

Apostila do curso

JavaScript para não especialistas em tecnologia: Uma introdução acessível

Celso Y. Ishida

Apresentação

Olá! Bem-vindo ao mundo mágico do JavaScript!

Se você não é da área de tecnologia, talvez esteja curioso sobre essa tal de "JavaScript" que tanto se ouve falar. Quem sabe você ouviu dizer que ela está por trás daquela animação legal em um site, daquele botão que faz algo diferente quando você clica, ou até mesmo de aplicativos que você usa no seu celular. E você está certo! O JavaScript é uma linguagem poderosa e muito presente no nosso dia a dia digital.

A primeira coisa que queremos que você saiba é: não precisa ter medo! Sei que o mundo da tecnologia pode parecer complicado e cheio de termos estranhos, mas este livro foi feito pensando em você, que talvez nunca tenha programado antes. Vamos juntos desmistificar o JavaScript e descobrir como ele pode ser mais simples e divertido do que você imagina.

Imagine o JavaScript como a varinha mágica da internet. Enquanto o HTML e o CSS são como os tijolos e a decoração de uma casa, dando forma e cor, o JavaScript é o que traz vida a essa casa. Ele permite que as coisas se movam, interajam com você e façam coisas dinâmicas. Sabe quando uma foto aparece ao passar o mouse? Ou quando um mapa se mexe na tela? Ou até mesmo quando um jogo online funciona no seu navegador? Tudo isso, e muito mais, pode ter o toque do JavaScript.

Nestas páginas, vamos dar os primeiros passos juntos. Vamos aprender o vocabulário básico, entender como dar "ordens" ao computador usando JavaScript e ver como essas ordens podem transformar uma página da web comum em algo muito mais interessante e interativo.

Este livro, revisado com auxílio de IA, resume os conceitos principais da linguagem e é complementado com os exercícios do curso de JavaScript ministrado pelo professor Celso Yoshikazu Ishida e os vídeos indicados pelos QrCode. Os exercícios aparecem na ordem crescente de aprendizagem. Alguns exercícios são recomendados a serem feitos com o IDE + IA. Existem algumas recomendadas, veja a explicação: <http://200.17.205.92/siteprototipo/curso/javascript/qrij001.html>



Não se preocupe se algumas palavras parecerem novas agora. Explicaremos tudo com calma e usando exemplos do mundo real para que você possa entender facilmente. O importante é ter curiosidade e vontade de aprender.

Então, prepare-se para essa jornada! Vamos explorar juntos o universo do JavaScript e descobrir o poder que essa linguagem pode colocar nas suas mãos. Esta apostila deve ser lida em paralelo ao curso. Quem sabe, ao final deste, você não só entenderá o que é JavaScript, mas também será capaz de criar suas próprias "mágicas" na web!

Vamos nessa?

ÍNDICE

Sumário

Apresentação.....	2
Tópicos importantes	5
Orientações para aproveitar a apostila	6
Orientações sobre o código Javascript	6
Exercícios de visão geral	7
Tipos, Variáveis e Operadores	8
Operadores: Símbolos para operações com valores.	9
Estruturas de Controle.....	10
Tomada de Decisões (Condicionais):	10
Repetição (Loops)	11
Controle de Loop	12
Arrays (Vetores)	13
Criação de Arrays	13
Acessando Elementos:.....	13
Propriedade length:	13
Métodos Comuns de Arrays.....	13
Iterando sobre Arrays.....	14
Objetos	15
Criação de Objetos	15
Acessando Propriedades	16
Adicionando e Modificando Propriedades	16
Métodos (Funções dentro de Objetos).....	16
Iterando sobre Propriedades	17
DOM (Document Object Model)	17
Eventos	19
Funções.....	24
Definição de Funções.....	24
Chamada de Funções (Invoking).....	24
Parâmetros e Argumentos.....	24
Retorno de Valores	25
Escopo de Função	25
BOM (Browser Object Model)	25
Armazenamento no Navegador.....	28
localStorage	28
sessionStorage	28

Cookies	29
O LocalStorage é uma boa prática?	30
Requisições HTTP	31
Exemplo (básico de requisição GET)	31
Fetch API	31
Exemplo (requisição POST com envio de dados JSON)	32
Google Analytics	33
Desafio Final	34

Tópicos importantes

Para construir uma base sólida em JavaScript, alguns tópicos são realmente essenciais. Pensando em um programador JavaScript, desde o iniciante até o mais experiente, estes são os pilares que considero mais importantes. Focando apenas no JavaScript puro, sem considerar bibliotecas como React JS ou Node.js, os conceitos essenciais que todo programador JavaScript deve conhecer são:

1. [Tipos de Dados](#): Entender os diferentes tipos de valores que o JavaScript pode manipular (números, textos, verdadeiro/falso, etc.) e como eles funcionam.
2. Variáveis: Como guardar e dar nomes a esses valores para usá-los depois no seu código. Aprender a escolher os tipos certos de "caixas" para guardar as informações.
3. Operadores: Os símbolos que usamos para fazer operações com esses valores, como somar, comparar, verificar condições, etc.
4. [Estruturas de Controle](#): As ferramentas que permitem que o seu código tome decisões ("se isso acontecer, faça aquilo") e repita ações ("faça isso várias vezes").
5. Funções: Blocos de código reutilizáveis que realizam tarefas específicas. Aprender a criar e usar funções para organizar e simplificar o código.
6. [Arrays \(Vetores\)](#): Formas de guardar coleções de itens em uma única variável, como listas. Aprender a acessar, adicionar e remover itens dessas listas.
7. [Objetos](#): Formas de agrupar informações relacionadas sob um único nome, usando pares de "nome" e "valor".

JavaScript no Navegador (Front-end):

8. [DOM \(Document Object Model\)](#): A representação da estrutura de uma página da web como uma árvore. Aprender como o JavaScript pode "enxergar" e modificar essa estrutura, o conteúdo e os estilos da página.
9. [Eventos](#): As ações que acontecem no navegador (cliques, movimentos do mouse, etc.). Aprender como o JavaScript pode "ouvir" esses eventos e reagir a eles, tornando a página interativa.
10. [BOM \(Browser Object Model\)](#): As ferramentas que o navegador oferece para interagir com a janela do navegador, o histórico de navegação, a localização da página, etc.
11. [Armazenamento no Navegador](#): As formas que o navegador oferece para guardar informações no computador do usuário para que elas fiquem disponíveis mesmo depois de fechar a página (como lembrar de preferências ou informações temporárias).
12. [Requisições HTTP \(AJAX / Fetch API\)](#): Como o JavaScript pode se comunicar com servidores na internet para buscar ou enviar informações sem precisar recarregar a página inteira.

Conceitos Avançados (Importantes para o Crescimento em JavaScript Puro):

13. Programação Assíncrona: Como lidar com tarefas que levam tempo para serem concluídas (como buscar informações na internet) sem que a página fique travada, usando mecanismos como Promises e Async/Await.
14. Closures: Um conceito que permite que funções "lembrem" do ambiente em que foram criadas, mesmo depois de serem executadas.

15. Orientação a Objetos (OO): Uma forma de organizar o código usando "objetos" que contêm dados e funcionalidades relacionadas. Aprender sobre classes e protótipos em JavaScript.
16. Programação Funcional: Uma forma de programar focada em usar funções como blocos de construção principais, evitando efeitos colaterais e favorecendo a imutabilidade.

Esses são os pilares do JavaScript puro. Dominando esses conceitos, você terá uma base sólida para construir interatividade e funcionalidades complexas diretamente no navegador.

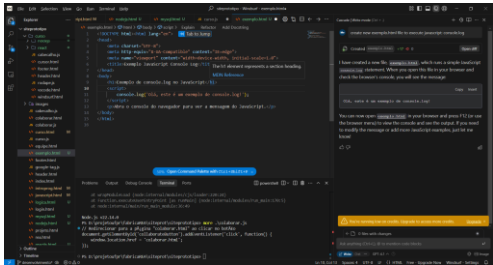

Orientações para aproveitar a apostila

Esta apostila deve ser utilizada em conjunto com a explicação do professor, leitura da parte escrita desta apostila e fazer os exercícios recomendados no curso. Os exercícios estão agrupados por assuntos e a sua ordem foi pensado para ser a introdução gradual dos assuntos. As respostas destes exercícios estão indicadas no final de cada grupo de assunto.

