Fundamentos de JavaScript

com Kristopher Mazzini

Sumário

Supervisão editorial: Thiago Coutinho de Oliveira

Apresentação do curso: Kristopher Mazzini

Produção de conteúdo: Bartolomeu Henrique Lopes

Kristopher Mazzini tem 24 anos e é formado em Engenharia Elétrica pela Universidade Federal de Juiz de Fora, tendo feito parte da graduação na Wrocław University of Science and Technology (Polônia). Começou a atuar profissionalmente com engenharia de software antes mesmo de se formar. Desde o início da sua jornada profissional, já atuou desenvolvendo softwares para diversas empresas dos setores industrial, automobilístico, bancário, logístico, entre outros. Graças a toda essa experiência, hoje atua como líder técnico de equipes.

Módulo 1
Conceitos básicos do JavaScript

Lição 1: Objetivos do módulo

Neste primeiro módulo estudaremos os fundamentos do JavaScript, serão apresentados os tipos de dados, variáveis e funções. Além disso, mostraremos os tipos de conversões e operadores.

Responderemos perguntas do tipo:

- ✓ O que é o JavaScript?
- ✓ Quais são os tipos de variáveis e como declará-las?
- ✓ Como escrever funções em JavaScript?

Lição 2: O que é JavaScript?

JavaScript é uma linguagem de programação de **alto nível** que é amplamente usada para desenvolver aplicativos da web interativos. É uma linguagem de script orientada a objetos que pode ser executada nos navegadores web para **criar e controlar o comportamento dinâmico de uma página da web**.

O JavaScript pode ser usado para realizar tarefas como validação de formulários, animações, manipulação de DOM (Modelo de Objeto do Documento), criação de efeitos visuais e interatividade em geral. Além disso, o JavaScript também pode ser usado do lado do servidor, usando plataformas como o Node.js. O JavaScript é uma das linguagens de programação mais populares do mundo e é amplamente utilizado na indústria de tecnologia.

Há várias vantagens em aprender JavaScript, incluindo:

- ✓ **Ampla utilização:** o JavaScript é uma das linguagens de programação mais populares do mundo, amplamente utilizado em desenvolvimento web, incluindo front-end e back-end, além de ser a principal linguagem para desenvolvimento de aplicações híbridas e nativas de dispositivos móveis.
- ✓ **Facilidade de aprendizado:** o JavaScript é relativamente fácil de aprender, pois tem uma sintaxe simples e é uma linguagem de programação orientada a objetos. Além disso, há uma grande quantidade de recursos e tutoriais disponíveis on-line para ajudar na aprendizagem.
- ✓ **Interatividade:** o JavaScript permite criar páginas web dinâmicas e interativas, o que aumenta o engajamento do usuário e aprimora a experiência do usuário.
- ✓ **Amplas bibliotecas e frameworks:** existem muitas bibliotecas e frameworks JavaScript disponíveis, como React, Angular, Vue.js, Node.js e muitas outras, que facilitam a criação de aplicativos da web de alta qualidade.

- ✓ **Versatilidade:** o JavaScript pode ser usado para desenvolver vários tipos de aplicativos, desde aplicações da web até aplicativos móveis e desktop.

Lição 3: Características do JavaScript e o interpretador NodeJS

Características do JavaScript

Algumas das características da linguagem JavaScript que a tornam uma das linguagens de programação mais populares para o desenvolvimento de aplicativos da web são:

- ✓ **Linguagem interpretada:** o JavaScript é uma linguagem interpretada, o que significa que o código é executado diretamente no navegador, sem a necessidade de ser compilado.
- ✓ **Linguagem de script:** o JavaScript é uma linguagem de script, o que significa que é projetada para processar pequenas tarefas e interações com o usuário em uma página da web.
- ✓ **Dinamicamente tipada:** o JavaScript é dinamicamente tipada, o que significa que o tipo de dados das variáveis pode mudar durante a execução do programa.
- ✓ **Orientada a objetos:** o JavaScript é uma linguagem orientada a objetos, permitindo a criação de objetos que possuem propriedades e métodos.
- ✓ **Integração com HTML e CSS:** o JavaScript é integrado com HTML e CSS, permitindo a criação de páginas da web interativas e dinâmicas.
- ✓ **Multiplataforma:** o JavaScript pode ser executado em diferentes sistemas operacionais e navegadores, tornando-o uma linguagem multiplataforma.

O interpretador Node.js

Como visto anteriormente, o JavaScript é uma linguagem que suporta diferentes plataformas. Para criação de páginas dinâmicas, ele pode ser executado no próprio navegador. Mas no inicio do curso, iremos utilizar o JavaScript como back-end, e para isso, precisamos de um interpretador, o **Node.js**.

O Node.js é um ambiente de tempo de execução JavaScript que permite que o código JavaScript seja executado fora do navegador. Ele usa o motor JavaScript V8, que é o mesmo motor usado pelo Google Chrome, para executar o código JavaScript. O Node.js fornece um ambiente de execução de JavaScript no lado do servidor e é amplamente utilizado para a criação de aplicativos web em tempo real, APIs, serviços de rede, entre outros.

Lição 4: Preparando o ambiente

Acompanhe nossa lição prática em vídeo.

Lição 5: Os tipos de dados no JavaScript

O JavaScript é uma linguagem de programação **dinamicamente tipada**, o que significa que o tipo de dados das variáveis pode mudar durante a execução do programa. Isso significa que, por exemplo, uma variável que armazena uma string pode ser atribuída a um valor numérico ou booleano posteriormente. Isso permite uma maior flexibilidade e facilidade de uso da linguagem, pois não é necessário definir o tipo de dados de uma variável antes de usá-la. Existem vários tipos de dados em JavaScript, cada um com sua própria finalidade e uso.

Os tipos primitivos em JavaScript incluem:

- ✓ **String:** usado para representar um texto, como palavras, frases e números.
- ✓ **Number:** usado para representar um número inteiro ou decimal.
- ✓ **Boolean:** usado para representar um valor verdadeiro ou falso.
- ✓ **Undefined:** usado quando uma variável é declarada, mas não foi atribuído um valor.
- ✓ **Null:** usado quando uma variável é explicitamente definida como nula.
- ✓ **Symbol:** usado para criar identificadores únicos.

Além desses tipos primitivos, JavaScript também possui dois tipos compostos:

- ✓ **Object:** usado para representar um conjunto de dados relacionados, como uma coleção de propriedades e métodos.
- ✓ **Array:** uma variante do tipo Object, usado para representar uma coleção ordenada de dados.

Alguns tipos de dados veremos com mais detalhes ao longo do curso.

Uma das vantagens dos tipos de dados em JavaScript é a capacidade de **realizar conversões entre eles**. Isso permite que dados sejam manipulados e processados de várias maneiras. Por exemplo, é possível converter um número em uma string ou um booleano em um número.

Outra vantagem dos tipos de dados em JavaScript é a capacidade de passá-los entre funções como argumentos. Isso permite que funções aceitem diferentes tipos de dados e ajuda a aumentar a modularidade e reusabilidade do código.

String

Em JavaScript, o tipo de dado "**string**" é usado para **representar texto ou sequências de caracteres**. Uma string é definida por uma sequência de caracteres entre aspas simples (' ') ou aspas duplas (" "). Aqui estão alguns exemplos de strings em JavaScript:

```
let nome = "Maria";
let mensagem = 'Olá, mundo!';
let endereco = "Rua do exemplo, 123";
```

As strings em JavaScript também podem ser vazias, o que significa que elas não contêm nenhum caractere. Isso é representado por uma string com aspas simples ou duplas sem nenhum caractere dentro delas:

```
let stringVazia1 = "";
let stringVazia2 = '';
```

As strings em JavaScript também têm uma propriedade chamada "**length**" que retorna o número de caracteres em uma string:

```
let mensagem = "Olá, mundo!";
console.log(mensagem.length); // imprime 12
```

Além disso, as strings em JavaScript suportam operações como concatenação e indexação. A concatenação é usada para unir duas ou mais strings em uma única string:

```
let saudacao = "Olá, ";
let nome = "Maria";
let mensagem = saudacao + nome; // mensagem será "Olá, Maria"
```

A indexação é usada para acessar caracteres individuais em uma string. Os caracteres em uma string são indexados começando em 0 para o primeiro caractere e incrementando de 1 para cada caractere subsequente:

```
let nome = "Maria";
console.log(nome[0]); // imprime "M"
console.log(nome[1]); // imprime "a"
console.log(nome[2]); // imprime "r"
```

Number

Em JavaScript, o tipo de dado "**number**" é usado para representar **valores numéricos**. Esses valores podem ser inteiros ou decimais (também conhecidos como números de ponto flutuante). Aqui estão alguns exemplos de valores numéricos em JavaScript:

```
let numeroInteiro = 42;
let numeroDecimal = 3.14;
```

Em JavaScript, o tipo "number" é um tipo primitivo, no entanto, ele tem métodos e propriedades embutidos que podem ser usados para realizar operações matemáticas e outras manipulações numéricas.

Uma característica importante do tipo "number" em JavaScript é que ele pode lidar com valores extremamente grandes ou pequenos. No entanto, é importante observar que devido à limitação do tipo "number" em lidar com precisão de ponto flutuante, pode haver problemas de arredondamento em operações matemáticas com números muito grandes ou muito pequenos.

Além disso, o tipo "number" em JavaScript também tem valores especiais como "**Infinity**" (infinito), "**-Infinity**" (negativo infinito) e "**NaN**" (Not a Number), que são usados para representar casos como divisão por zero ou outras operações matemáticas inválidas.

Boolean

Em JavaScript, o tipo de dado "**boolean**" é usado para representar um **valor lógico**, que pode ser verdadeiro (true) ou falso (false). O tipo boolean é usado principalmente em expressões condicionais, como instruções "**if**" e "**while**", veremos melhor sobre em lições futuras. Aqui estão alguns exemplos de valores booleanos em JavaScript:

```
let verdadeiro = true;
let falso = false;
```

Além disso, os valores booleanos em JavaScript também podem ser combinados usando **operadores lógicos** como "**&&**" (e), "**||**" (ou) e "**!**" (não). Esses operadores são usados para criar expressões booleanas mais complexas.

Lição 6: As formas de escrever as variáveis

Em JavaScript, as variáveis são declaradas usando a palavra-chave "**var**", "**let**" ou "**const**". Cada uma dessas palavras-chave tem um comportamento diferente em relação ao escopo e a mutabilidade da variável.

Aqui está um exemplo de como declarar uma variável em JavaScript usando a palavra-chave "**var**":

```
var nomeDaVariavel = valor;
```

Onde "**nomeDaVariavel**" é o nome da variável que está sendo declarada e "**valor**" é o valor que está sendo atribuído à variável.

Além da palavra-chave "var", é possível também declarar variáveis usando "**let**" e "**const**". A principal diferença entre essas palavras-chave é que "**let**" permite que a **variável seja**

reatribuída, enquanto "**const**" cria uma **variável imutável**. Elas são declaradas da mesma forma que o exemplo acima.

Ao declarar variáveis em JavaScript, é importante escolher nomes de variáveis descritivos e significativos. Isso torna o código mais fácil de ler e entender. Além disso, é uma boa prática inicializar as variáveis com um valor padrão para evitar erros durante a execução do programa.

A principal diferença entre a declaração de variáveis com "let" e "var" em JavaScript é como elas tratam o escopo da variável.

Ao declarar uma variável com "var", ela é elevada (hoisted) para o topo do escopo em que foi declarada, antes mesmo da execução do código. Isso significa que uma variável declarada com "var" pode ser usada antes mesmo de sua declaração no código. No entanto, o escopo de uma variável declarada com "var" é o escopo da função em que ela foi declarada ou, se não estiver dentro de uma função, o escopo global.

Por outro lado, ao declarar uma variável com "let", ela não é elevada para o topo do escopo e não pode ser usada antes de sua declaração no código. Além disso, o escopo de uma variável declarada com "let" é limitado ao bloco em que foi declarada, seja ele dentro de uma função, um loop ou qualquer outro bloco de código.

