APOSTILA DE REVISÃO V1

JAVASCRIPT MODERNO

Módulo 3: Fundamentos de JavaScript – Parte 2

FORMULA
WEB
PRO-MAX

Strict Mode

Em JavaScript, o modo estrito, ou "strict mode", é uma forma de operar seu código em um escopo mais restrito e seguro. Quando você ativa o modo estrito em um script ou função, o interpretador do JavaScript faz algumas mudanças no comportamento padrão, visando aprimorar a qualidade do código e reduzir erros comuns.

Aqui estão algumas das principais características do modo estrito:

1. **Erro por silencioso:** Em modo estrito, certas ações que gerariam apenas avisos em modo normal, agora lançam erros. Isso ajuda a identificar e corrigir problemas de forma mais rápida.
2. **Variáveis não declaradas:** Em modo estrito, atribuir valores a variáveis não declaradas resulta em erro. Em modo normal, isso cria uma variável global implícita.
3. **"this" em funções:** Em funções, o valor de this é undefined quando a função é chamada sem um contexto explícito. Em modo normal, isso seria o objeto global (window no navegador, global no Node.js).

Para ativar o modo estrito, você pode adicionar a declaração 'use strict'; no início de um script ou de uma função. Por exemplo:

```
'use strict';

const nome = ‘Uanela’;
```

O modo estrito é uma prática recomendada, pois ajuda a evitar erros comuns e torna o código mais previsível e seguro.

Funções

Em JavaScript, funções são blocos de código reutilizáveis que podem ser chamados para executar uma tarefa específica. Elas são fundamentais na linguagem e permitem a criação de código modular e organizado. Aqui estão alguns pontos importantes sobre funções em JavaScript:

1. **Declaração de Função:** Você pode declarar uma função usando a palavra-chave function, seguida pelo nome da função e, opcionalmente, uma lista de parâmetros entre parênteses. Por exemplo:

```
function minhaFuncao() {
      // corpo da função
}
```

2. **Chamada de Função:** Para executar o código dentro de uma função, você precisa chamá-la. Isso é feito usando o nome da função seguido por parênteses contendo os argumentos, se houver. Por exemplo:

```
minhaFuncao();
```

3. **Retorno de Valor:** As funções podem retornar um valor usando a palavra-chave return. Isso é útil quando você precisa que a função retorne um resultado para ser usado em outro lugar do seu código. Por exemplo:

```
function soma(a, b) {
    return a + b;
}
const resultado = soma(5, 3); // resultado será 8
```

4. **Funções Anônimas:** Você também pode criar funções sem nome, conhecidas como funções anônimas. Elas são úteis em situações como callbacks e expressões de função. Por exemplo:

```
const minhaFuncao = function() {
    // corpo da função
};
```

5. **Arrow Functions:** As arrow functions são uma forma mais concisa de escrever funções em JavaScript. Elas são especialmente úteis para funções simples. Por exemplo:

```
const soma = (a, b) => a + b;
```

6. **Escopo de Função:** Variáveis declaradas dentro de uma função são locais para essa função e não podem ser acessadas fora dela, a menos que você retorne essas variáveis ou as passe como argumentos para outra função.
7. **Hoisting:** Em JavaScript, as declarações de função são "elevadas" para o topo do contexto de execução. Isso significa que você pode chamar uma função antes mesmo de ela ser declarada no código. No entanto, isso não se aplica a funções declaradas como expressões de função.

Esses são apenas alguns aspectos básicos das funções em JavaScript. Elas são uma parte fundamental da linguagem e são usadas extensivamente para criar código modular e reutilizável.

