Práticas recomendadas em JS

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
==	===	Para comparações rigorosas em JavaScript	Comparações rigorosas	Evitar erros de tipo nas comparações em JavaScript	Quando não há necessidade de comparações rigorosas
alert()	Custom modal dialogs	Para melhorar a usabilidade e a experiência do usuário	Mensagens e interações de usuário	Melhorar a experiência do usuário com interfaces modernas	Quando não há necessidade de mensagens ou interações de usuário
arr.concat(arr2);	const merged = [...arr, ...arr2];	Para mesclar arrays	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de mesclar arrays
arr.filter(Boolean);	arr.filter(el => el);	Para remover valores falsy de arrays	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de remover valores falsy
arr.map(el => el * 2);	arr.map(el => el * 2).filter(el => el > 2);	Para transformar e filtrar arrays	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de transformar e filtrar arrays
arr[-1];	arr.at(-1);	Para acessar elementos de arrays com índices negativos	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de acessar elementos de arrays com índices negativos
array.flat();	const flatArray = array.flat();	Para aplanar arrays aninhados	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de aplanar arrays
array.flatMap(x => [x * 2]);	const flatMappedArray = array.flatMap(x => [x, x * 2]);	Para transformar e aplanar arrays	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de transformar e aplanar arrays

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
Array.prototype.concat	Spread operator (...)	Para combinar arrays de forma mais eficiente e legível	Combinar objetos e arrays	Sintaxe mais concisa e legível para combinar objetos e arrays	Quando não há necessidade de combinar arrays
Array.prototype.findIndex	Array.prototype.find	Para encontrar o primeiro item que satisfaça uma condição de forma mais eficiente e legível	Encontrar item em um array	Forma mais eficiente e legível de encontrar o primeiro item que satisfaça uma condição	Quando não há necessidade de encontrar itens em um array
Array.prototype.map	Array.from	Para criar arrays a partir de objetos iteráveis ou pseudo-arrays	Criação de arrays a partir de objetos iteráveis	Melhor suporte a objetos pseudo-arrays e iteráveis	Quando não há necessidade de criar arrays a partir de objetos iteráveis
Array.prototype.reduce para somar valores	Array.prototype.reduce com arrow function	Para melhorar a clareza e concisão ao usar a função de redução	Soma de valores em arrays	Forma mais clara e concisa de usar a função de redução	Quando não há necessidade de somar valores em arrays
Array.prototype.splice para remover itens	Array.prototype.filter	Para remover itens de um array de forma mais clara e imutável	Remover itens de um array	Forma mais clara e imutável de remover itens de um array	Quando não há necessidade de remover itens de um array
Atomics.wait(typedArray, index, value);	Atomics.waitSync(typedArray, index, value);	Para operações de espera síncrona em ambientes de multithreading	Sincronização de threads	Melhor controle sobre a sincronização	Quando não há necessidade de sincronização de threads
Bibliotecas de UI (Bootstrap, Material-UI, etc.) com sobrescrita de estilos	Sistemas de design (Design Systems) com componentes reutilizáveis	Para manter consistência visual e facilitar a manutenção em grandes aplicativos	Aplicativos web de grande escala com múltiplas equipes	Melhorar a consistência, reutilização e manutenibilidade da interface do usuário	Quando o aplicativo é pequeno ou não requer uma identidade visual forte
Callback hell	Promises ou async/await	Para evitar aninhamento excessivo de callbacks e melhorar a legibilidade do código	Operações assíncronas complexas	Melhor legibilidade e manejo de fluxos assíncronos	Quando não há operações assíncronas a serem realizadas
Callbacks	Promises ou async/await	Para lidar com operações assíncronas de forma mais	Operações assíncronas	Melhor legibilidade e manejo de fluxos assíncronos	Quando não há operações