Sempre que ver um qrcode, será uma página com vídeo interativo que complementa o conteúdo escrito fornecendo mais informações como, por exemplo, citação dos softwares recomendados para programação web e dicas de instalação.



<http://200.17.205.92/siteprototipo/curso/javascript/qrij002.html>

Para escrever o código utilize IDE	Veja o código interpretado no navegador
	

Orientações sobre o código Javascript

Os exemplos de código JavaScript estarão com uma cor de fundo diferente e muitas vezes aparecerá apenas o código javascript. Por exemplo, a seguir, o conteúdo do arquivo `exercicio01.js`:

```
console.log('Esta mensagem aparecerá no console do navegador');
```

```
// comentários aparecem depois de duas barras
console.log( 'Os comentários são ignorados pelo compilador de JavaScript');
/* quando precisar comentar mais de uma linha,
use barra e asterisco no início e asterisco e barra no final */
```

Mas códigos JavaScript podem ser encontrados dentro de páginas HTML como o exemplo a seguir. O código a seguir é o conteúdo do arquivo `exemploinicial.html`, onde o único comando Javascript é a linha com o `console.log`

```
<!DOCTYPE html><html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exemplo JavaScript Console Log</title>
</head>
<body>
  <h1>Exemplo de console.log no JavaScript</h1>
  <script>
    console.log('Olá, este é um exemplo de console.log!');
  </script>
  <p>Abra o console do navegador F12 para ver a mensagem do JavaScript.</p>
</body>
</html>
```

Ou podem estar num arquivo separado, exemplo `exercicio01.js`: mas a chamada deste arquivo ainda é dentro do código HTML. Como o exemplo a seguir:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exercício 1 de JavaScript - Título que aparece na aba do
navegador</title>
</head>
<body>
  <h1>Exercício 1 de JavaScript - código em arquivo externo</h1>
  <script src="exercicio01.js"></script>
  <footer>CYI</footer>
</body>
</html>
```

Exercícios de visão geral

Agora, vamos fazer os exercícios passando pelos principais assuntos de Javascript. O objetivo neste momento é ter uma visão geral da linguagem e sentir como é a programação utilizando IA. Vamos utilizar IA para ajudar a programar e, também, utilizar para aprender os comandos, sua sintaxe e entender o que foi gerado. Mãos à obra 😊

<http://200.17.205.92/siteprototipo/curso/javascript/qrij003.html>



Tipos, Variáveis e Operadores

Vamos imaginar que você deseja guardar o nome, idade de mensagem de uma pessoa. Em qualquer linguagem de programação você precisa de um lugar para cada uma destas informações. Este lugar, ou espaço reservado na memória, chamamos de variável.

No JavaScript, você pode criar variáveis como o exemplo a seguir.

```
let sobrenome = "Ishida"; // variável do tipo string
let idade = 20; // variável do tipo número
let mensagem = 'Navegando e sempre programando!';
let numeros = [10, 30, 40]; // variável do tipo array (vetor)
let conjunto = { sobrenome: "Silva", idade: 30}; // tipo objeto
```

Analizando a primeira linha, cria-se uma variável com o nome 'sobrenome' e ao mesmo tempo armazena (atribui) o valor 'Ishida'. Note que cada tipo tem o seu formato para atribuição. Se for do tipo numérico é preciso só dos números. Já as variáveis do tipo string aceita qualquer tipo de valor e devem estar entre aspas simples (exemplo na terceira linha) ou duplo (exemplo na primeira linha).

Existem alguns tipos mais complexos como os array (exemplo na terceira linha) e objetos (quinta linha). Os arrays aceitam diversos valores separados por vírgulas e agrupados entre colchetes. Já os objetos podem conter um conjunto de variáveis de qualquer tipo.

Você também criar, ou declarar, as variáveis com um único comando *let*, separado por vírgulas. No exemplo a seguir vemos as declarações das variáveis e as atribuições separadas.

```
let sobrenome, idade, mensagem;
sobrenome = "Ishida";
idade = 20;
mensagem = 'Navegando e sempre programando!';
let numeros, conjunto;
numeros = [10, 30, 40];
conjunto = {nome: "Silva", idade: 30};
```

Ou podemos fazer tudo numa única linha, como, por exemplo:

```
let sobrenome = "Ishida", idade = 20, mensagem = 'Outra!', numeros = [10, 30, 40],
conjunto = {nome: "Silva", idade: 30};
```

Em outras palavras, podemos escrever os comandos de diversas formas, para tal, é preciso saber a sintaxe da linguagem. A sintaxe é como os comandos devem ser escritos. É possível encontrar a sintaxe dos comandos na internet, por exemplo, para buscar sobre o comando *let*, dentro do VS Code deixe o mouse sobre o comando *let* (ou qualquer outro comando) e espere aparecer um diálogo de ajuda que aparecerá bem acima do mouse, ou abaixo, com informações da sintaxe e link para informações mais detalhadas.

As variáveis são úteis porque permitem que você armazene e manipule dados de forma dinâmica enquanto seu programa está sendo executado. Isso é essencial para criar programas interativos e funcionais. Os valores das variáveis podem variar de acordo com os comandos no programa.

Veja, os seguintes comandos. Obs: todos os comandos são executados na ordem e os valores das variáveis são definidos em tempo de execução (a medida que o programa é executado).

```
let contador = 1
console.log( 'Valor de contador: ' + contador)
contador = contador + 1
console.log( 'Valor de contador: ' + contador)
contador++
console.log( 'Valor de contador: ' + contador)
contador += 1
console.log( 'Valor de contador: ' + contador)
```

As linhas 3, 5 e 7 produzem o mesmo efeito prático. A linha 3 soma o valor atual da variável contador e soma o valor 1 e depois o resultado atribui para a variável contador. Já a linha 5 adiciona um ao valor da variável. E a linha 7 irá pegar o valor da variável e somar 1. O resultado deste código será:

```
Valor de contador: 1
Valor de contador: 2
Valor de contador: 3
Valor de contador: 4
```

Além do comando let, existe o comando const para declaração de variável. O comando const declara a variável e obrigatoriamente atribui um valor e este valor não pode ser modificado.

```
// Usando let (variável que pode ser reatribuída)
let contador = 0;
contador = contador + 1;

// Usando const (variável constante, não pode ser reatribuída)
const gravidade = 9.8;
// gravidade = 10; // Isso geraria um erro

// Usando var (escopo de função, menos recomendado em código moderno)
var mensagemVar = "Usando var";
mensagemVar += ' para variáveis globais '
```

Operadores: Símbolos para operações com valores.

A seguir exemplos de operadores e explicação nos comentários

```
let num1 = 10;
let num2 = 5;
```

```

// Operadores Aritméticos
let soma = num1 + num2; // 15
let subtracao = num1 - num2; // 5
let multiplicacao = num1 * num2; // 50
let divisao = num1 / num2; // 2
let resto = num1 % num2; // 0

// Operadores de Atribuição
let valor = 10;
valor += 5; // valor agora é 15
valor -= 2; // valor agora é 13

// Operadores de Comparação (retornam booleanos)
let igual = (num1 == num2); // false (comparação de valor)
let igualEstrito = (num1 === "10"); // false (comparação de valor e tipo)
let maiorQue = (num1 > num2); // true

// Operadores Lógicos
let condicao1 = true;
let condicao2 = false;
let e = (condicao1 && condicao2); // false (AND)
let ou = (condicao1 || condicao2); // true (OR)
let nao = (!condicao1); // false (NOT)

```

Estruturas de Controle

Estruturas de controle são ferramentas essenciais para dar lógica e dinamismo ao seu código JavaScript. Elas permitem que o programa tome decisões com base em condições e repita blocos de código várias vezes.

Tomada de Decisões (Condicionais):

- **if:** Executa um bloco de código se uma condição for verdadeira.
JavaScript