Introdução a Arrays

Arrays em JavaScript são objetos usados para armazenar múltiplos valores em uma única variável. Eles são usados para armazenar coleções de dados, como números, strings, objetos e até mesmo outras arrays. Aqui estão algumas características importantes sobre arrays em JavaScript:

1. **Declaração de Array:** Você pode declarar uma array de várias maneiras:

```
const array = []; // array vazia

const numeros = [1, 2, 3, 4, 5]; // array de números

const frutas = ['maçã', 'banana', 'laranja']; // array de strings
```

2. **Acesso a Elementos:** Você pode acessar elementos de uma array usando seu índice (começando em 0):

```
const fruta = frutas[0]; // 'maçã'
```

3. **Adição e Remoção de Elementos:** Você pode adicionar e remover elementos de uma array:

```
frutas.push('uva'); // adiciona 'uva' ao final da array

frutas.pop(); // remove o último elemento da array (neste caso, 'uva')
```

4. **Tamanho da Array:** Você pode verificar o tamanho de uma array usando a propriedade length:

```
const tamanho = frutas.length; // 3
```

5. **Métodos Úteis:** Arrays em JavaScript têm muitos métodos úteis, como indexOf, includes, push, unshist, pop, Shift, entre outros, que podem ajudar na manipulação e transformação de arrays.

Essas são apenas algumas das funcionalidades básicas de arrays em JavaScript. Elas são amplamente usadas em programação para lidar com conjuntos de dados e são uma parte fundamental da linguagem.

Introdução a Objectos

Em JavaScript, um objeto é uma coleção de pares chave-valor, onde as chaves são strings (ou símbolos) e os valores podem ser de qualquer tipo de dado, incluindo outros objetos. Os objetos em JavaScript são usados para representar entidades do mundo real, abstrações de dados e muito mais. Aqui está um exemplo simples de como criar e usar objetos em JavaScript:

```
// Criando um objeto pessoa

const pessoa = {

    nome: "João",

    idade: 30,

    profissao: "Desenvolvedor"

};

// Acessando propriedades do objeto

console.log(pessoa.nome); // Saída: João

console.log(pessoa.[‘idade’]]); // Saída: 30

console.log(pessoa.profissao); // Saída: Desenvolvedor

// Alterando uma propriedade do objeto

pessoa.idade = 31;

// Adicionando uma nova propriedade ao objeto

pessoa.cidade = "São Paulo"

// Deletando uma propriedade do objeto

delete pessoa.profissao;
```

Objetos em JavaScript também suportam métodos, que são funções associadas ao objeto. Esses métodos podem ser usados para realizar operações relacionadas ao objeto. Por exemplo:

```
const pessoa = {
    nome: "Maria",
    idade: 25,
    profissao: "Designer",
    saudacao: function() {
        return "Olá, meu nome é " + this.nome + " e tenho " + this.idade + " anos.";
    }
};
console.log(pessoa.saudacao()); // Saída: Olá, meu nome é Maria e tenho 25 anos.
```

Em JavaScript, os objetos são uma parte fundamental da linguagem e são amplamente utilizados para modelar dados e comportamentos complexos.

As principais diferenças entre objetos e arrays em JavaScript

1. **Chave de Acesso:** Em objetos, os valores são acessados por chaves (strings ou símbolos), enquanto em arrays, os valores são acessados por índices numéricos.
2. **Ordenação:** Os objetos não têm uma ordem específica para suas propriedades, enquanto os arrays têm uma ordem definida para seus elementos.
3. **Tipo de Dados:** Os objetos podem ter chaves de diferentes tipos de dados (como strings e símbolos), enquanto os arrays têm índices que são sempre números inteiros.

4. **Uso:** Os objetos são mais adequados quando se deseja associar chaves a valores e quando a ordem dos elementos não é importante. Os arrays são mais adequados quando se deseja uma coleção ordenada de elementos, acessíveis por índices.

Em termos de quando usar cada um:

Objetos: São ideais quando você precisa de uma estrutura de dados flexível, com chaves que descrevem o significado dos valores associados. Por exemplo, um objeto pode representar um usuário com propriedades como nome, idade, e-mail, etc.