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		legível e gerenciável	complexas		assíncronas a serem realizadas
Callbacks aninhados (callback hell)	Async/await com tratamento de erros try/catch	Para simplificar e melhorar a legibilidade do código assíncrono	Funções assíncronas complexas	Evitar o aninhamento excessivo de callbacks e melhorar a clareza do fluxo de controle	Quando não há operações assíncronas complexas
class MyClass { constructor() { this.myField = 'value'; } }	class MyClass { myField = 'value'; }	Para definir campos de classe de forma mais concisa	Declaração de classes	Sintaxe mais clara e eficiente	Quando não há necessidade de definir campos de classe
class MyClass { constructor() { this.myField = 'value'; } }	class MyClass { static myField = 'value'; }	Para definir campos estáticos de classe	Declaração de classes	Melhor organização e acessibilidade de dados estáticos	Quando não há necessidade de definir campos estáticos
console.log	Logging frameworks	Para um logging mais robusto e gerenciável em produção	Aplicações em produção	Logging mais robusto e gerenciável	Quando não há necessidade de logging em produção
const arr = [1, 2, 3]; const newArr = arr.map(x => x * 2);	const newArr = arr.flatMap(x => [x, x * 2]);	Para manipular arrays de forma eficiente	Manipulação de arrays	Sintaxe mais eficiente e legível	Quando não há necessidade de manipular arrays
const arr = [1, 2, 3]; const newArr = arr.map(x => x * 2);	const newArr = arr.flatMap(x => [x, x * 2]);	Para transformar e aplanar arrays	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de transformar e aplanar arrays
const bigInt = BigInt(1234567890);	const bigInt = 1234567890n;	Para criar BigInts	Manipulação de números grandes	Sintaxe mais clara e eficiente	Quando não há necessidade de manipular números grandes
const data = { name: 'Alice', age: 25 }; const name = data.name; const age = data.age;	const { name, age } = data;	Para extrair valores de objetos de forma concisa	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de extrair valores
const entries = Object.keys(obj).map(key	const entries = Object.entries(obj);	Para obter pares chave-valor	Manipulação de	Sintaxe mais clara e	Quando não há

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
=> [key, obj[key]]);		de um objeto	objetos	eficiente	necessidade de obter pares chave-valor de objetos
const global = Function('return this')();	globalThis;	Para acessar o objeto global	Manipulação de escopo global	Sintaxe mais clara e eficiente	Quando não há necessidade de acessar o objeto global
const merged = Object.assign({}, obj1, obj2);	const merged = { ...obj1, ...obj2 };	Para mesclar objetos	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de mesclar objetos
const name = person.name; const age = person.age;	const { name, age } = person;	Para desestruturar objetos	Desestruturação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de desestruturar objetos
const newArr = []; arr.forEach(item => newArr.push(item * 2));	const newArr = arr.map(item => item * 2);	Para transformar elementos de um array	Transformação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de transformar arrays
const newObj = Object.assign({}, obj1, obj2);	const newObj = { ...obj1, ...obj2 };	Para clonar ou combinar objetos de forma eficiente	Manipulação de objetos, especialmente em funções de utilidade	Sintaxe mais clara e eficiente	Quando não há necessidade de clonar ou combinar objetos
const now = new Date();	const now = Temporal.Now.plainDateTimeISO();	Para manipulação moderna de datas e horas	Manipulação de datas e horas	API mais robusta e coerente	Quando não há necessidade de manipulação de datas e horas
const numbers = [1, 2, 3]; const reversed = numbers.reverse();	const reversed = numbers.toReversed();	Para reverter arrays sem modificar o original	Manipulação de arrays	Evitar mutação do array original	Quando não há necessidade de reverter arrays
const obj = Object.assign({}, a, b);	const obj = { ...a, ...b };	Para mesclar objetos	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de mesclar objetos
const obj = { a: 1, b: 2 }; const copy = Object.assign({}, obj);	const copy = { ...obj };	Para copiar propriedades de objetos	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
					copiar objetos
const obj = {}; if (!obj.prop) { obj.prop = 'default'; }	obj.prop		= 'default';	Para atribuir valores padrão	Atribuição de valores
const obj = {}; if (obj.a) { obj.a(); }	obj.a?.();	Para evitar erros de acesso a propriedades	Manipulação de objetos	Sintaxe mais clara e segura	Quando não há necessidade de acessar propriedades
const path = 'C:\\path\\to\\file';	const path = String.rawC:\\path\\to\\file;	Para evitar processamento de escape sequences	Strings de caminho de arquivo, regex complexas	Evita erros de escape e melhora a legibilidade	Quando não há necessidade de strings não processadas
const regex = /[a-zA-Z]/;	const regex = /\p{L}/u;	Para correspondência de caracteres com propriedades Unicode	Manipulação de strings	Aumenta a precisão na correspondência de padrões	Quando não há necessidade de correspondência de padrões complexos
const regex = new RegExp('pattern', 'flags');	const regex = /pattern/flags;	Para buscas avançadas em strings e correspondências complexas	Processamento de strings, validação de entradas	Poderoso e flexível para correspondência de padrões	Quando simples correspondências de padrões são suficientes
const response = await fetch(url); const data = await response.json();	const data = await fetch(url).then(response => response.json());	Para simplificar chamadas de API assíncronas	Requisições HTTP assíncronas	Sintaxe mais clara e eficiente	Quando não há necessidade de fazer requisições HTTP
const str = ' Hello World '; str.trim();	str.trimStart(); str.trimEnd();	Para remover espaços em branco do início e do fim de strings	Manipulação de strings	Sintaxe mais clara e eficiente	Quando não há necessidade de remover espaços em branco
const str = '\uD800'; const isWellFormed = /^[^\uD800-\uDFFF]$/u.test(str);	const isWellFormed = str.isWellFormed();	Para verificar se uma string está bem formada	Manipulação de strings	Sintaxe mais clara e eficiente	Quando não há necessidade de verificar strings
const str = 'abc'; const regex = /a/; const result = str.match(regex);	const result = 'abc'.match(/a/);	Para simplificar a busca de padrões em strings	Manipulação de strings	Sintaxe mais clara e eficiente	Quando não há necessidade de buscar padrões