```

let idade = 18;
if (idade >= 18) {
  console.log("Você é maior de idade.");
}

```

- **if...else:** Executa um bloco se a condição for verdadeira e outro bloco se for falsa.
JavaScript

```

let temperatura = 15;
if (temperatura > 20) {

```

```

    console.log("Está quente!");
  } else {
    console.log("Está fresco.");
  }
}

```

- if...else if...else: Permite verificar múltiplas condições em sequência.
JavaScript

```

let nota = 7;
if (nota >= 9) {
  console.log("Excelente!");
} else if (nota >= 7) {
  console.log("Bom.");
} else if (nota >= 5) {
  console.log("Regular.");
} else {
  console.log("Reprovado.");
}

```

- switch: Avalia uma expressão e executa um bloco de código correspondente ao valor da expressão.
JavaScript

```

let diaSemana = 3;
switch (diaSemana) {
  case 1:
    console.log("Domingo");
    break;
  case 2:
    console.log("Segunda");
    break;
  case 3:
    console.log("Terça");
    break;
  default:
    console.log("Outro dia");
}

```

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrij004.html>



Repetição (Loops)

- for: Executa um bloco de código um número específico de vezes.
JavaScript

```

for (let i = 0; i < 5; i++) {
  console.log("Iteração número: " + i);
}

```

- while: Executa um bloco de código enquanto uma condição for verdadeira.
JavaScript

```

let contador = 0;
while (contador < 3) {
  console.log("Contador: " + contador);
  contador++;
}

```

- do...while: Similar ao while, mas garante que o bloco de código seja executado pelo menos uma vez.
JavaScript

```
let i = 0;
do {
  console.log("Valor de i: " + i);
  i++;
} while (i < 2);
```

- for...of: Itera sobre os valores de objetos iteráveis (como arrays e strings).
JavaScript

```
let frutas = ["Maçã", "Banana", "Laranja"];
for (let fruta of frutas) {
  console.log(fruta);
}
```

```
let texto = "JavaScript";
for (let letra of texto) {
  console.log(letra);
}
```

- for...in: Itera sobre os nomes das propriedades enumeráveis de um objeto.
JavaScript

```
let pessoa = { nome: "Ana", idade: 25 };
for (let propriedade in pessoa) {
  console.log(propriedade + ": " + pessoa[propriedade]);
}
```

Controle de Loop

- break: Sai imediatamente de um loop (for, while, do...while ou switch).
JavaScript

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i); // Imprime 0 a 4
}
```

- continue: Pula a iteração atual de um loop e passa para a próxima.
JavaScript

```
for (let i = 0; i < 5; i++) {
  if (i === 2) {
    continue;
  }
  console.log(i); // Imprime 0, 1, 3, 4 (pula o 2)
}
```

Dominar as estruturas de controle é essencial para escrever código JavaScript que execute diferentes ações sob diferentes condições e automatize tarefas repetitivas.

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/grj005.html>



Arrays (Vetores)

Arrays são estruturas de dados fundamentais em JavaScript para armazenar coleções ordenadas de itens. Esses itens podem ser de qualquer tipo (números, strings, booleanos, objetos, outros arrays, etc.).

Criação de Arrays

- **Array Literal:** A forma mais comum de criar um array, usando colchetes `[]` e separando os elementos por vírgulas.

JavaScript

```
let frutas = ["Maçã", "Banana", "Laranja"];
let numeros = [10, 20, 30, 40, 50];
let misturado = [1, "dois", true, null, { chave: "valor" }];
let vazio = [];
```

- **Construtor `Array()`:** Embora menos comum, também é possível criar arrays usando o construtor `Array()`.

JavaScript

```
let cores = new Array("Vermelho", "Verde", "Azul");
let outrosNumeros = new Array(5); // Cria um array com 5 posições vazias
```

Acessando Elementos:

Os elementos de um array são acessados usando seu índice, que começa em 0 para o primeiro elemento.

JavaScript

```
let frutas = ["Maçã", "Banana", "Laranja"];
console.log(frutas[0]); // Saída: Maçã
console.log(frutas[1]); // Saída: Banana
console.log(frutas[2]); // Saída: Laranja
```

Propriedade `length`:

A propriedade `length` retorna o número de elementos em um array.

JavaScript

```
let numeros = [10, 20, 30];
console.log(numeros.length); // Saída: 3
```

Métodos Comuns de Arrays

JavaScript oferece diversos métodos poderosos para manipular arrays:

- **`push()`:** Adiciona um ou mais elementos ao final do array e retorna o novo comprimento.

JavaScript

```
let frutas = ["Maçã", "Banana"];
frutas.push("Morango");
console.log(frutas); // Saída: ["Maçã", "Banana", "Morango"]
```

- **`pop()`:** Remove o último elemento do array e retorna esse elemento.

JavaScript

```
let frutas = ["Maçã", "Banana", "Morango"];
let ultimo = frutas.pop();
console.log(frutas); // Saída: ["Maçã", "Banana"]
```

```
console.log(ultimo); // Saída: Morango
```

- **unshift():** Adiciona um ou mais elementos no início do array e retorna o novo comprimento.
JavaScript

```
let cores = ["Verde", "Azul"];  
cores.unshift("Vermelho");  
console.log(cores); // Saída: ["Vermelho", "Verde", "Azul"]
```

- **shift():** Remove o primeiro elemento do array e retorna esse elemento.
JavaScript

```
let cores = ["Vermelho", "Verde", "Azul"];  
let primeiro = cores.shift();  
console.log(cores); // Saída: ["Verde", "Azul"]  
console.log(primeiro); // Saída: Vermelho
```

- **indexOf(elemento):** Retorna o primeiro índice em que o elemento pode ser encontrado no array (-1 se não estiver presente).
JavaScript

```
let animais = ["Cachorro", "Gato", "Pássaro", "Gato"];  
console.log(animais.indexOf("Gato")); // Saída: 1  
console.log(animais.indexOf("Peixe")); // Saída: -1
```

- **lastIndexOf(elemento):** Retorna o último índice em que o elemento pode ser encontrado no array (-1 se não estiver presente).
JavaScript

```
let animais = ["Cachorro", "Gato", "Pássaro", "Gato"];  
console.log(animais.lastIndexOf("Gato")); // Saída: 3
```

- **slice(inicio, fim):** Retorna uma cópia superficial de uma parte do array, começando pelo índice inicio e terminando no índice fim (não inclusivo). Se omitido, copia todo o array.
JavaScript

```
let numeros = [1, 2, 3, 4, 5];  
let parte1 = numeros.slice(1, 3); // [2, 3]  
let copia = numeros.slice(); // [1, 2, 3, 4, 5]
```

- **splice(inicio, quantosRemover, ...itensAdicionar):** Altera o conteúdo de um array removendo ou substituindo elementos existentes e/ou adicionando novos elementos no local.
JavaScript

```
let frutas = ["Maçã", "Banana", "Laranja"];  
frutas.splice(1, 1, "Manga"); // Remove "Banana" e adiciona "Manga"  
console.log(frutas); // Saída: ["Maçã", "Manga", "Laranja"]
```

```
frutas.splice(2, 0, "Kiwi", "Abacaxi"); // Adiciona "Kiwi" e "Abacaxi" na posição 2  
console.log(frutas); // Saída: ["Maçã", "Manga", "Kiwi", "Abacaxi", "Laranja"]
```

Iterando sobre Arrays

Existem várias maneiras de percorrer os elementos de um array:

- **for loop:**
JavaScript

```
let cores = ["Vermelho", "Verde", "Azul"];
```

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

- for...of loop (ES6+): Mais conciso para iterar sobre os valores.
JavaScript

```
let frutas = ["Maçã", "Banana", "Laranja"];  
for (let fruta of frutas) {  
  console.log(fruta);  
}
```

- Métodos de iteração (forEach, map, filter, reduce, etc.): Funções de ordem superior que permitem realizar operações em cada elemento do array.
JavaScript

```
let numeros = [1, 2, 3];  
numeros.forEach(function(numero) {  
  console.log(numero * 2);  
});
```

Em resumo: Arrays são listas ordenadas de itens que podem ser acessados por seu índice. JavaScript oferece uma rica variedade de métodos para adicionar, remover, encontrar, extrair e manipular elementos em arrays, tornando-os uma ferramenta essencial para trabalhar com coleções de dados.