Arrays: São úteis quando você precisa de uma coleção ordenada de elementos do mesmo tipo. Por exemplo, uma lista de itens de compra ou uma série de valores que representam dados sequenciais.

Em muitos casos, você pode usar tanto objetos quanto arrays, dependendo das necessidades específicas do seu código. Por exemplo, você pode usar um objeto para representar um item de dados complexo em uma lista, onde as chaves do objeto descrevem as propriedades do item e os valores são os dados reais.

Notação de ponto vs Notação de parênteses recto

Em JavaScript, você pode acessar as propriedades de um objeto de duas maneiras principais: usando a notação de ponto (.) e a notação de colchetes ([]).

1. **Notação de Ponto (.):** Esta é a forma mais comum de acessar propriedades de objetos. Você simplesmente coloca o nome da propriedade após o ponto. Por exemplo:

```
const pessoa = {
    nome: "João",
    idade: 30,
    profissao: "Desenvolvedor"
};
```

```
console.log(pessoa.nome); // Saída: João

console.log(pessoa.idade); // Saída: 30
```

A notação de ponto é mais conveniente e fácil de ler, especialmente quando você sabe o nome da propriedade antecipadamente.

Notação de Parênteses Recto ([]): Esta notação permite que você acesse propriedades de um objeto usando uma expressão, o que pode ser útil quando o nome da propriedade é dinâmico ou armazenado em uma variável. Por exemplo:

```
let pessoa = {
    nome: "João",
    idade: 30,
    profissao: "Desenvolvedor"
};

let propriedade = "nome";

console.log(pessoa[propriedade]); // Saída: João

let outraPropriedade = "idade";

console.log(pessoa[outraPropriedade]); // Saída: 30
```

Você também pode usar a notação de parênteses recto para acessar propriedades com nomes que não são válidos em identificadores JavaScript (por exemplo, propriedades com espaços ou caracteres especiais).

Em geral, a notação de ponto é mais comumente usada, mas a notação de parênteses recto oferece mais flexibilidade em certas situações, como ao acessar propriedades dinamicamente.

Métodos de Objectos

Em JavaScript, os objetos são essencialmente coleções de pares chave-valor, onde as chaves são strings (ou símbolos) e os valores podem ser de qualquer tipo de dado, incluindo funções. Essas funções associadas a objetos são chamadas de métodos. Os métodos em JavaScript são simplesmente propriedades que têm uma função como valor. Aqui está um exemplo simples de um objeto com um método:

```
let pessoa = {
    nome: "João",
    idade: 30,
    saudacao: function() {
        return "Olá, meu nome é " + this.nome + " e tenho " + this.idade + " anos.";
    }
};
console.log(pessoa.saudacao()); // Saída: Olá, meu nome é João e tenho 30 anos.
```

Neste exemplo, saudacao é um método do objeto pessoa. Quando chamamos pessoa.saudacao(), a função associada é executada e retorna uma saudação personalizada com base nas propriedades do objeto.

Os métodos podem ser adicionados a objetos de várias maneiras. Você pode definir diretamente uma função como valor de uma propriedade, como no exemplo acima, ou usar a sintaxe de definição de método abreviada disponível a partir do ES6:

```
let pessoa = {
    nome: "Maria",
    idade: 25,
    saudacao() {
        return "Olá, meu nome é " + this.nome + " e tenho " + this.idade
+ " anos.";
    }
};
console.log(pessoa.saudacao()); // Saída: Olá, meu nome é Maria e tenho
25 anos.
```

Os métodos em objetos são muito úteis para encapsular a lógica relacionada a um objeto específico e são uma parte fundamental da programação orientada a objetos em JavaScript.