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
const str = 'hello'; const chars = str.split('');	const chars = [...str];	Para dividir uma string em caracteres	Manipulação de strings	Sintaxe mais clara e eficiente	Quando não há necessidade de dividir strings em caracteres
const sym = Symbol('description');	const sym = Symbol.for('description');	Para compartilhar símbolos entre diferentes partes do código	Manipulação de símbolos	Reutilização de símbolos	Quando não há necessidade de compartilhar símbolos
const val = a		b;	const val = a ?? b;	Para atribuir valores padrão	Atribuição de valores
const value = (obj && obj.value)		'default';	const value = obj?.value ?? 'default';	Para atribuir valores padrão de forma mais segura	Atribuição de valores
const values = Object.keys(obj).map(key => obj[key]);	const values = Object.values(obj);	Para obter valores de um objeto	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de obter valores de objetos
const x = Math.max(a, b, c);	const x = Math.max(...[a, b, c]);	Para encontrar o valor máximo em um array	Manipulação de arrays	Sintaxe mais clara e eficiente	Quando não há necessidade de encontrar o valor máximo
CSS em linha	CSS-in-JS ou CSS Modules	Para estilos com escopo e melhor integração com JavaScript	Aplicativos web modernos com componentização	Melhor gerenciamento de estilos, escopo e dinâmica com JavaScript	Quando os estilos são simples e não requerem escopo ou dinamicidade
document.body.scrollTop	window.scrollY	Para obter a posição de rolagem vertical de forma mais precisa e moderna	Obter posição de rolagem vertical	Forma mais precisa e moderna de obter a posição de rolagem vertical	Quando não há necessidade de obter a posição de rolagem vertical
document.cookie	localStorage ou sessionStorage	Para armazenar dados no cliente de forma mais segura e gerenciável	Armazenamento de dados no cliente	Armazenamento mais seguro e gerenciável de dados do cliente	Quando não há necessidade de armazenar dados no cliente
document.createElement e appendChild	Element.insertAdjacentHTML	Para inserir elementos HTML	Inserção de HTML	Inserção mais eficiente e	Quando não há