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrij006.html>



Objetos

Objetos em JavaScript são coleções de pares chave-valor, onde cada chave (geralmente uma string) está associada a um valor (que pode ser qualquer tipo de dado JavaScript: número, string, booleano, outro objeto, array, função, etc.). Eles permitem agrupar informações relacionadas sob um único nome, representando entidades do mundo real de forma estruturada.

Criação de Objetos

- Objeto Literal: A forma mais comum de criar um objeto, usando chaves {} e definindo os pares chave-valor dentro delas.
JavaScript

```
let pessoa = {  
  nome: "Ana",  
  idade: 30,  
  cidade: "São Paulo"  
};
```

```
let carro = {  
  marca: "Ford",  
  modelo: "Ka",  
  ano: 2020,  
  cor: "Vermelho"  
};
```

```
let vazio = {};
```

Acessando Propriedades

Existem duas maneiras principais de acessar o valor de uma propriedade de um objeto:

- Notação de Ponto (.): Usada quando o nome da propriedade é um identificador válido (sem espaços ou caracteres especiais).

JavaScript

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

```
console.log(carro.modelo); // Saída: Ka
```

- Notação de Colchetes ([]): Usada quando o nome da propriedade é uma string (especialmente com espaços ou caracteres especiais) ou quando o nome da propriedade é armazenado em uma variável.

JavaScript

```
console.log(pessoa["cidade"]); // Saída: São Paulo
```

```
let propriedade = "ano";
```

```
console.log(carro[propriedade]); // Saída: 2020
```

Adicionando e Modificando Propriedades

Novas propriedades podem ser adicionadas a um objeto existente simplesmente atribuindo um valor a uma nova chave. Os valores de propriedades existentes podem ser modificados da mesma forma.

JavaScript

```
let livro = {  
  titulo: "O Pequeno Príncipe",  
  autor: "Antoine de Saint-Exupéry"  
};
```

```
livro.ano = 1943; // Adicionando uma nova propriedade
```

```
livro.titulo = "Pequeno Príncipe"; // Modificando o valor de uma propriedade existente
```

```
console.log(livro);
```

```
// Saída: { titulo: 'Pequeno Príncipe', autor: 'Antoine de Saint-Exupéry', ano: 1943 }
```

Métodos (Funções dentro de Objetos)

Objetos podem conter funções como valores de suas propriedades. Essas funções são chamadas de métodos e geralmente operam nos dados do próprio objeto usando a palavra-chave this.

JavaScript

```
let aluno = {  
  nome: "Pedro",  
  nota: 8.5,  
  dizerOla: function() {  
    console.log(`Olá, meu nome é ${this.nome}.`);  
  },  
  mostrarNota: function() {  
    console.log(`Minha nota é ${this.nota}.`);  
  }  
};
```



```
aluno.dizerOla(); // Saída: Olá, meu nome é Pedro.  
aluno.mostrarNota(); // Saída: Minha nota é 8.5.
```

Iterando sobre Propriedades

O loop `for...in` é comumente usado para iterar sobre os nomes das propriedades enumeráveis de um objeto.

JavaScript

```
let computador = {  
  processador: "Intel i7",  
  memoria: "16GB",  
  armazenamento: "1TB SSD"  
};  
  
for (let chave in computador) {  
  console.log(`${chave}: ${computador[chave]}`);  
}  
// Saída (a ordem pode variar):  
// processador: Intel i7  
// memoria: 16GB  
// armazenamento: 1TB SSD
```

Em resumo: Objetos são estruturas chave-valor flexíveis que permitem organizar dados relacionados e comportamentos (métodos) em uma única entidade. Eles são fundamentais para modelar dados complexos e construir aplicações JavaScript estruturadas.

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrij007.html>



DOM (Document Object Model)

O DOM (Document Object Model) é uma interface de programação para documentos HTML e XML. Ele representa a estrutura do documento como uma árvore de objetos, onde cada elemento HTML (como tags, atributos e texto) se torna um nó nessa árvore. O DOM permite que linguagens de script como JavaScript acessem e manipulem a estrutura, o estilo e o conteúdo do documento web de forma dinâmica.

Conceitos Chave:

- **Árvore de Nós:** O DOM organiza o documento HTML em uma hierarquia de nós. O nó raiz é o objeto `document`, que representa todo o documento HTML. Dentro dele, encontramos nós para elementos (`<html>`, `<head>`, `<body>`, etc.), atributos (como `id`, `class`, `src`) e texto.
- **Objetos:** Cada nó na árvore do DOM é um objeto com propriedades e métodos que permitem interagir com ele. Por exemplo, um elemento HTML (`<p>`, `<div>`) é representado por um objeto `Element` que possui propriedades como `textContent` (para acessar ou modificar o texto interno) e métodos como `addEventListener` (para adicionar um ouvinte de eventos).
- **Acesso e Manipulação:** JavaScript usa o objeto global `document` e seus métodos para selecionar elementos específicos no DOM (por exemplo, por ID, classe, tag) e, em seguida, manipular suas propriedades, estilos, atributos e até mesmo a estrutura da árvore (adicionando ou removendo nós).

Exemplo:

Imagine o seguinte trecho de HTML:

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>Exemplo DOM</title>
</head>
<body>
  <h1 id="titulo-principal">Olá, Mundo!</h1>
  <p class="paragrafo">Este é um parágrafo.</p>
  <button id="meu-botao">Clique Aqui</button>

  <script>
    // Selecionando elementos do DOM
    const titulo = document.getElementById('titulo-principal');
    const paragrafo = document.querySelector('.paragrafo');
    const botao = document.getElementById('meu-botao');

    // Manipulando o conteúdo
    titulo.textContent = 'Título Alterado com JavaScript!';
    paragrafo.innerHTML = 'Este parágrafo agora tem <strong>texto em negrito</strong>.';

    // Manipulando o estilo
    botao.style.backgroundColor = 'lightblue';

