;
})
```

A função de ordenação sempre recebe dois elementos. Se o primeiro for menor, devemos retornar um número menor que **0**. Se o primeiro for maior, devemos retornar um número maior do que **0**. Se forem iguais, retornamos **0**.

MÓDULO VI

Neste caso, poderíamos simplificar a ordenação de números das seguintes maneiras:

```
const orderedList = numbersList.sort((a, b) => a > b ? 1 : -1 );
// ou também:
const orderedList = numbersList.sort((a, b) => a - b )
```

Para ter a ordenação em ordem decrescente, basta inverter o retorno da função.

```
const orderedList = numbersList.sort((a, b) => a > b ? -1 : 1 );
```

MÓDULO VI

Podemos utilizar em elementos mais complexos:

```
const usersList = [
    {name: 'João', credit: 600},
    {name: 'Maria', credit: 900},
    {name: 'Carlos', credit: 300},
    {name: 'Vanessa', credit: 200},
];

const orderedUsers = usersList.sort((a, b) => a.credit - b.credit);
```

Com isso teremos um novo Array com os usuários ordenados pela quantidade de crédito.

MÓDULO VI

.reduce()

Esta função serve para reduzirmos o conteúdo de um Array para apenas um elemento.

O exemplo mais clássico é somar todos os valores de um Array.

```
const numbersList = [1, 2, 3];
const total = numbersList.reduce((total, currentElement) => total + currentElement)
/* Resultado:
    6
*/
```

A função que executamos recebe como primeiro parâmetro uma variável que irá acumular um valor e como segundo parâmetro teremos cada um dos elementos do Array a cada iteração.

MÓDULO VI

EXERCÍCIOS...

MÓDULO VI

FIM...

A Educação é o primeiro passo para um futuro melhor...