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		de forma mais direta e eficiente	dinâmico	segura de conteúdo HTML	necessidade de inserir HTML dinamicamente
document.createEvent	new Event ou CustomEvent	Para criar eventos personalizados de forma mais moderna e padronizada	Criação de eventos personalizados	Forma mais moderna e padronizada de criar eventos	Quando não há necessidade de criar eventos personalizados
document.execCommand('copy')	navigator.clipboard.writeText	Para copiar texto para a área de transferência de forma mais segura e moderna	Copiar texto para a área de transferência	Forma mais segura e moderna de copiar texto para a área de transferência	Quando não há necessidade de copiar texto para a área de transferência
document.forms['formName']	document.querySelector('form[name="formName"]')	Para selecionar formulários de forma mais moderna e flexível	Seleção de formulários	Seleção de formulários de forma mais moderna e flexível	Quando não há necessidade de selecionar formulários
document.getElementById	document.querySelector	Para selecionar elementos DOM com mais flexibilidade e consistência	Seleção de elementos DOM	Flexibilidade e consistência na seleção de elementos	Quando não há necessidade de selecionar elementos DOM
document.write	DOM manipulation methods	Para evitar práticas inseguras e obsoletas de manipulação do DOM	Atualizações dinâmicas no DOM	Evitar práticas obsoletas e inseguras como document.write	Quando não há necessidade de manipular o DOM
document.write('Hello World');	document.getElementById('example').textContent = 'Hello World';	Para evitar práticas inseguras e obsoletas de manipulação do DOM	Atualizações dinâmicas no DOM	Evitar práticas obsoletas e inseguras como document.write	Quando não há necessidade de manipular o DOM
element.attachEvent	element.addEventListener	Para adicionar eventos de forma mais padronizada e moderna	Adicionar eventos	Adição de eventos de forma padronizada e moderna	Quando não há necessidade de adicionar eventos
element.children	element.childNodes	Para obter todos os nós filhos, incluindo nós de texto e comentário	Obter todos os nós filhos	Forma mais abrangente de obter todos os nós filhos, incluindo nós de texto e comentário	Quando não há necessidade de obter todos os nós filhos

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
element.className = '...'	element.classList.add/remove/toggle	Para manipular classes de elementos de forma mais eficiente e flexível	Manipulação de classes de elementos	Sintaxe mais clara e eficiente para manipulação de classes	Quando não há necessidade de manipular classes de elementos
element.innerHTML += '...'	element.insertAdjacentHTML	Para adicionar HTML de forma mais eficiente e segura	Adicionar HTML dinâmico	Forma mais eficiente e segura de adicionar HTML ao DOM	Quando não há necessidade de adicionar HTML dinamicamente
Element.offsetTop e Element.offsetLeft	Element.getBoundingClientRect()	Para obter informações de posição e tamanho de elementos de forma mais precisa	Obter informações de posição e tamanho de elementos	Forma mais precisa e moderna de obter informações de posição e tamanho	Quando não há necessidade de obter informações de posição e tamanho
element.removeNode	element.remove	Para remover elementos do DOM de forma mais moderna e suportada	Remover elementos do DOM	Método moderno e suportado para remover elementos	Quando não há necessidade de remover elementos do DOM
element.scrollIntoView com opções padrão	element.scrollIntoView({ behavior: 'smooth' })	Para uma rolagem suave ao elemento	Rolagem suave ao elemento	Melhorar a experiência do usuário com rolagem suave	Quando não há necessidade de rolar até um elemento
element.setAttribute('style', '...')	element.style.property = '...'	Para manipular estilos inline de forma mais clara e controlada	Manipulação de estilos inline	Forma mais clara e controlada de manipular estilos	Quando não há necessidade de manipular estilos
element.setAttribute('style', 'color: red');	element.style.color = 'red';	Para manipular estilos inline de forma mais clara e controlada	Manipulação de estilos inline	Forma mais clara e controlada de manipular estilos	Quando não há necessidade de manipular estilos
eval()	Secure parsing functions	Para evitar vulnerabilidades de segurança	Avaliação de código dinamicamente	Evitar vulnerabilidades de segurança associadas ao uso de eval	Quando não há necessidade de avaliação de código dinamicamente
event.srcElement	event.target	Para acessar o elemento que disparou o evento de forma mais consistente	Acesso ao elemento que disparou o evento	Acesso consistente e confiável ao alvo do evento	Quando não há eventos sendo disparados