    // Adicionando um ouvinte de eventos
    botao.addEventListener('click', function() {
      alert('Botão foi clicado!');
    });
  </script>
</body>
</html>
```

Explicação do Exemplo:

1. Seleção de Elementos:

- `document.getElementById('titulo-principal')` seleciona o elemento `<h1>` com o ID "titulo-principal".
- `document.querySelector('.paragrafo')` seleciona o primeiro elemento com a classe "paragrafo" (usando sintaxe de seletor CSS).
- `document.getElementById('meu-botao')` seleciona o elemento `<button>` com o ID "meu-botao".

2. Manipulação de Conteúdo:

- `titulo.textContent = 'Título Alterado com JavaScript!'`; altera o texto dentro do elemento `<h1>`.
- `paragrafo.innerHTML = 'Este parágrafo agora tem texto em negrito.'`; altera o HTML interno do elemento `<p>`, permitindo adicionar tags HTML.

3. Manipulação de Estilo:

- `botao.style.backgroundColor = 'lightblue'`; altera a cor de fundo do botão diretamente através da propriedade `style`.

4. Adição de Ouvinte de Eventos:

- `botao.addEventListener('click', function() { ... });` adiciona uma função (callback) que será executada quando o evento de "click" ocorrer no botão. Neste caso, um alerta será exibido.

Em resumo: O DOM é a ponte que permite ao JavaScript interagir com a estrutura e o conteúdo de uma página web. Ao representar o HTML como uma árvore de objetos, ele oferece as ferramentas necessárias para criar páginas dinâmicas e interativas, respondendo às ações do usuário e modificando o conteúdo em tempo real.

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/grj008.html>



Eventos

Eventos em JavaScript são ações ou ocorrências que acontecem no navegador (ou dentro da página web) e que o JavaScript pode "ouvir" e reagir a elas. Essas ações podem ser iniciadas pelo usuário (clique em um botão, mover o mouse, digitar no teclado), pelo próprio navegador (a página terminar de carregar, ocorrer um erro), ou por outras interações.

Conceitos Chave:

- Ouvintes de Eventos (Event Listeners): São mecanismos que permitem que você especifique qual código JavaScript deve ser executado quando um evento específico acontece em um elemento HTML específico.
- Tipos de Eventos: Existem diversos tipos de eventos, cada um representando uma ação diferente. Alguns exemplos comuns incluem:
 - Mouse Events: click, mouseover, mouseout, mousedown, mouseup, mousemove.
 - Keyboard Events: keydown, keypress,keyup.
 - Form Events: submit, change, focus, blur, input.
 - Document/Window Events: load, DOMContentLoaded, resize, scroll.
- Objeto de Evento (Event Object): Quando um evento ocorre, um objeto Event é criado automaticamente e passado para a função de callback do ouvinte de eventos. Esse objeto contém informações detalhadas sobre o evento que aconteceu (por exemplo, qual elemento foi alvo do evento, as coordenadas do mouse, a tecla pressionada).
- Manipuladores de Eventos (Event Handlers): São as funções JavaScript que são executadas quando um evento específico acontece no elemento ao qual o ouvinte de eventos está anexado.

Exemplo:

Imagine o seguinte trecho de HTML:

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>Exemplo de Evento</title>
</head>
<body>
  <button id="meuBotao">Clique em Mim</button>
  <p id="mensagem"></p>

  <script>
    // Seleciona os elementos
    const botao = document.getElementById('meuBotao');
    const mensagemParagrafo = document.getElementById('mensagem');

    // Define a função que será executada quando o botão for clicado
```

```

function aoClicarBotao() {
    mensagemParagrafo.textContent = 'O botão foi clicado!';
    botao.style.backgroundColor = 'lightgreen';
}

// Adiciona um "ouvinte de eventos" ao botão
// Especifica o tipo de evento ('click') e a função a ser executada (aoClicarBotao)
botao.addEventListener('click', aoClicarBotao);

// Outro exemplo: ouvindo o evento de passar o mouse sobre o botão
botao.addEventListener('mouseover', function() {
    botao.textContent = 'Passei o mouse!';
});

// Ouvindo o evento de retirar o mouse do botão
botao.addEventListener('mouseout', function() {
    botao.textContent = 'Clique em Mim';
    botao.style.backgroundColor = ""; // Remove a cor de fundo
});
</script>
</body>
</html>

```

Explicação do Exemplo:

1. Seleção de Elementos: O código JavaScript primeiro seleciona o botão e o parágrafo usando seus IDs.
2. Definição do Manipulador de Eventos: A função `aoClicarBotao()` define o que acontecerá quando o botão for clicado: o texto do parágrafo é alterado e a cor de fundo do botão muda.
3. Adição de Ouvintes de Eventos:
 - o `botao.addEventListener('click', aoClicarBotao);` anexa um ouvinte de eventos ao botão. Ele "ouve" o evento de click e, quando esse evento ocorre, a função `aoClicarBotao()` é executada.
 - o Os outros `addEventListener` demonstram como ouvir diferentes tipos de eventos (`mouseover` e `mouseout`) e executar funções diferentes em resposta a eles.

Em resumo: Eventos são a maneira fundamental de tornar as páginas web interativas. Ao adicionar ouvintes de eventos aos elementos HTML, você pode especificar o código JavaScript que deve ser executado em resposta a diversas ações do usuário ou do navegador, permitindo criar interfaces dinâmicas e responsivas.

Criar uma tabela exaustiva com *todos* os eventos existentes em JavaScript seria bastante extenso, pois a especificação web é dinâmica e novos eventos podem ser adicionados. Além disso, o suporte a alguns eventos pode variar entre navegadores.

Por isso, a seguir uma tabela com os eventos mais comuns e importantes, agrupados por categoria, juntamente com um exemplo conciso para cada um.

Categoria	Evento	Descrição	Exemplo em JavaScript
Mouse Events	click	Ocorre quando um elemento é clicado.	<code><button onclick="alert('Botão Clicado!')">Clique</button></code>

	mouseover	Ocorre quando o cursor do mouse entra na área de um elemento.	<div onmouseover="this.style.backgroundColor='yellow'">Passe Aqui</div>
	mouseout	Ocorre quando o cursor do mouse sai da área de um elemento.	<div onmouseout="this.style.backgroundColor='white'">Saia Daqui</div>
	mousedown	Ocorre quando um botão do mouse é pressionado sobre um elemento.	<div onmousedown="console.log('Botão Pressionado')">Pressione</div>
	mouseup	Ocorre quando um botão do mouse é liberado sobre um elemento.	<div onmouseup="console.log('Botão Solto')">Solte</div>
	mousemove	Ocorre quando o cursor do mouse é movido enquanto está sobre um elemento.	<div onmousemove="console.log('Mouse Movendo')">Mova Aqui</div>
Keyboard Events	keydown	Ocorre quando uma tecla é pressionada.	<input onkeydown="console.log('Tecla Pressionada')">
	keyup	Ocorre quando uma tecla é liberada.	<input onkeyup="console.log('Tecla Liberada')">

	keypress	Ocorre quando uma tecla que produz um caractere é pressionada (obsoleto).	<code><input onkeypress="console.log('Caractere Digitado')"></code>
Form Events	submit	Ocorre quando um formulário é submetido.	<code><form onsubmit="alert('Formulário Enviado!')"><button type="submit">Enviar</button></form></code>
	change	Ocorre quando o valor de um elemento de formulário muda.	<code><input type="checkbox" onchange="console.log('Checkbox Alterado')"></code>
	input	Ocorre quando o valor de um elemento <code><input></code> , <code><textarea></code> ou <code><select></code> muda.	<code><input type="text" oninput="console.log('Valor Input Mudou')"></code>
	focus	Ocorre quando um elemento ganha o foco.	<code><input type="text" onfocus="this.style.borderColor='blue'"></code>
	blur	Ocorre quando um elemento perde o foco.	<code><input type="text" onblur="this.style.borderColor='gray'"></code>
Document/Window Events	load	Ocorre quando a página inteira (incluindo recursos) termina de carregar.	<code><body onload="console.log('Página Carregada')"></code>
	DOMContentLoaded	Ocorre quando o HTML inicial é	<code><script>document.addEventListener('DOMContentLoaded', () => console.log('DOM Pronto'));</script></code>

		completamente carregado e parseado.	
	resize	Ocorre quando a janela do navegador é redimensionada.	<code><body onresize="console.log('Janela Redimensionada')"></code>
	scroll	Ocorre quando a barra de rolagem de um elemento é movida.	<code><div style="height: 200px; overflow: auto;" onscroll="console.log('Scrollando')">...</div></code>

Observações Importantes:

- `onclick` e outros atributos `on...`: Os exemplos na tabela usam atributos HTML `on...` para simplificar. A abordagem mais moderna e recomendada é usar `addEventListener()` em JavaScript para separar a lógica do HTML.
- Objeto de Evento: Em manipuladores de eventos JavaScript (especialmente com `addEventListener()`), você recebe um objeto `event` como argumento, que contém informações detalhadas sobre o evento (ex: `event.target` para o elemento que disparou o evento, `event.clientX/clientY` para coordenadas do mouse, `event.key` para a tecla pressionada).
- Fases do Evento: Os eventos no DOM possuem fases de "captura" e "propagação" (bubbling). Por padrão, os listeners são executados na fase de bubbling.
- Eventos de Formulário Adicionais: Existem outros eventos relacionados a formulários, como `reset`.
- Eventos de Mídia: Para elementos `<audio>` e `<video>`, existem eventos como `play`, `pause`, `ended`, `volumechange`, etc.
- Eventos de Drag and Drop: Eventos como `dragstart`, `dragover`, `drop`.
- Eventos de Touch: Para dispositivos com tela sensível ao toque, existem eventos como `touchstart`, `touchmove`, `touchend`.
- Eventos de Animação e Transição CSS: Eventos como `animationstart`, `animationend`, `transitionend`.