Null

Em JavaScript, **null** é um valor que representa a **ausência intencional de qualquer objeto ou valor**. É considerado um tipo primitivo e é utilizado para indicar que uma variável ou objeto não possui um valor válido ou que não existe.

Ao contrário de undefined, que indica a ausência de um valor atribuído, o valor null é atribuído explicitamente a uma variável ou objeto quando o desenvolvedor deseja indicar que o valor é vazio ou inexistente.

Veja alguns exemplos de uso de null em JavaScript:

```
let nome = null; // indica que a variável nome não possui valor válido
let objeto = null; // indica que o objeto não possui valor ou não existe
```

É importante observar que null é considerado um valor primitivo, e não um objeto. Isso significa que, ao contrário de objetos, null não possui propriedades ou métodos. Além disso, o valor null é considerado um tipo de dado diferente de undefined.

Embora null possa ser útil em algumas situações, é importante ter cuidado ao utilizar esse valor em seu código, pois pode levar a comportamentos inesperados se não for usado corretamente.

Undefined

Em JavaScript, **undefined** é um valor primitivo que indica a **ausência de um valor definido para uma variável ou objeto**. É utilizado quando uma variável é declarada, mas não foi atribuído um valor a ela, ou quando uma propriedade de um objeto não foi definida.

Veja alguns exemplos de uso de undefined em JavaScript:

```
let x; // declaração de uma variável sem atribuição de valor
console.log(x); // undefined
```

Lição 7: Operadores em JavaScript

Em JavaScript, existem vários tipos de operadores que são usados para realizar operações em valores. Aqui estão alguns dos principais tipos de operadores:

1. **Operadores Aritméticos:** São usados para realizar operações matemáticas básicas, como adição, subtração, multiplicação, divisão e resto da divisão. Aqui estão alguns exemplos:

```
let x = 10;
let y = 5;

console.log(x + y); // Adição
console.log(x - y); // Subtração
console.log(x * y); // Multiplicação
console.log(x / y); // Divisão
console.log(x % y); // Resto da divisão
```

2. **Operadores de Atribuição:** São usados para atribuir valores a variáveis. Aqui estão alguns exemplos:

```
let x = 10;
x += 5; // Equivalente a x = x + 5
x -= 5; // Equivalente a x = x - 5
x *= 5; // Equivalente a x = x * 5
x /= 5; // Equivalente a x = x / 5
```

3. **Operadores de Comparação:** São usados para comparar dois valores e retornar um valor booleano (true ou false). Aqui estão alguns exemplos:

```
let x = 10;
let y = 5;

console.log(x > y); // Maior que
console.log(x < y); // Menor que
console.log(x >= y); // Maior ou igual a
console.log(x <= y); // Menor ou igual a
console.log(x == y); // Igual a (compara somente o valor)
console.log(x === y); // Igual a (compara o valor e o tipo de dado)
console.log(x != y); // Diferente de (compara somente o valor)
console.log(x !== y); // Diferente de (compara o valor e o tipo de dado)
```

4. **Operadores Lógicos:** São usados para combinar expressões booleanas e retornar um valor booleano (true ou false). Aqui estão alguns exemplos:
5. **Operadores de Concatenação:** São usados para concatenar duas ou mais strings em uma única string. O operador de concatenação em JavaScript é o sinal de mais (+). Aqui está um exemplo:

```
let nome = "João";
let sobrenome = "Silva";
console.log(nome + " " + sobrenome); // Concatenação de strings
```

Esses são apenas alguns dos principais tipos de operadores em JavaScript. Há também outros tipos, como operadores de deslocamento de bits, operadores ternários e operadores de incremento e decremento, entre outros.

Lição 8: Conversão Implícita e Explícita

Conversão Implícita

A **conversão implícita** em JavaScript ocorre quando um valor de um tipo de dado é automaticamente convertido para outro tipo de dado sem a necessidade de uma operação de conversão explícita por parte do desenvolvedor. Isso acontece quando uma operação é realizada entre valores de tipos de dados diferentes.

Por exemplo, quando um valor numérico é adicionado a uma string, o JavaScript converte automaticamente o número em uma string e concatena com a outra string. Veja o exemplo abaixo:

```
let x = 10;
let texto = "O número é: " + x;
console.log(texto); // O número é: 10
```

Nesse caso, o número 10 é convertido em uma string e concatenado com a string "O número é: ".

Outro exemplo de conversão implícita em JavaScript é quando um valor booleano é usado em uma expressão aritmética. Nesse caso, o valor booleano é convertido em um número, onde true é convertido para 1 e false é convertido para 0. Veja o exemplo abaixo:

```
let x = 10;
let y = true;
let resultado = x + y;
console.log(resultado); // 11
```

Nesse caso, o valor booleano true é convertido para o número 1, e a soma é realizada entre os valores 10 e 1.

A conversão implícita pode ser útil em algumas situações, mas também pode levar a erros e comportamentos inesperados se não for compreendida corretamente. Por isso, é importante entender como os valores são convertidos em diferentes tipos de dados em JavaScript.

Conversão Explícita

A conversão explícita em JavaScript ocorre quando o desenvolvedor força a conversão de um tipo de dado para outro, utilizando funções ou operadores específicos. Diferentemente da conversão implícita, a conversão explícita é realizada de forma consciente e intencional pelo desenvolvedor.

Existem várias funções e operadores em JavaScript que permitem realizar a conversão explícita entre diferentes tipos de dados. Algumas das principais são:

- **Number():** converte um valor em um número;
- **String():** converte um valor em uma string;
- **Boolean():** converte um valor em um booleano;

Veja alguns exemplos de conversão explícita em JavaScript:

```
let x = "10";
let numero = Number(x); // converte a string "10" em um número
console.log(typeof numero); // "number"

let y = 0;
let texto = String(y); // converte o número 0 em uma string "0"
console.log(typeof texto); // "string"

let z = "true";
let booleano = Boolean(z); // converte a string "true" em um booleano true
console.log(typeof booleano); // "boolean"
```

É importante lembrar que, ao realizar uma conversão explícita, o desenvolvedor deve estar ciente de como a conversão irá afetar o valor original e se certificar de que a conversão seja feita de forma correta para evitar comportamentos inesperados no código.

Lição 9: Trabalhando com booleanos e condicionais

Estruturas condicionais

As condicionais em JavaScript são uma estrutura de **controle de fluxo** que permite ao desenvolvedor executar determinado bloco de código apenas se uma determinada condição for atendida. Existem duas condicionais principais em JavaScript: **if/else** e **switch**.

A estrutura **if/else** verifica se uma condição é verdadeira e, em seguida, executa um bloco de código específico para essa condição. Se a condição for falsa, outro bloco de código pode ser executado. Veja um exemplo:

```
let idade = 18;

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

Neste exemplo, a condição idade >= 18 é verificada. Se a idade for maior ou igual a 18 anos, a mensagem "Você é maior de idade" será exibida. Se a condição for falsa, a mensagem "Você é menor de idade" será exibida.

A estrutura **switch** é utilizada quando o desenvolvedor precisa executar diferentes blocos de código para diferentes valores de uma variável. Veja um exemplo:

```
let diaSemana = 1;

switch (diaSemana) {
  case 1:
    console.log("Hoje é segunda-feira");
    break;
  case 2:
    console.log("Hoje é terça-feira");
    break;
  case 3:
    console.log("Hoje é quarta-feira");
    break;
  case 4:
    console.log("Hoje é quinta-feira");
    break;
  case 5:
    console.log("Hoje é sexta-feira");
    break;
  case 6:
    console.log("Hoje é sábado");
    break;
  case 7:
    console.log("Hoje é domingo");
    break;
  default:
    console.log("Dia inválido");
    break;
}
```

Neste exemplo, a variável diaSemana é verificada e diferentes blocos de código são executados de acordo com o valor dela. Se o valor for 1, a mensagem "Hoje é segunda-feira" será exibida. Se o valor for 2, a mensagem "Hoje é terça-feira" será exibida, e assim por diante. Se o valor da variável não for nenhum dos valores esperados, o bloco de código default será executado.

Truthy e Falsy

Truthy e Falsy em JavaScript referem-se aos valores que são considerados **verdadeiros ou falsos** em uma expressão booleana. Em outras palavras, quando uma variável é avaliada em uma expressão booleana, ela pode ser avaliada como verdadeira ou falsa, dependendo do seu valor.

Existem valores que são considerados "falsy", ou seja, são avaliados como false em uma expressão booleana. São eles:

- false
- 0 (zero)

- "" (string vazia)
- null
- undefined
- NaN (Not a Number)

Qualquer outro valor diferente desses é considerado "truthy", ou seja, é avaliado como true em uma expressão booleana.

Lição 10: Conhecendo o operador ternário

O operador ternário em JavaScript é uma **forma abreviada** de escrever uma estrutura **condicional if-else**. Ele permite que você avalie uma expressão e, dependendo do resultado, execute um bloco de código ou outro.

A sintaxe do operador ternário é a seguinte:

```
condição ? valor_se_verdadeiro : valor_se_falso
```

A condição é uma expressão booleana que será avaliada. Se a condição for verdadeira, o valor_se_verdadeiro será retornado. Caso contrário, o valor_se_falso será retornado. Por exemplo:

```
let idade = 18;
let mensagem = idade >= 18 ? "Você é maior de idade" : "Você é menor de idade";

console.log(mensagem); // "Você é maior de idade"
```

Neste exemplo, a condição é **idade >= 18**, que é verdadeira porque **idade** é igual a **18**. Portanto, a mensagem "Você é maior de idade" é atribuída à variável **mensagem**. Agora, vamos supor que idade seja igual a 16:

```
let idade = 16;
let mensagem = idade >= 18 ? "Você é maior de idade" : "Você é menor de idade";

console.log(mensagem); // "Você é menor de idade"
```

Neste caso, a condição é **idade >= 18**, que é falsa porque **idade** é menor que **18**. Portanto, a mensagem "Você é menor de idade" é atribuída à variável **mensagem**.

O operador ternário é útil quando você precisa atribuir um valor a uma variável com base em uma condição simples. No entanto, quando a lógica fica mais complexa, é recomendado utilizar a estrutura condicional **if-else** convencional.

Lição 11: Diferentes formas para manipular strings

Existem diferentes formas para manipular strings, algumas delas já vimos ao longo do curso, outras iremos comentar nessa lição.

Concatenação

Para juntar duas ou mais strings em JavaScript, basta utilizar o operador **+**:

```
let str1 = "Olá";
let str2 = "mundo";
let str3 = str1 + " " + str2; // "Olá mundo"
```

Acesso a caracteres

Para acessar um caractere específico de uma string em JavaScript, é possível utilizar a sintaxe de colchetes **[]**:

```
let str = "Hello world";
let char = str[0]; // "H"
```

Note que os índices de strings em JavaScript começam em zero.

Comprimento

Para obter o comprimento de uma string em JavaScript, basta utilizar a propriedade **length**:

```
let str = "Hello world";
let len = str.length; // 11
```

Substrings

Para obter uma substring de uma string em JavaScript, é possível utilizar o método **substring()**:

```
let str = "Hello world";
let substr = str.substring(0, 5); // "Hello"
```

O primeiro argumento de substring() é o índice do primeiro caractere da substring desejada, enquanto o segundo argumento é o índice do último caractere da substring desejada (mas não incluído na substring final).

Busca e substituição

Para buscar e substituir substrings em JavaScript, é possível utilizar os métodos **indexOf()** e **replace()**:

```
let str = "Hello world";
let index = str.indexOf("world"); // 6
let newStr = str.replace("world", "JavaScript"); // "Hello JavaScript"
```

O método **indexOf()** retorna o índice da primeira ocorrência de uma substring em uma string. Já o método **replace()** substitui a primeira ocorrência de uma substring por outra substring.

Essas são apenas algumas das formas de manipular strings em JavaScript. Existem muitas outras, como transformação de caixa (maiúsculas/minúsculas), remoção de espaços em branco, entre outras.

Template literal

Template literal é uma sintaxe introduzida na versão ES6 (ECMAScript 2015) do JavaScript que permite a **criação de strings com expressões embutidas**. Antes da introdução do template literal, era comum a utilização de concatenação de strings com variáveis ou expressões, utilizando o operador **+**. Com o template literal, é possível incluir expressões diretamente dentro da string utilizando o operador **${}**.

A sintaxe básica do template literal é a seguinte:

```
`string com expressões embutidas ${expressao1} mais expressões ${expressao2}`
```

Por exemplo, suponha que temos as seguintes variáveis:

```
const nome = 'João';
const idade = 25;
```

Podemos criar uma string que inclui essas variáveis da seguinte forma:

```
const mensagem = `Meu nome é ${nome} e eu tenho ${idade} anos.`;
```

O resultado da variável **mensagem** será a seguinte string:

```
Meu nome é João e eu tenho 25 anos.
```

Note que as variáveis **nome** e **idade** são incluídas dentro da string utilizando a sintaxe **${}**. O uso do template literal é bastante comum em frameworks e bibliotecas de JavaScript, principalmente na criação de interfaces de usuário.

Lição 12: A importância das funções e como utilizá-las

Em JavaScript, as **funções** são uma das **estruturas fundamentais da linguagem** e são usadas para agrupar instruções relacionadas e reutilizá-las em diferentes partes do código. Existem

várias maneiras de declarar funções em JavaScript, incluindo a **declaração de função**, **expressão de função**, **arrow function** e IIFE (Immediately Invoked Function Expression).

A declaração de função é a forma mais comum de definir uma função em JavaScript e tem a seguinte sintaxe:

```
function nomeDaFuncao(parametros) {
  // corpo da função
  // código que será executado quando a função for chamada
  return resultado; // opcional
}
```

Onde:

- ✓ **nomeDaFuncao:** é o nome que damos à função.
- ✓ **parametros:** são os parâmetros que a função recebe. Podem ser zero ou mais parâmetros separados por vírgulas.
- ✓ **corpo da função:** é o bloco de código que será executado quando a função for chamada.
- ✓ **return:** é a palavra-chave usada para retornar um valor da função. É opcional.

Por exemplo, a seguinte função recebe dois parâmetros e retorna a soma dos mesmos:

```
function somar(a, b) {
  return a + b;
}
```

Para chamar a função, basta utilizar seu nome e passar os valores dos parâmetros:

```
const resultado = somar(2, 3); // resultado é 5
```

Por fim, é importante destacar que as funções em JavaScript são de primeira classe, o que significa que elas podem ser atribuídas a variáveis, passadas como parâmetros para outras funções e retornadas como valores de outras funções. Esse recurso é muito útil para criar funções mais complexas e reutilizáveis.

Lição 13: Outras formas de escrever funções – expressão de funções e arrow function

Expressão de funções

A expressão de função em JavaScript é uma maneira de criar uma função atribuindo-a a uma variável, constante ou propriedade de objeto. A sintaxe da expressão de função é semelhante à da declaração de função, mas em vez de usar a palavra-chave function para declarar a função, usamos a palavra-chave **const**, **let** ou **var** para declarar uma variável e atribuir a ela uma função.

Abaixo está um exemplo de como criar uma expressão de função:

```
const minhaFuncao = function(parametro1, parametro2) {
  // código a ser executado pela função
};
```

Nesse exemplo, **minhaFuncao** é uma variável que armazena uma função que recebe dois parâmetros. Essa função pode ser chamada posteriormente passando os valores apropriados para seus parâmetros.

É importante notar que, diferentemente da declaração de função, a expressão de função **não é elevada ao topo do escopo**. Isso significa que, se você tentar chamar a função antes de declará-la, receberá um erro. Portanto, é uma boa prática declarar as expressões de função antes de chamá-las.

As expressões de função podem ser úteis quando você precisa passar uma função como argumento para outra função ou quando deseja atribuir diferentes funções a uma mesma variável em diferentes contextos.

Arrow function

As Arrow Functions, ou funções de seta, são uma sintaxe alternativa para a criação de funções em JavaScript, introduzidas na versão ES6 da linguagem. Elas são geralmente mais **concisas** e podem **tornar o código mais legível**.

A sintaxe básica das Arrow Functions é a seguinte:

```
const minhaFuncao = (param1, param2) => {
  // corpo da função
};
```

Nessa sintaxe, os parênteses envolvendo os parâmetros são opcionais se houver apenas um parâmetro. Se a função não receber parâmetros, deve-se usar um par de parênteses vazios. Também é possível omitir as chaves e a palavra-chave **return** se a função tiver apenas uma expressão a ser retornada:

```
const minhaFuncao = () => "retorno da função";
```

As Arrow Functions podem ser uma ótima opção para tornar o código mais conciso e legível em situações em que a complexidade da função é relativamente baixa. No entanto, é importante lembrar que elas não são uma substituição completa para as funções normais e que devem ser usadas com moderação e em situações apropriadas.

Lição 14: Entendendo sobre a ferramenta console.api

O **console** é um objeto global no navegador e no ambiente Node.js que fornece métodos para exibir informações no console do desenvolvedor, como mensagens de depuração, erros e outras informações importantes sobre a execução do código.

A interface **console** possui diversos métodos, como **console.log()**, **console.warn()**, **console.error()**, entre outros, que permitem que os desenvolvedores exibam informações no console do navegador ou do ambiente Node.js.

O método **console.log()** é o mais comumente usado e permite que você exiba informações na forma de uma string ou um objeto no console. Você pode passar vários argumentos para o método **console.log()** e ele os exibirá separados por vírgula no console.

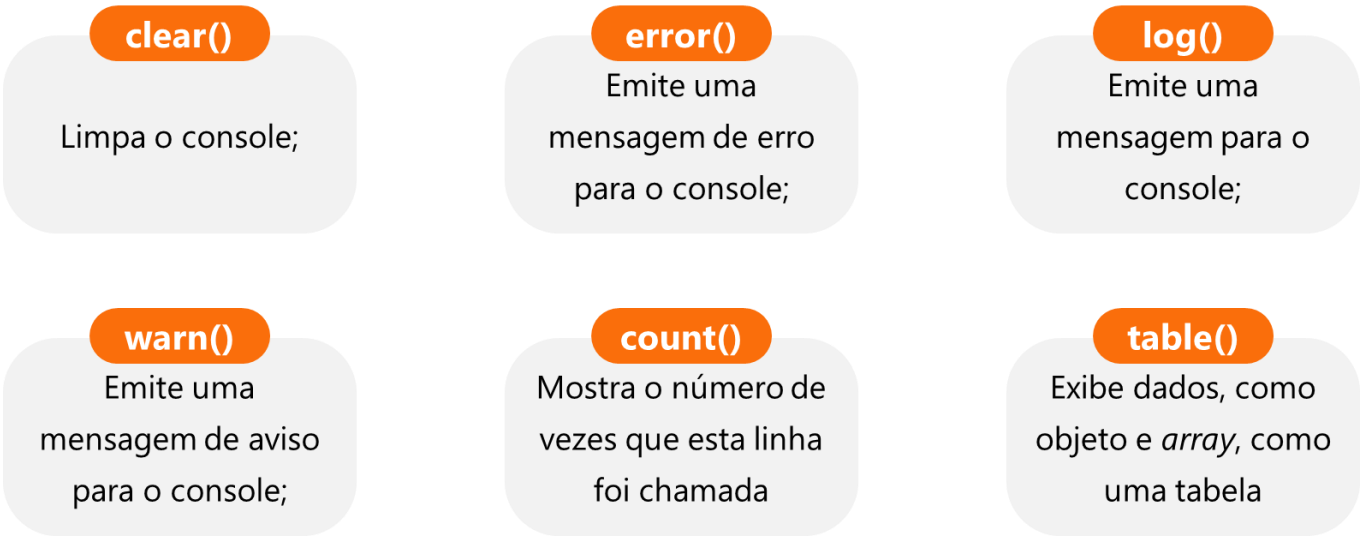
O console também possui métodos para exibir informações no console de forma mais estruturada e legível, como o **console.table()** que exibe um objeto em forma de tabela. Além disso, o console também pode ser usado para medir o tempo de execução do código usando o **console.time()** e **console.timeEnd()**:

```
console.time("tempo de execução");

// código a ser medido

console.timeEnd("tempo de execução");
```

Na imagem abaixo temos alguns exemplos de console:



O console é uma ferramenta essencial para o desenvolvimento e depuração de código JavaScript. É importante lembrar que o console deve ser usado apenas para depuração e nunca deve ser usado para exibir informações sensíveis ou confidenciais, pois as informações exibidas

no console podem ser acessadas por qualquer pessoa que tenha acesso ao console do navegador ou do ambiente Node.js.

Lição 15: Desafio Tech

Acompanhe nossa lição prática em vídeo.

Lição 16: 5 passos práticos para aplicar o que você aprendeu

1. Conheça os **principais conceitos** acerca do JavaScript;
2. Prepare seu espaço de programação a partir do seu **gosto pessoal**;
3. Saiba os **tipos de variáveis** e como declará-las;
4. Anote as diferentes formas de **declaração de função**;
5. Aplique todos os **conhecimentos aprendidos** para fixação.

Módulo 2
Arrays e estruturas de repetição

Lição 1: Objetivos do módulo

Neste segundo módulo, estudaremos o que são **arrays** e porque são importantes. Mostraremos como alterar arrays com os métodos do JavaScript. Além disso, entenderemos o que são **laços de repetição** e como usá-los.

Responderemos perguntas do tipo:

- ✓ O que são arrays?
- ✓ Quais são os métodos que podemos utilizar nos arrays?
- ✓ Como usar os laços de repetição em JavaScript?

Lição 2: Estruturas de repetição dentro do JavaScript

As estruturas de repetição em JavaScript permitem que você execute uma ou mais instruções várias vezes. Existem duas estruturas de repetição em JavaScript: **for** e **while**.

Estrutura de repetição **"for"**

A estrutura de repetição **for** é útil quando você **sabe quantas vezes deseja executar** um conjunto de instruções. A sintaxe é a seguinte:

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

Aqui está um exemplo de como usar um loop for para imprimir os números de 0 a 4:

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

Neste exemplo, a variável **i** é inicializada com 0 e incrementada em 1 a cada iteração do loop. O loop será executado enquanto **i** for menor que 5.

Estrutura de repetição **"while"**

A estrutura de repetição **while** é útil quando você **não sabe quantas vezes deseja executar** um conjunto de instruções, mas sabe qual condição deve ser atendida para continuar executando o loop. A sintaxe é a seguinte:

```
while (condição) {
  // instruções a serem executadas
}
```

Aqui está um exemplo de como usar um loop **while** para imprimir os números de 0 a 4:

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

Neste exemplo, a variável **i** é inicializada com 0 fora do loop. Enquanto **i** for menor que 5, o loop continuará a imprimir o valor atual de **i** e incrementar **i**.

Estrutura de repetição **“do...while”**

A estrutura de repetição **do...while** é semelhante à estrutura de repetição **while**, exceto que a **condição é verificada no final do loop, em vez de no início**. Isso significa que o código dentro do loop será executado pelo menos uma vez, mesmo se a condição for falsa desde o início. A sintaxe é a seguinte:

```
do {
  // instruções a serem executadas
} while (condição);
```

Aqui está um exemplo de como usar um loop **do...while** para imprimir os números de 0 a 4:

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

Neste exemplo, a variável **i** é inicializada com 0 fora do loop. O código dentro do loop será executado pelo menos uma vez, mesmo que **i** seja inicialmente maior ou igual a 5. A condição é verificada no final do loop, então o loop continuará a ser executado enquanto **i** for menor que 5.

Loops infinitos

Loops infinitos em JavaScript são loops que **não têm uma condição de saída definida corretamente**, fazendo com que o loop continue a ser executado para sempre. Isso pode causar travamentos e problemas no navegador ou no ambiente em que o código está sendo executado. Um exemplo de loop infinito em JavaScript é o seguinte:

```
while (true) {
  console.log('Este é um loop infinito!');
}
```

Nesse exemplo, a condição do loop é **true**, o que significa que ele será executado continuamente sem parar. Para evitar loops infinitos, é importante ter certeza de que a condição de saída do loop seja definida de forma adequada, garantindo que ele pare de ser executado após atingir um determinado número de repetições ou quando uma condição específica for satisfeita.

Lição 3: O que são arrays e como utilizá-las?

Em JavaScript, um **array** é uma **estrutura de dados** que permite **armazenar uma coleção de valores, como números, strings, objetos e até mesmo outros arrays**. Cada elemento em um array é identificado por um índice numérico, que começa em 0 para o primeiro elemento e aumenta em 1 para cada elemento subsequente.

Os arrays em JavaScript podem ser declarados de várias maneiras, incluindo:

- Usando a palavra-chave new e o construtor Array:

```
let myArray = new Array();
```

- Usando uma sintaxe de literal de array, que é uma maneira mais comum de declarar um array:

```
let myArray = [];
```

Os elementos de um array podem ser **acessados e manipulados** usando **seus índices numéricos**. Por exemplo, para acessar o terceiro elemento de um array, podemos usar a seguinte sintaxe:

```
let myArray = [1, 2, 3, 4, 5];
console.log(myArray[2]); // Saída: 3
```

Os arrays em JavaScript também têm vários métodos úteis que podem ser usados para manipular e trabalhar com seus elementos, como **push()**, **pop()**, **shift()**, **unshift()**, **slice()**, **splice()**

e muitos outros. Esses métodos permitem adicionar ou remover elementos do array, concatenar arrays, copiar partes do array e muito mais.

Lição 4: Métodos de array – Para o que servem? – Parte 1

Existem muitos **métodos de array** em JavaScript que permitem manipular e trabalhar com os elementos do array. Alguns dos métodos mais comuns são:

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

```
let myArray = [1, 2, 3];
myArray.push(4);
console.log(myArray); // Saída: [1, 2, 3, 4]
```

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

```
let myArray = [1, 2, 3];
let removedElement = myArray.pop();
console.log(myArray); // Saída: [1, 2]
console.log(removedElement); // Saída: 3
```

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

```
let myArray = [1, 2, 3];
let removedElement = myArray.shift();
console.log(myArray); // Saída: [2, 3]
console.log(removedElement); // Saída: 1
```

- **unshift():** adiciona um ou mais elementos ao início do array e retorna o novo comprimento do array.

```
let myArray = [1, 2, 3];
myArray.unshift(0, -1);
console.log(myArray); // Saída: [-1, 0, 1, 2, 3]
```

Lição 5: Métodos de array – Para o que servem? – Parte 2

Nessa lição veremos outros métodos de array, são eles:

- **slice():** retorna uma cópia do array com os elementos selecionados. Os parâmetros **start** e **end** especificam os índices do array a serem selecionados (**o end é exclusivo**).

```
let myArray = [1, 2, 3, 4, 5];
let selectedElements = myArray.slice(1, 4);
console.log(selectedElements); // Saída: [2, 3, 4]
```

- **splice():** adiciona ou remove elementos do array. Os parâmetros **start** e **deleteCount** especificam os índices do array a serem removidos e a quantidade de elementos a serem removidos. Os parâmetros adicionais especificam os elementos a serem adicionados ao array.

```
let myArray = [1, 2, 3, 4, 5];
myArray.splice(1, 2, 'a', 'b', 'c');
console.log(myArray); // Saída: [1, 'a', 'b', 'c', 4, 5]
```

- **concat():** retorna um novo array que é uma concatenação de dois ou mais arrays.

```
let myArray1 = [1, 2, 3];
let myArray2 = [4, 5, 6];
let concatenatedArray = myArray1.concat(myArray2);
console.log(concatenatedArray); // Saída: [1, 2, 3, 4, 5, 6]
```

Os métodos push, pop, unshift, shift e splice alteram o array original, enquanto os métodos slice e concat não alteram o array original, geralmente atribuímos o resultado a uma variável determinada.

Lição 6: Usando métodos para procurar elementos em um array

Em JavaScript, existem vários métodos que podem ser usados para procurar elementos em um array. Aqui estão alguns dos métodos mais comuns:

- **indexOf():** Este método procura um elemento específico em um array e retorna o índice da primeira ocorrência desse elemento. Se o elemento não for encontrado, o método retorna -1. O **indexOf()** aceita um argumento obrigatório, que é o elemento a ser procurado.

```
const array = [1, 2, 3, 4, 5];
const index = array.indexOf(3);
console.log(index); // output: 2
```

- **lastIndexOf():** Este método funciona de maneira semelhante ao **indexOf()**, mas começa a procurar o elemento a partir do final do array. Ele retorna o índice da última ocorrência do elemento ou -1 se o elemento não for encontrado.

```
const array = [1, 2, 3, 4, 3, 5];
const index = array.lastIndexOf(3);
console.log(index); // output: 4
```

- **find():** Este método retorna o primeiro elemento no array que atende a uma determinada condição. Ele aceita uma função de retorno de chamada como argumento, que deve retornar verdadeiro ou falso para cada elemento no array. O método retorna o primeiro elemento para o qual a função de retorno de chamada retorna verdadeiro.

```
const array = [1, 2, 3, 4, 5];
const found = array.find(element => element > 3);
console.log(found); // output: 4
```

- **findIndex():** Este método é semelhante ao **find()**, mas em vez de retornar o elemento encontrado, ele retorna o índice do primeiro elemento que atende à condição especificada.

```
const array = [1, 2, 3, 4, 5];
const index = array.findIndex(element => element > 3);
console.log(index); // output: 3
```

Esses são apenas alguns dos métodos disponíveis para procurar elementos em um array em JavaScript. Existem muitos outros, como **filter()**, **includes()**, **some()** e **every()**, que podem ser usados para realizar operações mais complexas em arrays.

Lição 7: Funções callbacks para estruturas de repetição

As **funções de callback** são muito úteis quando usadas em conjunto com estruturas de repetição em JavaScript. Elas permitem **executar uma função em cada elemento de um array**, ou em cada item de um conjunto de dados, e fazer algo com esse elemento.

Aqui estão alguns exemplos de como usar funções de callback com estruturas de repetição:

- **forEach():** Este método executa uma função de callback em cada elemento de um array. Ele aceita uma função de retorno de chamada como argumento, que é chamada uma vez para cada elemento no array. A função de retorno de chamada recebe **três argumentos**: o **valor atual** do elemento, o **índice** do elemento e o **próprio array**.

```
const array = [1, 2, 3, 4, 5];
array.forEach(element => console.log(element));
```

- **map():** Este método é semelhante ao **forEach()**, mas em vez de apenas executar uma função de callback em cada elemento, ele cria um novo array com os resultados da função de retorno de chamada aplicada a cada elemento no array original. A função de retorno de chamada recebe **três argumentos**: o **valor atual** do elemento, o **índice** do elemento e o **próprio array**.

```
const array = [1, 2, 3, 4, 5];
const newArray = array.map(element => element * 2);
console.log(newArray); // output: [2, 4, 6, 8, 10]
```

Lição 8: Outros métodos com função callback – Filter e Reduce

Nessa lição veremos outros dois métodos com função callback, são eles:

- **filter():** Este método cria um novo array com todos os elementos do array original que atendem a uma determinada condição. Ele aceita uma função de retorno de chamada como argumento, que é chamada uma vez para cada elemento no array original. A função de retorno de chamada recebe **três argumentos**: o **valor atual** do elemento, o **índice** do elemento e o **próprio array**.

```
const array = [1, 2, 3, 4, 5];
const newArray = array.filter(element => element > 3);
console.log(newArray); // output: [4, 5]
```

- **reduce():** Este método executa uma função de callback em cada elemento de um array, retornando um único valor resultante. Ele aceita uma função de retorno de chamada como argumento, que é chamada uma vez para cada elemento no array original. A função de retorno de chamada recebe **dois argumentos**: o **valor acumulado** e o **valor atual** do elemento.

```
const array = [1, 2, 3, 4, 5];
const sum = array.reduce((accumulator, currentValue) => accumulator + currentValue);
console.log(sum); // output: 15
```

Esses são apenas alguns exemplos de como as funções de callback podem ser usadas com estruturas de repetição em JavaScript. Existem muitas outras possibilidades e é importante conhecer bem esses conceitos para desenvolver soluções eficientes e escaláveis.

Lição 9: Desafio Tech

Acompanhe nossa lição prática em vídeo.

Lição 10: 5 passos práticos para aplicar o que você aprendeu

1. Saiba como utilizar **estruturas de repetições** em JavaScript;
2. Entenda como **declarar *arrays***, como funcionam e como acessar suas propriedades.
3. Conheça os **principais métodos** de *arrays*;
4. Entenda o que são **funções callback** e como elas são importantes para manipular *arrays*;
5. Aplique todos os **conhecimentos aprendidos** para fixação.

Módulo 3
Objetos e protótipos

Lição 1: Objetivos do módulo

Neste terceiro módulo estudaremos o que são **objetos** e por que são importantes. Entenderemos o que são suas **propriedades e métodos**, e como podemos adicioná-los, acessá-los e alterá-los. Veremos também o conceito de **protótipos** no JavaScript.

Responderemos perguntas do tipo:

- ✓ O que são objetos e protótipos?
- ✓ Como criar e manipular objetos?
- ✓ Como percorrer os objetos?

Lição 2: Objetos em JavaScript – como declarar e acessar os elementos

Objetos em JavaScript são uma **estrutura de dados** que permite **armazenar informações em pares de chave-valor**. As chaves são strings que identificam os valores associados a elas. Os valores podem ser de qualquer tipo de dados em JavaScript, incluindo outros objetos.

A sintaxe básica para criar um objeto em JavaScript é usando chaves **{}** e separando cada chave-valor com uma vírgula. Por exemplo:

```
let pessoa = {
  nome: "João",
  idade: 30,
  email: "joao@example.com"
};
```

Neste exemplo, o objeto pessoa possui **três propriedades**: nome, idade e email. Os valores são uma string, um número e outra string, respectivamente.

Para acessar os valores das propriedades de um objeto em JavaScript, podemos usar a **notação de ponto (.)** ou a **notação de colchetes ([])**. Por exemplo:

```
console.log(pessoa.nome); // "João"
console.log(pessoa['idade']); // 30
```

Outra característica importante dos objetos em JavaScript é que eles podem ter métodos, que são funções associadas a um objeto. Esses métodos podem ser definidos ao criar o objeto ou adicionados posteriormente, iremos ver com mais detalhes nas próximas lições.

Lição 3: Como adicionar ou alterar valores nos objetos

Em JavaScript, é possível adicionar ou alterar valores em um objeto de várias maneiras simplesmente atribuindo um novo valor a uma chave. A notação de ponto é usada para acessar as propriedades do objeto diretamente e alterá-las. Para adicionar uma nova propriedade, basta especificar o nome da propriedade e atribuir um valor a ela:

```
const objeto = {};
objeto.propriedade1 = "valor1";
objeto.propriedade2 = "valor2";
```

Para alterar uma propriedade existente, basta acessá-la usando a notação de ponto e atribuir um novo valor a ela:

```
objeto.propriedade1 = "novo_valor1";
```

Para deletar uma propriedade de um objeto em JavaScript, podemos usar o operador **delete**. A sintaxe é a seguinte:

```
delete objeto.propriedade;
```

Onde objeto é o **objeto** do qual queremos deletar a propriedade e **propriedade** é o nome da propriedade que queremos deletar. Por exemplo:

```
const pessoa = { nome: 'Maria', idade: 30 };
delete pessoa.idade;
console.log(pessoa); // { nome: 'Maria' }
```

Nesse exemplo, estamos deletando a propriedade **idade** do objeto **pessoa**. Depois de deletada, a propriedade não existe mais no objeto e ao imprimir o objeto no console, apenas a propriedade **nome** é exibida.

É importante lembrar que a propriedade é deletada apenas do objeto em si, e não da sua "classe" ou protótipo. Ou seja, se o objeto em questão possuir uma propriedade herdada de seu protótipo, essa propriedade continuará existindo, porém entenderemos sobre protótipo e classes nas próximas lições.

Lição 4: A notação de colchete – outra forma de acessar valores nos objetos

Em JavaScript, é possível acessar as propriedades de um objeto usando a **notação de colchetes**. Essa notação permite que você acesse as propriedades de um objeto dinamicamente, ou seja, usando uma variável ou uma expressão para definir o nome da propriedade.

Em geral, a notação do ponto é mais comum e mais fácil de ler, especialmente quando o nome da propriedade é conhecido antecipadamente. Além disso, a notação do ponto é mais eficiente em termos de desempenho.

Já a **notação de colchetes** é **mais flexível**, permitindo que você acesse as propriedades do objeto de forma **dinâmica**, usando uma variável ou uma expressão para definir o nome da propriedade. Isso é especialmente útil quando você não sabe o nome da propriedade de antemão ou quando precisa acessar uma propriedade com um nome inválido em JavaScript. Por exemplo:

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

let propriedade = "idade";
console.log(pessoa[propriedade]); // 30
```

Observe que a variável "propriedade" é usada dentro dos colchetes para definir o nome da propriedade que queremos acessar. Isso nos permite acessar as propriedades do objeto de forma dinâmica, sem precisar saber o nome da propriedade de antemão.