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
for (const value of asyncIterable) { await someFunction(value); }	for await (const value of asyncIterable) { await someFunction(value); }	Para iterar sobre dados assíncronos	Funções assíncronas que processam listas de dados	Simplifica o código e melhora a legibilidade ao lidar com promessas em loop	Quando não há necessidade de iteração assíncrona
for loops	for...of ou forEach	Para iterar sobre arrays de forma mais clara e moderna	Iterações sobre arrays	Sintaxe mais clara e moderna, melhorando a legibilidade	Quando não há necessidade de iterar sobre arrays
for...in	Object.keys/values/entries	Para iterar sobre as propriedades de objetos de forma mais segura	Iterar sobre propriedades de objetos	Iteração mais segura e controlada sobre propriedades de objetos	Quando não há necessidade de iterar sobre as propriedades de objetos
for...in para arrays	for...of	Para iterar sobre elementos de um array de forma mais segura e eficiente	Iterar sobre elementos de um array	Iteração mais segura e eficiente sobre arrays	Quando não há necessidade de iterar sobre elementos de um array
function	Arrow functions (=>)	Para funções anônimas e manter o contexto do this	Funções anônimas e callbacks	Manter o contexto de this e sintaxe mais concisa	Quando não há necessidade de criar funções dinâmicas
function getData() { return new Promise((resolve, reject) => { ... }); }	async function getData() { ... }	Para lidar com promessas de forma mais simples e legível	Funções assíncronas	Simplificação de promessas	Quando não há necessidade de operações assíncronas
function resolveAfter2Seconds(x) { return new Promise(resolve => setTimeout(() => resolve(x), 2000)); }	async function resolveAfter2Seconds(x) { return await new Promise(resolve => setTimeout(() => resolve(x), 2000)); }	Para lidar com operações assíncronas de forma mais legível e gerenciável	Operações assíncronas complexas	Melhor legibilidade e manejo de fluxos assíncronos	Quando não há operações assíncronas a serem realizadas
function sum(a, b) { return a + b; }	const sum = (a, b) => a + b;	Para funções anônimas e manter o contexto do this	Funções anônimas e callbacks	Manter o contexto de this e sintaxe mais concisa	Quando não há necessidade de criar funções dinâmicas
function.call(context, arg1, arg2)	function.apply(context, [arg1, arg2])	Para passar um array de	Passar arrays como	Sintaxe mais legível para	Quando não há

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		argumentos de forma mais eficiente e clara	argumentos	passar múltiplos argumentos	necessidade de passar múltiplos argumentos
Function.prototype.apply	Spread operator (...)	Para passar arrays como argumentos de funções de forma mais legível	Passar arrays como argumentos	Sintaxe mais legível para passar múltiplos argumentos	Quando não há necessidade de passar múltiplos argumentos
Function.prototype.bind	Arrow functions (=>)	Para preservar o contexto de this de forma mais concisa e legível	Preservar contexto de this	Sintaxe mais concisa e legível para funções anônimas	Quando não há necessidade de preservar o contexto de this
Funções IIFE para escopo	Módulos ES6 (import/export)	Para melhor modularização e reutilização de código, mantendo o escopo privado	Modularização e escopo privado	Melhor modularização e reutilização de código	Quando não há necessidade de modularização de código
getElementsByClassName	querySelectorAll	Para selecionar elementos DOM com mais flexibilidade e compatibilidade moderna	Seleção de elementos DOM	Seleção de elementos com mais flexibilidade e compatibilidade moderna	Quando não há necessidade de selecionar elementos DOM
HTMLElement.style.cssText	CSSStyleDeclaration (ex. element.style.property = 'value')	Para manipular estilos inline de forma mais controlada e eficiente	Manipulação de estilos inline	Forma mais controlada e eficiente de manipulação de estilos	Quando não há necessidade de manipular estilos inline
if (a == b) { ... }	if (a === b) { ... }	Para comparações rigorosas em JavaScript	Comparações rigorosas	Evitar erros de tipo nas comparações em JavaScript	Quando não há necessidade de comparações rigorosas
if (array.indexOf(element) !== -1) { ... }	if (array.includes(element)) { ... }	Para verificar se um array contém um elemento	Verificação de elementos em arrays	Sintaxe mais simples e legível	Quando não há necessidade de verificar elementos em arrays
if (obj && obj.prop) { ... }	if (obj?.prop) { ... }	Para verificar propriedades de objetos	Manipulação de objetos	Sintaxe mais clara e segura	Quando não há necessidade de verificar