Esta tabela oferece uma visão geral dos eventos mais comuns. Para uma lista completa e detalhada, você pode consultar a documentação oficial da W3C e do MDN Web Docs. Lembre-se que a forma como você lida com esses eventos usando `addEventListener()` em JavaScript é a prática mais recomendada para um código mais limpo e estruturado.

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrj009.html>



Funções

Funções são blocos de código reutilizáveis que realizam tarefas específicas. Elas permitem organizar o código, torná-lo mais legível e evitar repetições. Funções podem receber entradas (parâmetros), executar tarefas e, opcionalmente, retornar um resultado.

Definição de Funções

- **Declaração de Função (Function Declaration):** Usa a palavra-chave `function`, seguida pelo nome da função, parênteses (para parâmetros) e um bloco de código entre chaves¹ `{}`.
JavaScript

```
function saudar(nome) {  
  console.log("Olá, " + nome + "!");  
}
```

- **Expressão de Função (Function Expression):** Atribui uma função (geralmente anônima) a uma variável.
JavaScript

```
const dobrar = function(numero) {  
  return numero * 2;  
};
```

- **Arrow Function (ES6+):** Uma sintaxe mais concisa para definir funções.
JavaScript

```
const multiplicarPorTres = (numero) => numero * 3;  
  
// Com múltiplos parâmetros ou corpo mais complexo:  
const somar = (a, b) => {  
  const resultado = a + b;  
  return resultado;  
};
```

Chamada de Funções (Invoking)

Para executar o código dentro de uma função, você precisa chamá-la usando seu nome seguido de parênteses, passando os argumentos (valores para os parâmetros) se a função os definir.

JavaScript

```
saudar("Maria"); // Saída: Olá, Maria!  
let resultadoDobro = dobrar(5);  
console.log(resultadoDobro); // Saída: 10  
let resultadoMultiplicacao = multiplicarPorTres(7);  
console.log(resultadoMultiplicacao); // Saída: 21  
let somaResultado = somar(2, 8);  
console.log(somaResultado); // Saída: 10
```

Parâmetros e Argumentos

- **Parâmetros:** Variáveis listadas na definição da função que recebem valores quando a função é chamada.

- Argumentos: Os valores reais passados para a função durante a chamada.

JavaScript

```
function apresentar(nome, idade) {  
  console.log("Meu nome é " + nome + " e tenho " + idade + " anos.");  
}
```

apresentar("Carlos", 28); // "Carlos" é o argumento para o parâmetro 'nome', e 28 para 'idade'.

Retorno de Valores

A palavra-chave `return` dentro de uma função especifica o valor que a função deve retornar ao código que a chamou. Se `return` não for usado explicitamente, a função retorna `undefined`.

JavaScript

```
function calcularAreaRetangulo(base, altura) {  
  const area = base * altura;  
  return area;  
}
```

```
let areaRet = calcularAreaRetangulo(10, 5);  
console.log("A área do retângulo é: " + areaRet); // Saída: A área do retângulo é: 50
```

Escopo de Função

As variáveis declaradas dentro de uma função (com `let` ou `const`) têm escopo local, o que significa que só podem ser acessadas dentro dessa função.

JavaScript

```
function exemploEscopo() {  
  let mensagemLocal = "Esta é uma mensagem local.";  
  console.log(mensagemLocal);  
}
```

```
exemploEscopo(); // Saída: Esta é uma mensagem local.  
// console.log(mensagemLocal); // Isso geraria um erro, pois mensagemLocal não está definida fora da função.
```

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrij010.html>



BOM (Browser Object Model)

O BOM (Browser Object Model) é um conjunto de objetos fornecidos pelos navegadores web que permitem interagir com a janela do navegador e seu ambiente. Ao contrário do DOM, que se concentra no conteúdo do documento HTML, o BOM lida com aspectos como a história de navegação, a localização (URL), a tela do usuário, os temporizadores e as caixas de diálogo.

O BOM permite que o JavaScript interaja com a janela do navegador em si, fornecendo funcionalidades para navegação, informações sobre o navegador e a tela, temporizadores e

interação básica com o usuário através de caixas de diálogo. É essencial para criar aplicações web mais ricas e conscientes do ambiente do navegador.

Objetos Principais do BOM:

- **window**: O objeto global que representa a janela do navegador. É o objeto de nível superior e contém todas as outras propriedades e métodos do BOM. Você geralmente não precisa escrever `window` explicitamente (ex: `alert()` é o mesmo que `window.alert()`).
- **navigator**: Contém informações sobre o navegador do usuário (nome, versão, sistema operacional, etc.).
- **location**: Contém informações sobre a URL atual da página e métodos para redirecionar o navegador.
- **history**: Permite manipular o histórico de navegação do navegador (voltar, avançar).
- **screen**: Contém informações sobre a tela do usuário (largura, altura, resolução).
- **document**: Embora tecnicamente a raiz do DOM, `document` também é uma propriedade do objeto `window` e serve como ponto de entrada para interagir com o conteúdo HTML.
- `setTimeout()` e `setInterval()`: Funções para executar código com atraso ou em intervalos repetidos.
- `alert()`, `confirm()`, `prompt()`: Funções para exibir caixas de diálogo interativas.

Exemplos:

JavaScript

```
<!DOCTYPE html>
<html>
<head>
  <title>Exemplo BOM</title>
</head>
<body>
  <button onclick="redirecionar()">Ir para Google</button>
  <button onclick="voltarPagina()">Voltar</button>
  <button onclick="mostrarLarguraJanela()">Mostrar Largura</button>
  <div id="info"></div>

  <script>
    const infoDiv = document.getElementById('info');

    // Objeto window
    console.log("Largura interna da janela:", window.innerWidth);
    console.log("Altura interna da janela:", window.innerHeight);

    // Objeto navigator
    console.log("User Agent:", navigator.userAgent);

    // Objeto location
    function redirecionar() {
      window.location.href = "https://www.google.com";
    }
    console.log("URL atual:", window.location.href);

    // Objeto history
    function voltarPagina() {
      window.history.back();
    }
    console.log("Histórico de navegação (tamanho):", window.history.length);

    // Objeto screen
    console.log("Largura da tela:", window.screen.width);
```

```

console.log("Altura da tela:", window.screen.height);

// Temporizadores (setTimeout)
setTimeout(function() {
  infoDiv.textContent = "Mensagem exibida após 2 segundos.";
}, 2000);

// Caixas de diálogo
function confirmarAcao() {
  if (window.confirm("Você tem certeza?")) {
    alert("Ação confirmada!");
  } else {
    alert("Ação cancelada.");
  }
}

function perguntarNome() {
  const nome = window.prompt("Digite seu nome:");
  if (nome) {
    alert("Olá, " + nome + "!");
  } else {
    alert("Nenhum nome digitado.");
  }
}
</script>

<button onclick="confirmarAcao()">Confirmar</button>
<button onclick="perguntarNome()">Digite seu Nome</button>
</body>
</html>

```

Explicação do Exemplo:

- `window.innerWidth` e `window.innerHeight`: Obtêm as dimensões internas da janela do navegador.
- `navigator.userAgent`: Fornece uma string com informações sobre o navegador do usuário.
- `window.location.href`: Define ou obtém a URL da página atual. A função `redirecionar()` demonstra como mudar a URL para navegar para outra página.
- `window.history.back()` e `window.history.length`: Permitem voltar na história de navegação e obter o número de entradas no histórico.
- `window.screen.width` e `window.screen.height`: Fornecem as dimensões da tela do usuário.
- `setTimeout()`: Exibe uma mensagem após um atraso de 2000 milissegundos (2 segundos).
- `window.confirm()`: Exibe uma caixa de diálogo com botões "OK" e "Cancelar", retornando um booleano.
- `window.prompt()`: Exibe uma caixa de diálogo que solicita entrada do usuário e retorna o texto digitado (ou null se cancelado).