Objects.keys

Em JavaScript, **Object.keys()** é um método que permite obter um array contendo todas as chaves (propriedades) de um objeto. O método recebe um objeto como argumento e retorna um array contendo as chaves do objeto. A sintaxe para o uso do **Object.keys()** é a seguinte:

```
Object.keys(objeto);
```

Onde **objeto** é o objeto do qual se deseja obter as chaves. Por exemplo, suponha que tenhamos o seguinte objeto:

```
const carro = {
  marca: "Honda",
  modelo: "Civic",
  ano: 2020,
};
```

Podemos obter um array contendo todas as chaves do objeto **carro** usando o método **Object.keys()** da seguinte forma:

```
const chaves = Object.keys(carro);
console.log(chaves); // ["marca", "modelo", "ano"]
```

O método **Object.keys()** é útil para obter as chaves de um objeto e usá-las em iterações ou outras operações com o objeto.

Lição 5: A utilização do for...in para percorrer objetos

O **for...in** é uma estrutura de repetição do JavaScript que pode ser utilizada para **percorrer as propriedades de um objeto**. A sintaxe básica é a seguinte:

```
for (let propriedade in objeto) {
  // código a ser executado para cada propriedade
}
```

Onde **propriedade** é uma variável que receberá o nome de cada propriedade do objeto e **objeto** é o objeto a ser percorrido. Por exemplo, considerando o seguinte objeto:

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

Podemos percorrer suas propriedades utilizando o **for...in**, da seguinte forma:

```
for (let propriedade in pessoa) {
  console.log(propriedade + ": " + pessoa[propriedade]);
}
```

Neste exemplo, para cada propriedade do objeto **pessoa**, a string composta pela propriedade e seu valor é exibida no console. O resultado seria:

```
nome: João
idade: 30
profissao: Desenvolvedor
```

Cabe destacar que o **for...in** pode não percorrer as propriedades de um objeto na ordem em que foram definidas, e sim em uma ordem arbitrária. Além disso, ele também percorre as propriedades herdadas do protótipo do objeto, o que pode não ser desejado em alguns casos. Por isso, é importante ter cautela ao utilizar o for...in para percorrer objetos.

Lição 6: Outros tipos de dados – arrays e encadeamento de objetos

Além de strings e números, objetos podem armazenar outros tipos de dados como arrays e até mesmo outros objetos. Veremos com mais detalhes nessa lição.

Arrays dentro de objetos

Arrays são estruturas de dados que permitem armazenar uma coleção de elementos de um mesmo tipo em uma única variável. Por sua vez, os objetos em JavaScript são estruturas de dados que permitem armazenar um conjunto de pares chave/valor. Mas e se quisermos armazenar uma lista de valores em um objeto? Podemos usar arrays dentro de objetos.

Para fazer isso, basta atribuir um array a uma propriedade de um objeto. Por exemplo:

```
const meuObjeto = {
  nome: 'João',
  idade: 30,
  hobbies: ['futebol', 'cinema', 'leitura']
};
```

Nesse exemplo, a propriedade **hobbies** do objeto **meuObjeto** é um array que armazena três elementos. Podemos acessar esses elementos da mesma forma que em qualquer outro array:

```
console.log(meuObjeto.hobbies[0]); // futebol
console.log(meuObjeto.hobbies[1]); // cinema
console.log(meuObjeto.hobbies[2]); // leitura
```

Também podemos usar métodos de array para trabalhar com os elementos do array dentro do objeto. Por exemplo, podemos adicionar um novo hobby usando o **método push**:

```
meuObjeto.hobbies.push('pintura');
console.log(meuObjeto.hobbies); // ['futebol', 'cinema', 'leitura', 'pintura']
```

Também é possível ter arrays de objetos dentro de objetos, criando assim estruturas mais complexas de dados.

Encadeamento de objetos

Em JavaScript, é possível ter objetos dentro de objetos, o que é conhecido como **objetos aninhados ou objetos compostos**. Isso significa que uma propriedade de um objeto pode ser um outro objeto.

Para criar um objeto dentro de outro objeto, basta declarar uma propriedade com o nome desejado e atribuir a ela um novo objeto. Por exemplo:

```
const pessoa = {
  nome: "João",
  idade: 30,
  endereco: {
    rua: "Rua A",
    numero: 123,
    cidade: "São Paulo"
  }
};
```

Nesse exemplo, o objeto **pessoa** tem três propriedades: **nome**, **idade** e **endereco**. A propriedade **endereco** é por sua vez outro objeto, com as propriedades **rua**, **numero** e **cidade**. Para acessar uma propriedade de um objeto dentro de outro objeto, basta encadear as notações de ponto ou colchete. Por exemplo:

```
console.log(pessoa.endereco.rua); // "Rua A"
console.log(pessoa["endereco"]["cidade"]); // "São Paulo"
```

É importante lembrar que a utilização de objetos dentro de objetos pode aumentar a complexidade do código e exigir cuidado na manipulação das propriedades.

Lição 7: O que são as listas de objetos e como trabalhar com elas?

Em JavaScript, é possível ter um **array que contém objetos como elementos**. Isso significa que cada posição do array contém um objeto. Essa é uma maneira comum de armazenar e manipular dados em JavaScript. Para criar um array de objetos, basta declarar um array vazio e adicionar objetos a ele. Por exemplo:

```
let pessoas = [
  { nome: "João", idade: 25 },
  { nome: "Maria", idade: 30 },
  { nome: "Pedro", idade: 20 }
];
```

Neste exemplo, **pessoas** é um array que contém três objetos, cada um representando uma pessoa com nome e idade. É possível acessar os elementos do array normalmente, usando a notação de colchetes e o índice desejado:

```
console.log(pessoas[0]); // { nome: "João", idade: 25 }
console.log(pessoas[1].nome); // "Maria"
console.log(pessoas[2].idade); // 20
```

Também é possível iterar sobre o array usando um loop **for**. Este loop percorre todos os objetos do array e imprime o nome e idade de cada pessoa, por exemplo:

```
for (let i = 0; i < pessoas.length; i++) {
  console.log(pessoas[i].nome + " tem " + pessoas[i].idade + " anos.");
}
```

Além disso, é possível adicionar ou remover objetos do array usando os métodos de array, como **push(), pop(), shift() e unshift()**. Por exemplo, para adicionar uma nova pessoa ao final do array:

```
pessoas.push({ nome: "Ana", idade: 27 });
```

Dessa forma, é possível criar e manipular arrays de objetos em JavaScript para armazenar e acessar dados de maneira organizada.

Lição 8: Métodos de objetos

Em JavaScript, um objeto é uma coleção de propriedades, onde cada propriedade é uma associação de nome (também conhecido como chave) e valor. Os **métodos** em objetos **são funções que estão associadas a uma propriedade**.

Os métodos permitem que você execute ações específicas em um objeto, como adicionar, remover ou modificar propriedades, ou realizar operações que utilizem essas propriedades. Alguns dos métodos mais comuns em objetos em JavaScript são:

- **Object.keys():** Retorna um array contendo os nomes de todas as propriedades enumeráveis de um objeto.

```
Exemplo:
const obj = {a: 1, b: 2, c: 3};
console.log(Object.keys(obj)); // ["a", "b", "c"]
```

- **Object.values():** Retorna um array contendo os valores de todas as propriedades enumeráveis de um objeto.

```
Exemplo:
const obj = {a: 1, b: 2, c: 3};
console.log(Object.values(obj)); // [1, 2, 3]
```

- **Object.entries():** Retorna um array contendo as propriedades e valores de um objeto como sub-arrays.

```
Exemplo:
const obj = {a: 1, b: 2, c: 3};
console.log(Object.entries(obj)); // [["a", 1], ["b", 2], ["c", 3]]
```

- **Object.assign():** Copia os valores de todas as propriedades enumeráveis de um ou mais objetos de origem para um objeto de destino.

```
Exemplo:
const obj1 = {a: 1};
const obj2 = {b: 2};
const obj3 = Object.assign(obj1, obj2);
console.log(obj3); // {a: 1, b: 2}
```

- **Object.freeze():** Congela um objeto, impedindo que as propriedades existentes sejam modificadas e adicionadas ou removidas novas propriedades.

```
Exemplo:
const obj = {a: 1};
Object.freeze(obj);
obj.a = 2;
console.log(obj.a); // 1
```

- **Object.seal():** Sela um objeto, impedindo que as propriedades existentes sejam removidas e novas propriedades sejam adicionadas, mas permitindo que as propriedades existentes sejam modificadas.

```
Exemplo:
const obj = {a: 1};
Object.seal(obj);
obj.a = 2;
obj.b = 3;
delete obj.a;
console.log(obj); // {a: 2}
```

- **Object.hasOwnProperty():** Retorna um valor booleano indicando se o objeto tem a propriedade especificada como uma propriedade direta do objeto, e não como uma propriedade herdada da cadeia de protótipos.

```
Exemplo:
const obj = {a: 1, b: 2};
console.log(obj.hasOwnProperty("a")); // true
console.log(obj.hasOwnProperty("toString")); // false
```

Lição 9: Spread operator – como juntar dois objetos em apenas um

O **spread operator** é um recurso introduzido no ECMAScript 6 (também conhecido como ES6 ou ES2015) que permite que um elemento iterável, como um array ou uma string, seja expandido em um local onde zero ou mais argumentos são esperados. Isso significa que é possível usar o operador spread para **desmembrar um objeto em partes separadas e depois usá-las para criar um novo objeto ou fazer outras operações**.

A sintaxe do spread operator é a reticências (três pontos) seguidas pelo nome da variável que contém o objeto a ser desmembrado. Por exemplo, se tivermos um array chamado "arr", podemos desmembrá-lo usando o spread operator da seguinte forma:

```
const arr = [1, 2, 3];
const newArr = [...arr]; // cria uma cópia do array original
```

No exemplo acima, a variável "newArr" é uma cópia do array "arr", que pode ser modificada sem afetar o array original.