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
					propriedades
if (obj.hasOwnProperty('key')) { ... }	if (Object.hasOwn(obj, 'key')) { ... }	Para verificar a propriedade de um objeto	Manipulação de objetos	Verificação mais segura de propriedades	Quando não há necessidade de verificar propriedades
if (obj.hasOwnProperty('prop')) { ... }	if (Object.prototype.hasOwnProperty.call(obj, 'prop')) { ... }	Para verificar a propriedade de um objeto	Manipulação de objetos	Verificação mais segura de propriedades	Quando não há necessidade de verificar propriedades de objetos
if (obj.hasOwnProperty('prop')) { ... }	if (Object.hasOwn(obj, 'prop')) { ... }	Para verificar a propriedade de um objeto de forma mais segura	Manipulação de objetos	Verificação mais segura de propriedades	Quando não há necessidade de verificar propriedades
if (str.startsWith('abc')) { ... }	str.padStart(10, '0');	Para preencher strings	Manipulação de strings	Melhor formatação de strings	Quando não há necessidade de preencher strings
Inline styles	CSS classes	Para separar estrutura e estilo, melhorar manutenção e performance	Aplicação de estilos	Separar estrutura HTML e estilo CSS, melhorar manutenção e performance	Quando não há necessidade de alterar estilos
innerHTML	textContent	Para inserir texto e evitar vulnerabilidades de XSS	Elementos de texto estático	Evitar vulnerabilidades de XSS ao inserir texto puro	Quando não há necessidade de inserir texto dinamicamente
JSON.parse(JSON.stringify(obj)) para clonar objetos	structuredClone(obj)	Para clonar objetos de forma mais eficiente e segura	Clonar objetos	Clonagem de objetos de forma eficiente e segura	Quando não há necessidade de clonar objetos
let xhr = new XMLHttpRequest();	fetch(url).then(response => response.json()).then(data => console.log(data));	Para fazer requisições assíncronas de forma mais simples e moderna	Requisições HTTP assíncronas	Simplicidade e promessa embutida para melhor gerenciamento de requisições	Quando não há necessidade de fazer requisições HTTP
localStorage	IndexedDB ou outras soluções de armazenamento modernas	Para armazenamento de dados do lado do cliente,	Aplicativos web que requerem	Melhor desempenho, capacidade e recursos para	Quando o armazenamento de

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		especialmente para grandes quantidades de dados estruturados	armazenamento local avançado	armazenamento local	dados é simples e limitado
Manipulação de classes com strings	Manipulação de classes com classList	Para adicionar, remover e alternar classes de forma mais eficiente e sem riscos de erros de digitação	Manipulação de classes	Adicionar, remover e alternar classes de forma mais eficiente e segura	Quando não há necessidade de manipular classes
Math.pow(2, 3);	2 ** 3;	Para realizar operações de potência	Cálculos matemáticos	Sintaxe mais concisa	Quando não há necessidade de cálculos exponenciais
Math.pow(2, 53) - 1;	Number.MAX_SAFE_INTEGER;	Para obter o maior número inteiro seguro	Manipulação de números	Sintaxe mais clara e eficiente	Quando não há necessidade de manipular números grandes
Math.pow(x, y);	x ** y;	Para realizar operações matemáticas de potência	Cálculos matemáticos	Sintaxe mais concisa e legível	Quando não há necessidade de cálculos exponenciais
Math.random()	Cryptographically secure methods	Para gerar números aleatórios seguros (ex. crypto.getRandomValues())	Geração de números aleatórios em segurança	Garantir segurança criptográfica, por exemplo, com crypto.getRandomValues()	Quando não há necessidade de geração de números aleatórios
Navegação por âncoras (#)	API History	Para navegação em SPAs (Single Page Applications)	Aplicativos web de página única	Melhor controle sobre o histórico de navegação e URLs mais limpas	Quando a navegação tradicional por âncoras é suficiente
navigator.getUserMedia	navigator.mediaDevices.getUserMedia	Para acessar a câmera e o microfone de forma mais segura e moderna	Acesso à câmera e microfone	Forma mais segura e moderna de acessar mídia do dispositivo	Quando não há necessidade de acessar a câmera e o microfone
new Date().getTime()	Date.now()	Para obter o timestamp atual de forma mais direta e eficiente	Obter timestamp atual	Forma mais direta e eficiente de obter o timestamp	Quando não há necessidade de obter o timestamp