Iteração com Loop for

O loop for é uma estrutura de controle fundamental em muitas linguagens de programação, incluindo JavaScript. Ele é usado para repetir um bloco de código várias vezes, com base em uma condição específica. O formato básico de um loop for em JavaScript é o seguinte:

```
for (inicialização; condição; expressão final) {
    // bloco de código a ser repetido
}
```

- A **inicialização** é executada uma vez antes do início do loop. Normalmente, é usada para inicializar um contador ou variável de controle.

- A **condição** é avaliada antes de cada iteração do loop. Se for verdadeira, o bloco de código dentro do loop é executado. Se for falsa, o loop é encerrado.
- A **expressão final** é executada após cada iteração do loop. Normalmente, é usada para atualizar o contador ou variável de controle.

Por exemplo, o seguinte código imprime os números de 1 a 5 no console:

```
for (let i = 0; i <= 5; i++) {
    console.log(i);
}
```

Este loop for tem uma inicialização (let i = 0;), uma condição (i <= 5;) e uma expressão final (i++). Ele inicializa i como 0, executa o bloco de código (que imprime o valor de i no console), e então incrementa i em 1 após cada iteração, repetindo até que i seja maior que 5.

Iterando Arrays

Iterar arrays significa percorrer cada elemento do array, um por um, geralmente para realizar alguma operação em cada elemento. Em JavaScript, existem várias formas de iterar sobre um array. Abaixo você tem como iterar usando o laço for:

O loop for é uma maneira tradicional de iterar sobre um array. Você pode usar um índice para acessar cada elemento do array.

```
const frutas = [“banana”, “manga”, “uva”, “laranja”, “pera”];
for (let i = 0; i < frutas.length; i++) {
    console.log(frutas[i]);
}
```

Em loops nós podemos ainda interromper o seu fluxo normal atráves do uso das palavras-chaves **continue** e **break**, que são palavras-chave em JavaScript usadas para controlar o fluxo de loops, especialmente loops for. Aqui está como elas funcionam em loops for:

- **continue**: A palavra-chave continue é usada para pular a iteração atual do loop e continuar com a próxima iteração. Isso significa que o código dentro do loop após a palavra-chave continue não será executado para a iteração atual, mas o loop continuará com a próxima iteração.

  Exemplo:

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue; // Pula a iteração quando i for igual a 2
    }
    console.log(i); // Será executado para i de 0 a 4, exceto quando
i for igual a 2
}
```

  Neste exemplo, o número 2 será pulado e não será impresso no console.

- **break**: A palavra-chave break é usada para interromper o loop imediatamente semelhante ao return em uma função. Quando o JavaScript encontra a palavra-chave break, ele sai do loop e continua com a execução do código após o loop.

  Exemplo:

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        break; // Interrompe o loop quando i for igual a 2
    }
    console.log(i); // Será executado apenas para i de 0 a 1
}
```

  Neste exemplo, o loop é interrompido quando i é igual a 2, então apenas os números de 0 a 1 são impressos no console.

Ambas as palavras-chave continue e break são úteis para controlar o fluxo de execução em loops for e podem ser usadas para evitar a execução de código desnecessário ou interromper a execução do loop quando uma condição específica for atendida.

Loop Reverso

Loop reverso é uma técnica usada para iterar sobre uma coleção de elementos, como um array, de trás para frente. Em JavaScript, podemos usar um loop for para isso, começando do último elemento até o primeiro. Aqui está um exemplo simples:

```
const frutas = ['banana', 'maçã', 'laranja', 'uva', 'morango'];
for (let i = frutas.length - 1; i >= 0; i--) {
    console.log(frutas[i]);
}
```

Neste exemplo, começamos com i igual ao último índice do array (frutas.length - 1) e decrementamos i em cada iteração. O loop continua enquanto i for maior ou igual a 0, e acessamos cada elemento do array usando frutas[i], imprimindo-os no console.

O loop reverso é útil em situações em que precisamos iterar sobre uma coleção em ordem inversa, por exemplo, para processar uma lista de mensagens em uma caixa de entrada de e-mail da mais recente para a mais antiga.