Agora, vamos para os exercícios?

<http://200.17.205.92/siteprototipo/curso/javascript/qrj011.html>



Armazenamento no Navegador

O armazenamento no navegador permite que as aplicações web persistam dados localmente no navegador do usuário, melhorando a experiência e permitindo funcionalidades offline. As principais formas de armazenamento são:

localStorage

- **Persistência:** Os dados permanecem armazenados mesmo após o fechamento do navegador e reabertura.
- **Escopo:** Limitado à origem (protocolo, domínio e porta). Diferentes sites não podem acessar os dados uns dos outros.
- **Capacidade:** Geralmente em torno de 5-10MB por origem (varia entre navegadores).
- **Síncrono:** As operações de leitura e escrita bloqueiam a thread principal do navegador, o que pode impactar a performance para grandes operações.
- **Uso:** Ideal para preferências do usuário, dados não sensíveis que precisam persistir.

```
// Salvar dados
localStorage.setItem('tema', 'escuro');
localStorage.setItem('usuario', JSON.stringify({ nome: 'João', idade: 30 }));

// Recuperar dados
const temaSalvo = localStorage.getItem('tema');
const usuarioSalvoString = localStorage.getItem('usuario');
const usuarioSalvo = JSON.parse(usuarioSalvoString);

// Remover dados
localStorage.removeItem('tema');
localStorage.clear(); // Remove todos os dados da origem
```

sessionStorage

- **Persistência:** Os dados são armazenados apenas durante a sessão atual do navegador (enquanto a aba ou janela estiver aberta). São perdidos ao fechar a aba/janela.
- **Escopo:** Limitado à origem e à aba/janela.
- **Capacidade:** Similar ao localStorage, mas a persistência é menor.
- **Síncrono:** As operações também são síncronas.
- **Uso:** Útil para dados temporários relacionados à sessão do usuário, como o estado de um formulário ou itens em um carrinho de compras antes do envio.

```
JavaScript

// Salvar dados de sessão
sessionStorage.setItem('carrinho', JSON.stringify(['produtoA', 'produtoB']));

// Recuperar dados de sessão
const carrinhoSalvoString = sessionStorage.getItem('carrinho');
const carrinhoSalvo = JSON.parse(carrinhoSalvoString);

// Remover dados de sessão
sessionStorage.removeItem('carrinho');
sessionStorage.clear(); // Remove todos os dados da sessão atual
```

Cookies

- Persistência: Podem ser persistentes (com uma data de expiração) ou de sessão (removidos ao fechar o navegador).
- Escopo: Associados a um domínio e podem ter um caminho específico. Podem ser acessados pelo servidor (via cabeçalhos HTTP) e pelo cliente (via JavaScript).
- Capacidade: Limitada (geralmente em torno de 4KB por cookie).
- Síncrono: A manipulação via JavaScript é síncrona.
- Uso: Tradicionalmente usados para rastreamento de usuários, gerenciamento de sessões (com identificadores), preferências e outras informações que precisam ser compartilhadas com o servidor.

JavaScript

```
// Definir um cookie
document.cookie = "usuariold=123; expires=Fri, 24 Apr 2026 23:59:59 GMT; path=/";
document.cookie = "tema=claro; path=/";

// Ler todos os cookies (retorna uma string com todos os cookies separados por '; ')
const todosCookies = document.cookie;

// Função para obter o valor de um cookie específico
function getCookie(nome) {
  const nomeEQ = nome + "=";
  const ca = document.cookie.split(';');
  for(let i = 0; i < ca.length; i++) {
    let c = ca[i];
    while (c.charAt(0) === ' ') c = c.substring(1, c.length);
    if (c.indexOf(nomeEQ) === 0) return c.substring(nomeEQ.length, c.length);
  }
  return null;
}

const usuariold = getCookie('usuariold');
const temaCookie = getCookie('tema');

// Excluir um cookie (definindo a data de expiração para um tempo passado)
document.cookie = "usuariold=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";
```

Outras opções (mais avançadas):

- IndexedDB: Um sistema de banco de dados NoSQL transacional para armazenamento estruturado de grandes quantidades de dados no navegador. Assíncrono.
- Web SQL (Deprecated): Uma API para bancos de dados relacionais usando SQL no navegador. Não é mais recomendado para uso.
- Cache API: Parte dos Service Workers, permite armazenar respostas de requisições HTTP para uso offline e melhor performance.

Em resumo: O armazenamento no navegador oferece diferentes maneiras de persistir dados localmente. `localStorage` é para armazenamento persistente de dados não sensíveis, `sessionStorage` para dados temporários da sessão e cookies para informações compartilhadas com o servidor. A escolha da tecnologia de armazenamento depende dos requisitos de persistência, capacidade, segurança e complexidade dos dados da sua aplicação.

Agora, vamos para os exercícios?
<http://200.17.205.92/siteprototipo/curso/javascript/qrij012.html>



O localStorage é uma boa prática?

O uso do `localStorage` pode ser uma boa prática dependendo do contexto e de como ele é utilizado. Ele oferece uma maneira conveniente e simples de armazenar dados no navegador do usuário de forma persistente entre sessões. No entanto, é crucial estar ciente de suas limitações e potenciais desvantagens.

Quando o `localStorage` é geralmente uma boa prática:

- Armazenamento de preferências do usuário: Salvar configurações como temas, idioma, preferências de layout, etc., para melhorar a experiência do usuário em visitas futuras.
- Persistência de dados simples: Armazenar pequenos volumes de dados não sensíveis que precisam persistir mesmo após o fechamento do navegador (por exemplo, o progresso em um jogo offline simples, itens em um carrinho de compras antes do checkout).
- Melhoria de performance: Em alguns casos, armazenar dados estáticos ou resultados de chamadas de API no `localStorage` pode reduzir a necessidade de recarregá-los, melhorando a velocidade de carregamento da página.
- Funcionalidades offline básicas: Para PWAs (Progressive Web Apps), o `localStorage` pode armazenar dados essenciais para funcionalidades offline limitadas.

Quando o `localStorage` pode NÃO ser uma boa prática ou requer cautela:

- Armazenamento de informações sensíveis: Nunca armazene informações confidenciais como senhas, tokens de autenticação (prefira `sessionStorage` ou mecanismos mais seguros), dados bancários ou informações de saúde no `localStorage`. Os dados não são criptografados e podem ser acessados por qualquer JavaScript na mesma origem (protocolo, domínio e porta).
- Grandes volumes de dados: O `localStorage` tem limites de armazenamento (geralmente em torno de 5-10MB por origem, mas varia entre navegadores). Tentar armazenar grandes quantidades de dados pode levar a problemas de performance e exceder a cota, causando erros. Para grandes volumes, considere `IndexedDB`.
- Dados transacionais ou temporários: Para dados que só precisam existir durante a sessão atual do usuário, o `sessionStorage` é mais apropriado, pois é limpo quando a aba ou janela é fechada.
- Performance em operações frequentes: Embora a leitura seja geralmente rápida, operações frequentes de escrita no `localStorage` podem impactar a performance da aplicação, pois são síncronas e podem bloquear a thread principal do navegador.
- Acessibilidade e segurança: Usuários podem inspecionar e modificar os dados armazenados no `localStorage` através das ferramentas de desenvolvedor do navegador. Isso reforça a importância de não armazenar informações sensíveis.

Em resumo:

O `localStorage` é uma ferramenta útil para casos de uso específicos, principalmente para armazenar pequenas quantidades de dados não sensíveis que precisam persistir. No entanto, é crucial entender suas limitações de segurança, capacidade e performance, e escolher a solução de armazenamento apropriada para cada cenário. Para dados sensíveis ou grandes volumes, outras opções como `sessionStorage` (para dados de sessão), `IndexedDB` (para armazenamento estruturado maior) ou cookies (para informações trocadas com o servidor) podem ser mais adequadas.