O spread operator também pode ser usado para concatenar arrays, como no exemplo abaixo:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const newArr = [...arr1, ...arr2]; // concatena os dois arrays
```

Além disso, o spread operator pode ser usado para desestruturar um objeto em suas propriedades individuais. Por exemplo:

```
const obj = { x: 1, y: 2 };
const { x, y } = { ...obj }; // desestrutura o objeto
```

Em resumo, o spread operator é um recurso poderoso em JavaScript que permite que elementos iteráveis sejam expandidos em locais onde zero ou mais argumentos são esperados, facilitando a criação de cópias de objetos, a concatenação de arrays e a desestruturação de objetos em suas propriedades individuais.

Lição 10: As funções dentro dos objetos

Em JavaScript, é possível ter **funções dentro de objetos**, o que pode ser bastante útil para trabalhar com objetos que possuem comportamentos específicos. A sintaxe para definir uma função dentro de um objeto é a seguinte:

```
const objeto = {
  propriedade1: valor1,
  propriedade2: valor2,
  nomeDaFuncao: function() {
    // corpo da função
  }
}
```

Nesse exemplo, foi adicionado ao objeto uma propriedade chamada **nomeDaFuncao**, que recebe como valor uma função anônima. A partir desse momento, é possível chamar essa função utilizando a notação de ponto:

```
objeto.nomeDaFuncao();
```

É possível também definir uma função dentro de um objeto utilizando a sintaxe de função arrow. Além disso, uma outra forma de definir funções dentro de objetos é utilizando a sintaxe de método de objeto:

```
const objeto = {
  propriedade1: valor1,
  propriedade2: valor2,
  nomeDaFuncao() {
    // corpo da função
  }
}
```

Essa sintaxe é uma abreviação da sintaxe anterior, em que o nome da função é definido sem a palavra reservada **function**. Da mesma forma, é possível chamar essa função utilizando a notação de ponto.

Quando uma função é definida dentro de um objeto, ela tem acesso às propriedades e métodos desse objeto através da palavra reservada **this**. Por exemplo:

```
const pessoa = {
  nome: "João",
  sobrenome: "Silva",
  nomeCompleto() {
    return this.nome + " " + this.sobrenome;
  }
}
```

Nesse exemplo, a função **nomeCompleto** tem acesso às propriedades **nome** e **sobrenome** do objeto **pessoa** através da palavra reservada **this**. Para chamar essa função e obter o nome completo da pessoa, basta utilizar a notação de ponto:

```
pessoa.nomeCompleto(); // retorna "João Silva"
```

Lição 11: O que é Prototype?

Em JavaScript, todos os **objetos são construídos por meio de funções construtoras** e cada função construtora tem um **objeto prototype** associado a ela. O objeto prototype é um objeto que contém propriedades e métodos que são compartilhados por todos os objetos criados a partir da mesma função construtora. Em outras palavras, o **objeto prototype** é uma **espécie de modelo ou "molde" para todos os objetos criados a partir de uma função construtora**.

Quando você cria um objeto a partir de uma função construtora usando a palavra-chave "new", o objeto criado herda todas as propriedades e métodos do objeto prototype da função construtora. Isso significa que você pode adicionar métodos e propriedades ao objeto prototype de uma função construtora e esses métodos e propriedades serão herdados por todos os objetos criados a partir dessa função.

A utilização do objeto prototype é uma maneira eficiente de compartilhar código e propriedades entre objetos sem ter que criar novas propriedades e métodos para cada objeto individualmente, o que pode levar a uma duplicação de código e aumento do consumo de memória. Além disso, a utilização do objeto prototype permite que os objetos herdem propriedades e métodos adicionados posteriormente ao objeto prototype, mesmo depois de terem sido criados.

Lição 12: As propriedades do prototype e como manipular protótipos

Em JavaScript, todos os **objetos têm um protótipo**, que é um objeto que é usado como modelo para criar o objeto. O protótipo é uma propriedade interna de um objeto que é usada pelo motor JavaScript para fornecer herança de propriedades e métodos entre objetos.

As propriedades do prototype podem ser acessadas usando a sintaxe de ponto ou a sintaxe de colchetes:

```
objectName.propertyName
objectName['propertyName']
```

O objeto prototype tem algumas propriedades e métodos internos que podem ser usados para criar e manipular protótipos. Algumas das propriedades do prototype são:

- **constructor:** é uma propriedade que aponta para a função construtora usada para criar o objeto.
- **proto:** é uma propriedade que aponta para o protótipo do objeto.

Além das propriedades do prototype, há também alguns métodos que podem ser usados para criar e manipular protótipos. Alguns desses métodos são:

- **Object.create():** é um método que cria um novo objeto com o protótipo especificado.
- **Object.getPrototypeOf():** é um método que retorna o protótipo de um objeto especificado.
- **Object.setPrototypeOf():** é um método que define o protótipo de um objeto especificado.

A manipulação de protótipos pode ser útil para estender objetos existentes ou criar novos tipos de objetos com base em objetos existentes. No entanto, é importante ter cuidado ao manipular protótipos, pois pode levar a problemas de desempenho e a erros difíceis de depurar.

Lição 13: Desafio Tech

Acompanhe nossa lição prática em vídeo.

Lição 14: 5 passos práticos para aplicar o que você aprendeu

1. Entenda o que são **objetos** e como utilizá-los;
2. Saiba trabalhar com **diferentes tipos de dados** nos objetos;
3. Conheça os **métodos de objetos** e como percorrê-los;
4. Entenda o que são **os protótipos** e como são importantes para o funcionamento do JavaScript;
5. Aplique todos os **conhecimentos aprendidos** para fixação.

Módulo 4
Programação Orientada a Objetos

Lição 1: Objetivos do módulo

Neste quarto módulo estudaremos os conceitos principais de **orientação a objetos**, como **classes**, **herança**, **encapsulamento** e **polimorfismo**, entendendo como utilizar cada princípio no contexto do JavaScript.

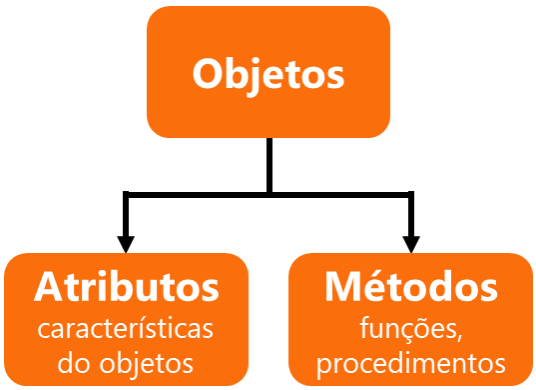
Responderemos perguntas do tipo:

- ✓ Qual a diferença entre classes e objetos?
- ✓ Como utilizar herança e polimorfismo para criar classes?
- ✓ O que é encapsulamento?

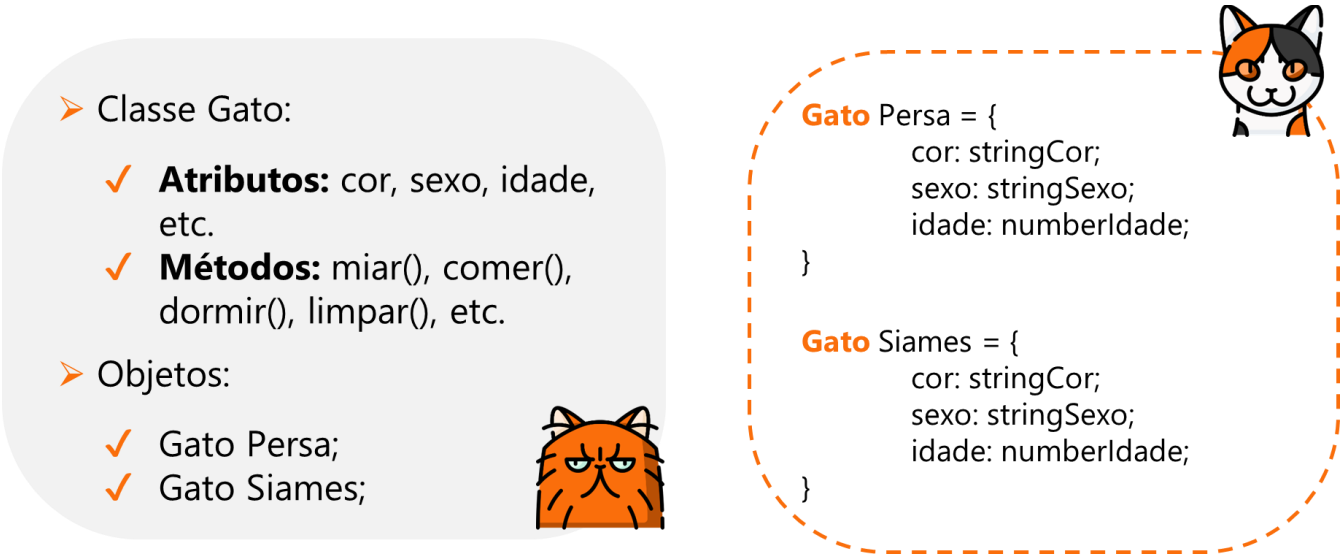
Lição 2: O conceito da programação orientada a objetos

Programação Orientada a Objetos (POO) é um paradigma de programação que se baseia no conceito de objetos, que podem conter dados e comportamentos relacionados. **Cada objeto é uma instância de uma classe**, que define a estrutura e o comportamento do objeto.

Na POO, as classes são utilizadas para definir as características comuns a um conjunto de objetos. Elas especificam os atributos que os objetos terão, bem como os métodos (funções) que podem ser executados nesses objetos.

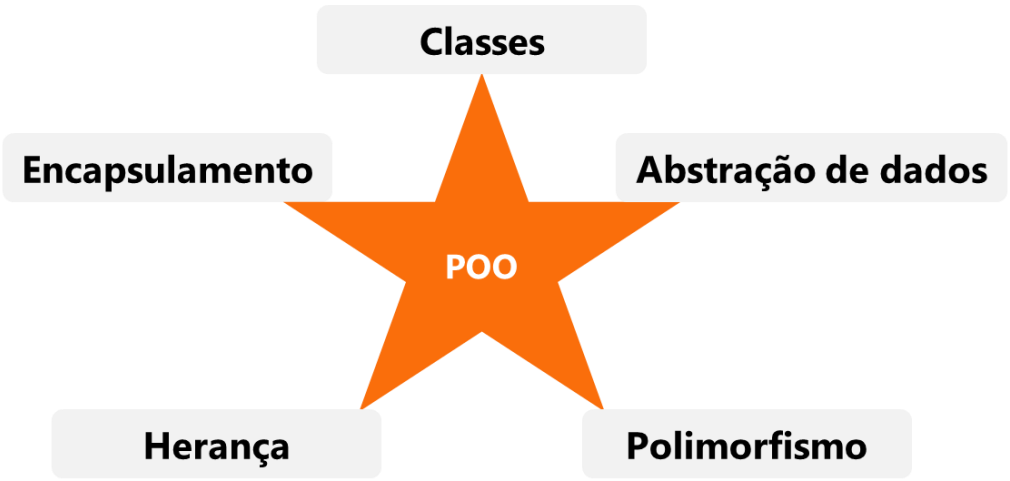


Os objetos são criados a partir das classes, e cada objeto pode ser tratado de forma independente, com suas próprias características e comportamentos.



Um dos principais conceitos da POO é a **encapsulação**, que consiste em **proteger as propriedades e métodos de um objeto de acesso externo não autorizado**. Isso permite que o objeto seja manipulado apenas através de uma interface definida pela classe.

Outros conceitos importantes da POO incluem a **herança**, que permite que **uma classe herde características de outra classe**; e o **polimorfismo**, que permite que **objetos de diferentes classes possam ser tratados de forma semelhante**.



Lição 3: Conhecendo a linguagem UML e modelando nosso projeto

A **Linguagem de Modelagem Unificada** (*Unified Modeling Language*, UML) é uma **linguagem visual de modelagem** amplamente utilizada no desenvolvimento de software. A UML foi criada para ser uma linguagem padrão para a especificação, visualização, construção e documentação de artefatos de software orientado a objetos.

A UML fornece aos desenvolvedores uma linguagem comum para modelar sistemas, permitindo que eles **criem diagramas e modelos** que possam ser compartilhados entre diferentes

membros da equipe de desenvolvimento, bem como com outras partes interessadas no projeto, como clientes e gerentes de projeto.

A UML possui uma ampla gama de diagramas que permitem que os desenvolvedores representem diferentes aspectos de um sistema, incluindo diagramas de casos de uso, diagramas de classes, diagramas de sequência, diagramas de atividade, diagramas de componentes e muito mais.

A UML é uma ferramenta valiosa para ajudar os desenvolvedores a visualizar e comunicar a arquitetura e o design de um sistema, bem como para documentar os requisitos e especificações do sistema. A UML também pode ser usada como uma ferramenta para ajudar na análise de requisitos e na validação do design do sistema.

Lição 4: O conceito de classes no JavaScript

Em JavaScript, as **classes** são uma forma de **criar objetos que compartilham propriedades e métodos**. Uma classe é como um modelo para um objeto e é definida usando a palavra-chave **class**.

A sintaxe básica para criar uma classe em JavaScript é a seguinte:

```
class NomeDaClasse {
  constructor() {
    // código do construtor
  }

  método1() {
    // código do método 1
  }

  método2() {
    // código do método 2
  }

  // outros métodos e propriedades
}
```

O **construtor** é um método especial que é executado automaticamente quando um objeto é criado a partir da classe. Ele é usado para **inicializar as propriedades do objeto**. Para criar uma nova instância da classe, você pode usar a palavra-chave **new** seguida do nome da classe, e quaisquer argumentos necessários para o construtor:

```
const obj = new NomeDaClasse(arg1, arg2);
```

Os métodos definidos na classe são compartilhados por todas as instâncias da classe. Dentro de um método, você pode se referir ao objeto atual usando a palavra-chave **this**. Veja um exemplo de uma classe em JavaScript com um construtor e alguns métodos:

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }

  saudacao() {
    console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);
  }

  aniversario() {
    this.idade++;
    console.log(`Feliz aniversário! Agora eu tenho ${this.idade} anos.`);
  }
}

const pessoa1 = new Pessoa("João", 25);
pessoa1.saudacao(); // Olá, meu nome é João e eu tenho 25 anos.
pessoa1.aniversario(); // Feliz aniversário! Agora eu tenho 26 anos.
```

Neste exemplo, criamos uma classe **Pessoa** com um construtor que recebe um **nome** e uma **idade**. A classe também tem dois métodos: **saudacao**, que imprime uma mensagem de saudação com o nome e a idade da pessoa, e **aniversario**, que aumenta a idade da pessoa em 1 e imprime uma mensagem de feliz aniversário. Criamos uma instância da classe chamada pessoa1 e usamos os métodos para interagir com o objeto.

Lição 5: Herança de classes – Parte 1

Em JavaScript, é possível criar uma hierarquia de classes utilizando a **herança**. Para isso, uma classe pode ser definida como filha de outra classe, **herdando assim seus métodos e propriedades**. A herança é implementada utilizando a palavra-chave **extends**, que indica qual classe será utilizada como classe pai.

A seguir, temos um exemplo de uma classe **Animal** que é a classe **pai**, e a classe **Cachorro** que é **filha** da classe Animal:

```
class Animal {
  constructor(nome) {
    this.nome = nome;
  }

  falar() {
    console.log(`${this.nome} emite um som.`);
  }
}

class Cachorro extends Animal {
  falar() {
    console.log(`${this.nome} latindo.`);
  }
}

let c = new Cachorro('Rex');
c.falar(); // Saída: Rex latindo.
```

No exemplo acima, a classe **Cachorro** é definida utilizando a palavra-chave **extends**, indicando que ela é uma classe filha da classe **Animal**. A classe **Cachorro herda o construtor e o método falar() da classe Animal**.

A classe **Cachorro** redefine o método **falar()** e adiciona um comportamento específico para essa classe. Dessa forma, quando o método **falar()** é chamado para um objeto da classe **Cachorro**, o comportamento definido na classe Cachorro será executado.

A herança em JavaScript permite a reutilização de código e facilita a manutenção do sistema, pois as mudanças realizadas na classe pai afetarão automaticamente as classes filhas que a herdam.

Lição 6: Herança de classes – Parte 2

Os módulos em JavaScript são uma forma de **organizar e reutilizar código**, permitindo que diferentes partes do código sejam definidas em arquivos separados e importados onde necessário. Os módulos ajudam a manter o código organizado, evitando o acoplamento excessivo e permitindo que o código seja dividido em partes reutilizáveis.

Antes da especificação do ECMAScript 6 (ES6), os módulos não eram uma funcionalidade nativa do JavaScript e, portanto, os desenvolvedores precisavam usar bibliotecas de terceiros ou técnicas de nomenclatura para simular o comportamento dos módulos. Com a introdução do ES6, o JavaScript introduziu a funcionalidade de módulos nativos.

Existem duas maneiras principais de exportar e importar módulos em JavaScript:

1. Exportação e importação padrão (default);
2. Exportação e importação nomeada.

Exportação e importação padrão (default)

Com a exportação padrão, apenas uma única parte do código pode ser exportada de um arquivo. O valor padrão é declarado usando a palavra-chave **export default**. A importação padrão pode ser feita sem a necessidade de usar chaves {} e o nome da variável importada pode ser qualquer nome válido em JavaScript.

Exemplo de exportação padrão:

```
// no arquivo modulo.js
export default function soma(a, b) {
  return a + b;
}
```

Exemplo de importação padrão:

```
// no arquivo principal.js
import soma from './modulo.js';

console.log(soma(2, 3)); // saída: 5
```

Exportação e importação nomeada

A exportação nomeada permite que várias partes do código sejam exportadas de um arquivo, e a importação nomeada requer que as chaves {} sejam usadas e o nome da variável importada precisa corresponder exatamente ao nome da variável exportada.

Exemplo de exportação nomeada:

```
// no arquivo modulo.js
export const nome = 'João';
export function soma(a, b) {
  return a + b;
}
```

Exemplo de importação nomeada:

```
// no arquivo principal.js
import { nome, soma } from './modulo.js';

console.log(nome); // saída: "João"
console.log(soma(2, 3)); // saída: 5
```

Lição 7: Como tornar seus atributos privado

Em ES6 (ECMAScript 2015) e versões posteriores, é possível utilizar a sintaxe de classes para criar atributos privados em JavaScript. Uma das formas de tornar um atributo privado em uma classe é utilizando a notação de # antes do nome do atributo. Por exemplo:

```
class Exemplo {
  #atributoPrivado = 'valor';

  getValorAtributoPrivado() {
    return this.#atributoPrivado;
  }

  setValorAtributoPrivado(novoValor) {
    this.#atributoPrivado = novoValor;
  }
}

const exemplo = new Exemplo();

console.log(exemplo.getValorAtributoPrivado()); // retorna 'valor'

exemplo.setValorAtributoPrivado('novo valor');

console.log(exemplo.getValorAtributoPrivado()); // retorna 'novo valor'
```

Nesse exemplo, a classe **Exemplo** possui um atributo privado **#atributoPrivado** que só pode ser acessado pelos métodos da própria classe, nesse caso **getValorAtributoPrivado()** e **setValorAtributoPrivado(novoValor)**. Entenderemos um pouco mais sobre essas formas de acessar atributos privados na próxima lição.

Lição 8: Propriedades acessors – os getters e setters

Propriedades **acessoras (getters e setters)** são formas de definir métodos para acessar e modificar um valor de um objeto em JavaScript. Eles permitem que você controle como um objeto é modificado e acessado, permitindo a validação dos valores atribuídos e a execução de outras ações quando um valor é acessado ou definido.

Os getters e setters são definidos por meio de uma propriedade de objeto e são usados para controlar o acesso a outras propriedades do objeto. Eles podem ser definidos tanto em objetos literais quanto em funções construtoras.

Um **getter** é um método que é usado para **retornar o valor de uma propriedade privada**, enquanto um **setter** é um método que é usado **para definir o valor de uma propriedade privada**. Quando um getter é acessado, ele executa o código dentro dele e retorna o valor desejado. Quando um setter é acessado, ele executa o código dentro dele e define um novo valor para a propriedade privada.

Veja um exemplo de como definir um getter e um setter em um objeto:

```
const pessoa = {
  nome: 'João',
  idade: 30,
  get nomeCompleto() {
    return this.nome + ' Silva';
  },
  set nomeCompleto(valor) {
    const nomeSeparado = valor.split(' ');
    this.nome = nomeSeparado[0];
  }
};

console.log(pessoa.nomeCompleto); // 'João Silva'

pessoa.nomeCompleto = 'Maria Silva';

console.log(pessoa.nome); // 'Maria'
console.log(pessoa.nomeCompleto); // 'Maria Silva'
```

Nesse exemplo, a propriedade **nomeCompleto** é definida como um **getter** e um **setter**. Quando a propriedade **nomeCompleto** é acessada, o **getter** é executado, que retorna o valor do nome com o sobrenome 'Silva' adicionado. Quando a propriedade **nomeCompleto** é definida, o **setter** é executado, que define o valor do nome baseado no valor passado.

Lição 9: Conhecendo mais sobre polimorfismo

O **polimorfismo** é uma das características da Programação Orientada a Objetos (POO) e se refere à habilidade de um **objeto poder ser referenciado de diferentes formas**. Em JavaScript, é possível alcançar polimorfismo através de herança e da capacidade de uma função manipular objetos de diferentes tipos.

A herança permite que objetos filhos herdem características e comportamentos de objetos pais, podendo assim ser referenciados como o objeto pai ou como o objeto filho. Dessa forma, é possível criar um único método que trabalhe com objetos de diferentes tipos, desde que eles tenham as mesmas características e comportamentos herdados da mesma classe pai.

Além da herança, é possível alcançar o polimorfismo em JavaScript através da capacidade de uma função manipular objetos de diferentes tipos. Por exemplo, se temos duas classes diferentes que implementam um mesmo método, é possível criar uma função que recebe como parâmetro objetos dessas classes e realizar a operação desejada em ambos, sem a necessidade de criar funções diferentes para cada classe.

Vamos ver um exemplo simples de polimorfismo em JavaScript:

```
class Animal {
  constructor(nome) {
    this.nome = nome;
  }

  fazerSom() {
    console.log("O animal faz algum som.");
  }
}

class Cachorro extends Animal {
  fazerSom() {
    console.log("O cachorro late.");
  }
}

class Gato extends Animal {
  fazerSom() {
    console.log("O gato mia.");
  }
}

function fazerAnimalFazerSom(animal) {
  animal.fazerSom();
}

const cachorro = new Cachorro("Rex");
const gato = new Gato("Mittens");

fazerAnimalFazerSom(cachorro); // Output: "O cachorro late."
fazerAnimalFazerSom(gato); // Output: "O gato mia."
```

Lição 10: Princípios SOLID – Single Responsability Principle

Os princípios SOLID são um conjunto de cinco princípios que foram definidos para ajudar a **desenvolver softwares mais limpo, modular e escalável**. Eles foram introduzidos por Robert C. Martin, também conhecido como Uncle Bob, em seu livro "Agile Software Development, Principles, Patterns, and Practices".

Os princípios SOLID são:

Esses princípios ajudam a criar sistemas que são mais fáceis de manter, estender e testar, além de serem mais robustos e flexíveis.

- ***Single Responsibility Principle*** **(Princípio da Responsabilidade Única):** uma classe deve ter apenas uma razão para mudar, ou seja, ela deve ter apenas uma responsabilidade. Isso significa que uma classe deve ter apenas um motivo para ser alterada, e isso é importante para que as classes sejam mais fáceis de entender e manter.
- ***Open/Closed Principle*** **(Princípio Aberto/Fechado):** uma classe deve estar aberta para extensão, mas fechada para modificação. Isso significa que uma classe deve permitir a adição de novas funcionalidades sem precisar ser modificada. Esse princípio incentiva o uso de herança, polimorfismo e interfaces para criar sistemas mais flexíveis.
- ***Liskov Substitution Principle*** **(Princípio da Substituição de Liskov):** uma subclasse deve ser substituível por sua classe base. Isso significa que uma classe derivada deve ser capaz de substituir sua classe base sem afetar o comportamento do programa. Esse princípio é importante para garantir a consistência do sistema e evitar comportamentos inesperados.
- ***Interface Segregation Principle*** **(Princípio da Segregação de Interfaces):** clientes não devem ser forçados a depender de interfaces que eles não utilizam. Isso significa que uma interface deve ser focada em uma única responsabilidade e não deve ter métodos que não são

relevantes para a sua finalidade. Esse princípio ajuda a criar sistemas mais coesos e evita acoplamento desnecessário.