É importante notar que, ao usar um loop reverso, devemos ter cuidado com o acesso a elementos do array, especialmente ao remover elementos, pois os índices mudam a cada iteração.

Loops Aninhados

Loops dentro de loops, também conhecidos como loops aninhados, são uma técnica comum em programação para lidar com estruturas de dados multidimensionais, como matrizes ou tabelas. Isso envolve usar um loop dentro de outro loop para iterar sobre os elementos de uma coleção de maneira mais complexa. Aqui está um exemplo simples em JavaScript de como isso pode ser feito:

```
for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
        console.log(`i: ${i}, j: ${j}`);
    }
}
```

Neste exemplo, temos um loop for externo que itera sobre i de 0 a 2 e um loop for interno que itera sobre j de 0 a 2 para cada valor de i. Isso resultará em uma saída que mostra todas as combinações possíveis de i e j.

Os loops aninhados são frequentemente usados em situações em que precisamos percorrer uma matriz bidimensional ou multidimensional. Por exemplo, para percorrer uma matriz 2D e imprimir cada elemento, podemos fazer o seguinte:

```
const matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
for (let i = 0; i < matriz.length; i++) {
    for (let j = 0; j < matriz[i].length; j++) {
        console.log(matriz[i][j]);
    }
}
```

Este exemplo mostra como podemos usar loops aninhados para acessar cada elemento de uma matriz 2D e realizar operações com eles. É importante ter cuidado ao usar loops aninhados, pois eles podem facilmente levar a iterações excessivas e a um código difícil de entender.

Loop While e Do While

O laço while é uma estrutura de controle de fluxo em JavaScript (e em muitas outras linguagens de programação) que permite executar um bloco de código repetidamente enquanto uma condição específica for verdadeira. A sintaxe básica do while é a seguinte:

```
while (condição) {
    // bloco de código a ser executado enquanto a condição for verdadeira
}
```

O bloco de código dentro do while é repetidamente executado enquanto a condição especificada dentro dos parênteses for avaliada como verdadeira. É importante garantir que a condição possa ser falsa em algum momento, caso contrário, o loop se tornará um loop infinito.

Aqui está um exemplo simples de como usar um loop while para contar de 1 a 5:

```
let contador = 1;
while (contador <= 5) {
    console.log(contador);
    contador++;
}
```

Neste exemplo, o loop while é executado enquanto o valor da variável contador for menor ou igual a 5. A cada iteração do loop, o valor de contador é impresso no console e depois incrementado em 1. O loop termina quando contador atinge o valor de 6 e a condição contador <= 5 se torna falsa.

É importante ter cuidado ao usar loops while para evitar loops infinitos, pois eles podem causar o travamento do programa. Certifique-se sempre de que a condição de parada seja alcançada em algum momento durante a execução do loop.

Laço do while

O laço do...while é semelhante ao while, mas com uma diferença fundamental: ele executa o bloco de código pelo menos uma vez, mesmo que a condição seja falsa na primeira iteração. A estrutura básica do do...while é a seguinte:

```
do {
    // bloco de código a ser executado
} while (condição);
```

O bloco de código dentro do do é executado primeiro e depois a condição é verificada. Se a condição for verdadeira, o bloco de código é executado novamente e o processo se repete. Se a condição for falsa, o loop é encerrado.

A principal diferença entre o do...while e o while é que o do...while garante que o bloco de código seja executado pelo menos uma vez, mesmo que a condição seja falsa desde o início.

Aqui está um exemplo simples de como usar um loop do...while para contar de 1 a 5:

```
let contador = 1;
do {
    console.log(contador);
    contador++;
} while (contador <= 5);
```

Neste exemplo, o bloco de código dentro do do é executado uma vez, mesmo que a condição contador <= 5 seja falsa na primeira iteração. Depois disso, a condição é verificada e o bloco de código é repetidamente executado enquanto a condição for verdadeira.