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
					atual
new RegExp('pattern')	Literal de expressão regular (/pattern/)	Para criar expressões regulares de forma mais legível e eficiente	Criação de expressões regulares	Sintaxe mais legível e eficiente para criar expressões regulares	Quando não há necessidade de criar expressões regulares
Object.assign	Spread operator (...)	Para copiar propriedades de objetos de forma mais concisa e legível	Copiar e combinar objetos e arrays	Sintaxe mais concisa e legível para operações comuns	Quando não há necessidade de copiar ou combinar objetos e arrays
Object.entries(obj).forEach(([key, value]) => { obj[key] = value; });	Object.fromEntries(Object.entries(obj).map(([key, value]) => [key, value]));	Para transformar pares chave-valor em um objeto	Manipulação de objetos	Sintaxe mais clara e eficiente	Quando não há necessidade de transformar pares chave-valor
Object.getOwnPropertyDescriptors(obj);	Object.entries(obj).reduce((descriptors, [key, value]) => { descriptors[key] = { value }; return descriptors; }, {});	Para obter descritores de propriedades	Manipulação de objetos	Sintaxe mais eficiente	Quando não há necessidade de obter descritores de propriedades
Object.prototype.hasOwnProperty	Object.hasOwn	Para verificar a propriedade de um objeto de forma mais segura e moderna	Verificar propriedade de objeto	Verificação mais segura e moderna de propriedades de objetos	Quando não há necessidade de verificar propriedades de objetos
Polling para atualizações em tempo real	WebSockets, Server-Sent Events (SSE) ou WebRTC	Para comunicação em tempo real eficiente entre cliente e servidor	Aplicativos web que requerem atualizações em tempo real	Melhor desempenho e eficiência na comunicação em tempo real	Quando atualizações em tempo real não são necessárias
Promise.all	Promise.allSettled	Para lidar com múltiplas promessas, mesmo que algumas sejam rejeitadas	Operações assíncronas em paralelo	Obter resultados de todas as promessas, independentemente de sua resolução ou rejeição	Quando é necessário que todas as promessas sejam resolvidas para prosseguir
Promise.resolve().then(() => { ... });	async () => { ... }	Para operações assíncronas	Funções assíncronas	Simplificação de operações assíncronas	Quando não há necessidade de

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
					operações assíncronas
promise.then(() => { /* success / }).catch(() => { / error / }).finally(() => { / always */ });	promise.finally(() => { /* always */ });	Para garantir que ações de limpeza sejam executadas independentemente do resultado da promessa	Qualquer lugar onde promessas são usadas e necessitam de um bloco final de limpeza	Reduz a duplicação de código	Quando não há ações de limpeza necessárias após a promessa
Renderização no cliente (Client-Side Rendering) para todos os casos	Renderização híbrida (SSR + CSR) com base em rotas ou componentes	Para equilibrar desempenho, SEO e interatividade	Aplicativos web modernos com requisitos variados	Aproveitar os benefícios de SSR e CSR onde mais apropriado	Quando o aplicativo é completamente estático ou completamente interativo
Renderização no servidor (SSR) com templates	Renderização no servidor (SSR) com frameworks JavaScript (Next.js, Nuxt.js, etc.)	Para melhorar o desempenho, SEO e experiência do usuário em aplicativos web	Aplicativos web que requerem renderização no servidor	Melhor desempenho, SEO e experiência do usuário com ferramentas modernas de SSR	Quando a renderização no cliente é suficiente
require()	import/export	Para modularização em JavaScript moderno (ES6+)	Modularização de código JavaScript	Melhor reutilização e manutenção de código em JavaScript moderno (ES6+)	Quando não há necessidade de modularização de código
setInterval	requestAnimationFrame	Para animações mais suaves e eficientes, sincronizadas com a taxa de atualização da tela	Animações	Animações mais suaves e eficientes, sincronizadas com a taxa de atualização da tela	Quando não há necessidade de animações
SQL Queries sem preparação	Prepared Statements	Para prevenir injeções SQL	Consultas SQL dinâmicas	Prevenir injeções SQL e melhorar a segurança	Quando não há necessidade de consultas SQL dinâmicas
String concatenation	Template literals	Para construir strings de forma mais eficiente e legível	Construção de strings complexas	Sintaxe mais legível e eficiente para interpolação de strings	Quando não há necessidade de construção de strings complexas
String.prototype.match	RegExp.prototype.test	Para testar uma string contra uma expressão regular de	Testar strings contra expressões regulares	Forma mais eficiente e legível de testar strings	Quando não há necessidade de