- ***Dependency Inversion Principle*** **(Princípio da Inversão de Dependência):** módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações. Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações. Isso significa que as classes devem depender de interfaces e não de implementações concretas. Esse princípio ajuda a criar sistemas mais flexíveis e fáceis de testar.

Single Responsability Principle

O princípio da Responsabilidade Única (Single Responsibility Principle, ou SRP) é um dos princípios SOLID da **programação orientada a objetos**. Ele afirma que uma classe deve ter apenas uma responsabilidade, ou seja, uma razão para mudar.

Isso significa que uma classe deve ter um único propósito e só deve lidar com uma única responsabilidade. Se uma classe tiver várias responsabilidades, ela se torna mais difícil de manter e mais propensa a erros e bugs.

O SRP é importante porque promove a **modularidade e a coesão** em um sistema. Se cada classe tiver uma única responsabilidade, é mais fácil fazer alterações em uma parte do sistema sem afetar outras partes, e é mais fácil entender como as partes do sistema se relacionam entre si.

Para seguir o SRP, é importante analisar as responsabilidades da classe e avaliar se ela está fazendo mais do que deveria. Se a classe estiver fazendo várias coisas diferentes, pode ser necessário dividi-la em classes menores e mais específicas.

Lição 11: Desafio Tech

Acompanhe nossa lição prática em vídeo.

Lição 12: 5 passos práticos para aplicar o que você aprendeu

1. Entenda o que é **programação orientada a objetos**;
2. Conheça as **diretrizes importantes** que abrangem esse paradigma de programação;
3. Entenda como **as diretrizes são utilizadas** no contexto do JavaScript;
4. Use as **boas práticas** de programação;
5. Aplique todos os **conhecimentos aprendidos** para fixação.

Módulo 5
JavaScript para desenvolvimento web

Lição 1: Objetivos do módulo

Neste último módulo começaremos a aplicar JavaScript no contexto de uma **página web**. Em uma página já construída com HTML e CSS, vamos utilizar o JavaScript para manipular essa página e deixá-la **mais interativa**.

Responderemos perguntas do tipo:

- ✓ Como manipular os elementos da sua página?
- ✓ O que são eventos?
- ✓ Como utilizar e validar formulários?

Lição 2: Introduzindo conceitos – Olá mundo com JavaScript

O que é DOM?

DOM é a sigla para ***Document Object Model***, que em português significa **Modelo de Objetos de Documento**. É uma representação em forma de árvore de todos os elementos HTML de uma página web, que podem ser acessados e manipulados através de uma interface de programação de aplicações (API) em JavaScript.

O DOM é uma interface de programação que **permite que os desenvolvedores acessem e manipulem o conteúdo**, **estrutura** e **estilo de uma página web**. Isso significa que podemos usar o JavaScript para adicionar, alterar ou remover elementos e atributos HTML, estilos CSS e interagir com eventos do usuário.

O DOM é uma especificação padrão que define como os navegadores devem construir a árvore de objetos que representam a página web. Através do DOM, é possível navegar pela hierarquia dos elementos HTML, acessar suas propriedades e métodos, e fazer alterações dinâmicas no conteúdo e estilo da página.

Inserindo um script no HTML

Para inserir um script no HTML, você pode utilizar a tag **<script>**. Existem duas formas principais de fazer isso:

1. Incluir o script diretamente na página HTML:

```
<html>
  <head>
    <title>Minha Página</title>
    <script>
      // Seu script aqui
    </script>
  </head>
  <body>
    <!-- Conteúdo da página aqui -->
  </body>
</html>
```

Neste caso, o script será executado assim que o navegador carregar a página.

2. Incluir o script em um arquivo externo:

```
<html>
  <head>
    <title>Minha Página</title>
    <script src="meu_script.js"></script>
  </head>
  <body>
    <!-- Conteúdo da página aqui -->
  </body>
</html>
```

Neste caso, o arquivo meu_script.js deve estar localizado no mesmo diretório do arquivo HTML e conter o código JavaScript que você deseja executar.

Ambas as opções permitem que você execute scripts em suas páginas HTML, **mas a segunda opção é geralmente preferível**, pois permite que você mantenha o **código JavaScript em arquivos separados**, facilitando a organização e manutenção do seu código.

Lição 3: Utilizando o querySelector para selecionar elementos

Antes de começarmos a deixar nossa página interativa, precisamos entender sobre o **document**. Esse objeto possui uma série enorme de propriedades e métodos que são muito úteis para nós

Conhecendo o **document**

O objeto **document** é um objeto global no navegador que representa o documento atual que está sendo exibido na janela do navegador. Ele é criado automaticamente pelo navegador assim que a página HTML é carregada.

O objeto document é usado para **acessar e manipular elementos HTML** e suas propriedades e métodos. Por exemplo, podemos usar o document para selecionar elementos HTML usando o método **querySelector()**, para criar novos elementos HTML usando o método **createElement()**, para modificar o conteúdo de um elemento HTML usando a propriedade **innerHTML** e para adicionar ou remover classes CSS de um elemento HTML usando a propriedade **classList**.

Aqui estão alguns exemplos de como usar o objeto document:

- Selecionar um elemento HTML usando **querySelector()**

```
const meuElemento = document.querySelector('#meuId');
```

- Criar um novo elemento HTML usando **createElement()**

```
const novoElemento = document.createElement('div');
```

- Modificar o conteúdo de um elemento HTML usando **innerHTML**

```
meuElemento.innerHTML = '<h1>Novo título</h1>';
```

- Adicionar ou remover classes CSS de um elemento HTML usando **classList**

```
meuElemento.classList.add('ativo');
meuElemento.classList.remove('inativo');
```

Em resumo, o objeto document é um objeto muito importante no desenvolvimento de aplicações web em JavaScript, pois permite interagir com o conteúdo HTML de uma página web e manipular dinamicamente sua aparência e comportamento.

querySelector e querySelectorAll

O método **querySelector** é uma função nativa do JavaScript que permite selecionar elementos HTML em uma página da web usando seletores CSS. Ele **retorna o primeiro elemento correspondente ao seletor fornecido**.

O querySelector recebe um argumento que é um seletor CSS para selecionar um elemento. Por exemplo, se você deseja selecionar um elemento com uma classe "destaque", pode usar **querySelector(".destaque")**. O ponto antes do nome da classe indica que você está selecionando uma classe.

Além do querySelector, há também o método **querySelectorAll**, que é semelhante, mas **retorna todos os elementos correspondentes ao seletor em uma coleção NodeList**. É possível percorrer esses elementos com um loop e manipulá-los de acordo.

Aqui está um exemplo de uso do querySelector:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo de QuerySelector</title>
  </head>
  <body>
    <h1 class="titulo">Título da página</h1>
    <p class="paragrafo">Este é um parágrafo de exemplo.</p>

    <script>
      const titulo = document.querySelector(".titulo");
      console.log(titulo.textContent); // "Título da página"

      const paragrafo = document.querySelector(".paragrafo");
      paragrafo.textContent = "Este é um novo parágrafo.";
    </script>
  </body>
</html>
```

Neste exemplo, o querySelector é usado para selecionar o elemento **<h1>** com a classe **"titulo"** e o seu conteúdo é exibido no console. Em seguida, o querySelector é usado novamente para selecionar o elemento **<p>** com a classe **"paragrafo"** e seu conteúdo é atualizado para **"Este é um novo parágrafo"**.

Lição 4: Manipulação de elementos HTML com JavaScript

O **innerHTML** é uma propriedade do objeto Element no JavaScript que permite acessar e modificar o conteúdo HTML dentro de um elemento. Quando você usa o innerHTML, pode **recuperar o conteúdo HTML de um elemento**, bem como **substituir o conteúdo** existente por um novo conteúdo HTML.

Para acessar o conteúdo HTML de um elemento usando o innerHTML, você pode fazer o seguinte:

```
const element = document.querySelector('#meu-elemento');
const conteudoHtml = element.innerHTML;
console.log(conteudoHtml);
```

O innerHTML pode ser usado para alterar o conteúdo HTML de um elemento. Por exemplo, para alterar o conteúdo de um elemento **<p>** para **"Olá, Mundo!"**, você pode fazer o seguinte:

```
const element = document.querySelector('#meu-elemento');
element.innerHTML = 'Olá, Mundo!';
```

O innerHTML também pode ser usado para **adicionar novos elementos HTML** a um elemento existente. Por exemplo, para adicionar um novo elemento **<li>** a uma lista não ordenada **<ul>**, você pode fazer o seguinte:

```
const ul = document.querySelector('#minha-lista');
ul.innerHTML += '<li>Novo item</li>';
```

É importante lembrar que a propriedade innerHTML pode ser usada para injetar conteúdo HTML na página, mas também pode apresentar problemas de segurança se o conteúdo injetado vier de uma fonte não confiável. Portanto, é importante garantir que o conteúdo injetado seja seguro e verificado antes de ser exibido.

Lição 5: Eventos e formulários – adicionando alunos

Existem duas maneiras principais de adicionar eventos a elementos HTML: via HTML ou via JavaScript.

Adicionando eventos via HTML

Para adicionar um evento a um elemento HTML via HTML, você pode usar o atributo **"on"** seguido do nome do evento e do código a ser executado quando o evento ocorrer. Por exemplo, se você quiser executar uma função chamada **"myFunction"** quando um botão for clicado, você pode usar o seguinte código HTML:

```
<button onclick="myFunction()">Clique aqui</button>
```

Neste exemplo, **"onclick"** é o nome do evento que ocorre quando o botão é clicado, e **"myFunction()"** é o código a ser executado quando o evento ocorrer.

Adicionando eventos via JavaScript

Para adicionar eventos a um elemento HTML via JavaScript, você pode usar o método **"addEventListener"**. Este método recebe dois parâmetros: o **nome do evento** a ser adicionado e

a função a ser executada quando o evento ocorrer. Por exemplo, se você quiser executar uma função chamada "myFunction" quando um botão for clicado, você pode usar o seguinte código JavaScript:

```
const myButton = document.querySelector('button');
myButton.addEventListener('click', myFunction);
```

Neste exemplo, **"document.querySelector"** é usado para selecionar o botão que será clicado. Em seguida, **"addEventListener"** é usado para adicionar um **"click"** ao botão e chamar a função **"myFunction"** quando o evento ocorrer.

Lista de eventos

Segue abaixo uma lista de alguns eventos em JavaScript:

- **click**: evento disparado quando um elemento é clicado.
- **submit**: evento disparado quando um formulário é enviado.
- **keydown**: evento disparado quando uma tecla do teclado é pressionada.
- **load**: evento disparado quando um objeto é carregado.
- **change**: evento disparado quando o valor de um elemento é alterado.
- **mouseover**: evento disparado quando o mouse é colocado sobre um elemento.
- **mouseout**: evento disparado quando o mouse é retirado de um elemento.
- **focus**: evento disparado quando um elemento ganha o foco.
- **blur**: evento disparado quando um elemento perde o foco.
- **resize**: evento disparado quando a janela do navegador é redimensionada.

Esses são apenas alguns exemplos, existem muitos outros eventos disponíveis em JavaScript que podem ser utilizados em diferentes situações.

Lição 6: Validações de formulários

Acompanhe nossa lição prática em vídeo.

Lição 7: Como remover alunos da sua tabela

Acompanhe nossa lição prática em vídeo.

Lição 8: Filtrando uma tabela – expressão regular

Acompanhe nossa lição prática em vídeo.

Lição 9: Desafio Tech

Acompanhe nossa lição prática em vídeo.

Lição 10: 5 passos práticos para aplicar o que você aprendeu

1. Entenda **o que é o DOM** e qual a sua **importância para criar aplicações dinâmicas** e interativas;
2. Aprenda a acessar e manipular **os elementos** de sua página;
3. Faça bom uso de eventos para adicionar funcionalidades na página;
4. Entenda como utilizar e validar **formulários**;
5. Aplique todos os **conhecimentos aprendidos** para fixação.