Sempre avalie cuidadosamente os requisitos de segurança, a quantidade de dados, a necessidade de persistência e o impacto na performance antes de decidir usar o `localStorage`.

Requisições HTTP

Requisições HTTP são a espinha dorsal da comunicação entre o navegador web (cliente) e os servidores web (backend). Elas permitem que as páginas web busquem dados dinamicamente, enviem informações e interajam com APIs sem a necessidade de recarregar a página inteira. JavaScript oferece duas maneiras principais de realizar essas requisições: a tradicional `XMLHttpRequest` (AJAX) e a mais moderna `Fetch API`.

1. `XMLHttpRequest` (AJAX - Asynchronous JavaScript and XML):

- **Histórico:** Foi a primeira maneira amplamente utilizada para realizar requisições HTTP assíncronas em JavaScript.
- **Complexidade:** A sintaxe pode ser um pouco verbosa e baseada em eventos.
- **Funcionalidades:** Suporta vários métodos HTTP (GET, POST, PUT, DELETE, etc.), envio de dados, tratamento de respostas, progresso de upload/download.

Exemplo (básico de requisição GET)

JavaScript

```
const xhr = new XMLHttpRequest();
const url = 'https://api.example.com/data';

xhr.open('GET', url);

xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status < 300) {
    console.log('Dados recebidos:', xhr.responseText);
    // Processar a resposta (geralmente JSON)
  } else {
    console.error('Erro na requisição:', xhr.status, xhr.statusText);
  }
};

xhr.onerror = function() {
  console.error('Erro de rede.');
```

Fetch API

- **Moderno:** Introduzida para fornecer uma interface mais poderosa e flexível para realizar requisições HTTP.
- **Simplicidade:** Utiliza Promises, tornando o código assíncrono mais fácil de escrever e gerenciar com `then()` e `catch()`.
- **Poderoso:** Oferece mais controle sobre a requisição e a resposta, incluindo headers, corpo, modo (cors, no-cors, same-origin), cache, etc.
- **Retorno:** A função `fetch()` retorna uma Promise que resolve para um objeto `Response` (mesmo para erros HTTP). Você precisa chamar métodos como `.json()`, `.text()`, `.blob()` na resposta para obter os dados reais.

Exemplo (básico de requisição GET):

JavaScript

```
const url = 'https://api.example.com/data';

fetch(url)
```

```

.then(response => {
  if (!response.ok) {
    throw new Error(`Erro HTTP! status: ${response.status}`);
  }
  return response.json(); // Ou response.text(), response.blob(), etc.
})
.then(data => {
  console.log('Dados recebidos:', data);
  // Processar os dados
})
.catch(error => {
  console.error('Erro na requisição:', error);
});

```

Exemplo (requisição POST com envio de dados JSON)

JavaScript

```

const url = 'https://api.example.com/submit';
const data = { nome: 'Usuário', email: 'usuario@example.com' };

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(response => {
  if (!response.ok) {
    throw new Error(`Erro HTTP! status: ${response.status}`);
  }
  return response.json();
})
.then(responseData => {
  console.log('Resposta do servidor:', responseData);
})
.catch(error => {
  console.error('Erro ao enviar dados:', error);
});

```

Conceitos Chave:

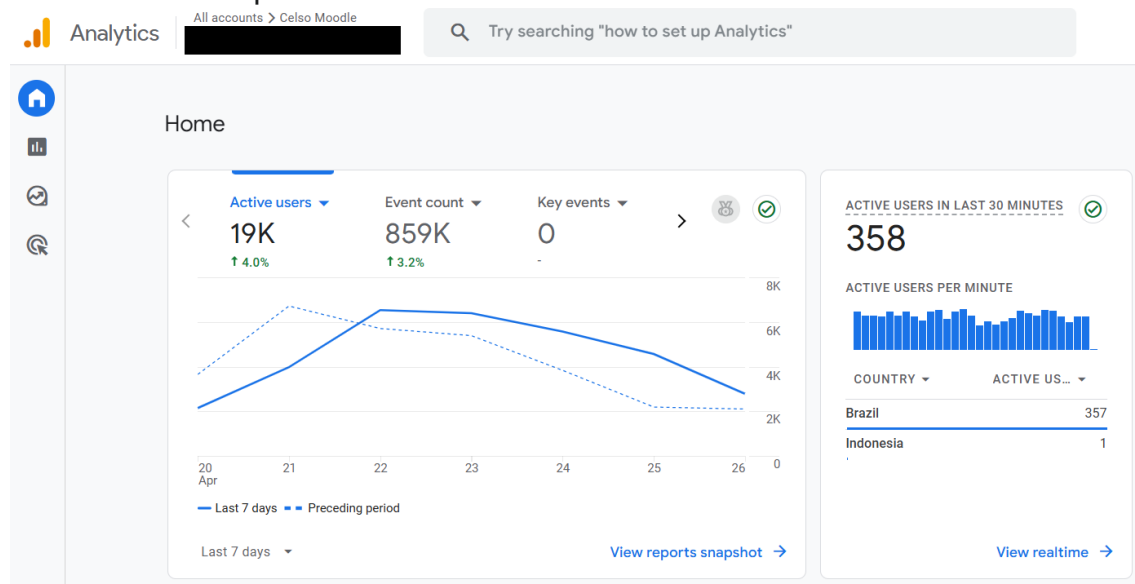
- Métodos HTTP: GET (obter dados), POST (enviar dados para criar), PUT/PATCH (enviar dados para atualizar), DELETE (remover).
- URL (Uniform Resource Locator): O endereço do recurso no servidor.
- Headers: Metadados sobre a requisição ou a resposta (ex: Content-Type, Authorization).
- Corpo (Body): Os dados que são enviados com a requisição (para métodos como POST e PUT).
- Status Codes: Códigos numéricos que indicam o resultado da requisição (ex: 200 OK, 404 Not Found, 500 Internal Server Error).
- Promises (na Fetch API): Representam o resultado eventual de uma operação assíncrona.
- JSON (JavaScript Object Notation): Um formato de dados leve e comum para troca de informações na web.

Em resumo: As requisições HTTP (AJAX e Fetch API) são essenciais para criar aplicações web dinâmicas que interagem com servidores. A Fetch API é a abordagem mais moderna e recomendada devido à sua sintaxe mais limpa e maior flexibilidade, utilizando Promises para lidar com a natureza assíncrona das operações de rede. Ambas permitem que o JavaScript

busque e envie dados sem recarregar a página, proporcionando uma experiência de usuário mais fluida.

Google Analytics

Um uso muito comum do Javascript é um código para que o Google possa te ajudar a fazer estatísticas do seu site. Você coloca o código em todas as páginas e o Google organiza a estatística. Exemplo de estatísticas:



Os passos são simples:

1. Crie uma conta no Google Analytics: Primeiro, você precisa criar uma conta no Google Analytics.
2. Obtenha o código de rastreamento: Após criar a conta e configurar uma propriedade para seu site, o Google Analytics fornecerá um código de rastreamento.
3. Adicione o código ao seu site: Copie o código de rastreamento e cole-o no <head> do seu arquivo HTML. Isso garante que o código seja carregado em todas as páginas do seu site.

Aqui está um exemplo de como o código pode parecer:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Meu Site</title>
  <!-- Código de rastreamento do Google Analytics -->
  <script async src="https://www.googletagmanager.com/gtag/js?id=UA-XXXXXXXXX-
X"></script>
```

```
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
  gtag('config', 'UA-XXXXXXXXX-X');
</script>
</head>
<body>
  <!-- Conteúdo do seu site -->
</body>
</html>
```

Substitua UA-XXXXXXXXX-X pelo seu ID de rastreamento fornecido pelo Google Analytics.

Desafio Final

Vamos para o desafio final para finalizar este curso?

<http://200.17.205.92/siteprototipo/curso/javascript/qrj013.html>