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		forma mais eficiente			testar strings contra expressões regulares
switch (response.status) { case 200: handleSuccess(); break; case 404: handleNotFound(); break; default: handleError(); }	match (response.status) { when 200 -> handleSuccess(), when 404 -> handleNotFound(), when _ -> handleError() }	Para lógica condicional complexa	Controle de fluxo	Sintaxe mais clara e menos propensa a erros	Quando não há necessidade de lógica condicional complexa
Testes manuais	Testes automatizados (unitários, integração, e2e)	Para garantir a qualidade e confiabilidade do código	Em todo o ciclo de desenvolvimento	Detectar regressões, melhorar a confiabilidade e facilitar a manutenção	Quando o projeto é muito pequeno ou os testes manuais são suficientes
try { ... } catch (e) { ... }	try { ... } catch { ... }	Para simplificar blocos try-catch	Tratamento de exceções	Sintaxe mais clara e eficiente	Quando não há necessidade de tratamento de exceções
try { ... } catch (error) { console.error(error); }	try { ... } catch (error) { console.error('An error occurred:', error); }	Para adicionar contexto a erros	Tratamento de exceções	Melhorar a depuração e rastreamento de erros	Quando não há necessidade de tratamento de exceções
try...catch	Optional chaining (?.)	Para acessar propriedades de objetos sem necessidade de try...catch	Acesso a propriedades de objetos	Evitar erros ao acessar propriedades inexistentes	Quando não há necessidade de acessar propriedades de objetos
try...catch sem finally	try...catch...finally	Para garantir a execução de código de limpeza ou finalização, independentemente do sucesso ou falha da operação	Gerenciamento de exceções	Garantir a execução de código de limpeza ou finalização	Quando não há necessidade de tratamento de exceções
Uso de eval para executar código dinamicamente	Function constructor ou safer parsing methods	Para evitar vulnerabilidades de segurança associadas ao eval	Avaliação de código dinamicamente	Evitar vulnerabilidades de segurança associadas ao uso de eval	Quando não há necessidade de avaliação de código dinamicamente
var	let ou const	Para declarar variáveis em	Declaração de	Escopo de bloco para let e	Quando não há

Antigo	Atual	Quando	Local de Uso	Motivo Específico	Quando Não Usar Nenhuma
		ES6+; const para constantes, let para variáveis mutáveis	variáveis dentro de blocos ou funções	constantes imutáveis com const	necessidade de declarar novas variáveis
var x = 10;	let x = 10; const y = 20;	Para declarar variáveis em ES6+	Declaração de variáveis dentro de blocos ou funções	Escopo de bloco para let e constantes imutáveis com const	Quando não há necessidade de declarar novas variáveis
window.location.href para redirecionamento	window.location.assign ou window.location.replace	Para redirecionamentos, com assign adicionando uma entrada no histórico e replace substituindo a atual	Redirecionamentos	assign para adicionar ao histórico, replace para substituir a entrada atual	Quando não há necessidade de redirecionar
window.onload	DOMContentLoaded	Para garantir que o DOM esteja totalmente carregado antes da execução	Scripts que manipulam o DOM	Garantir que o DOM esteja totalmente carregado antes da execução	Quando não há scripts que manipulam o DOM
XMLHttpRequest	fetch	Para fazer requisições assíncronas de forma mais simples e moderna	Requisições HTTP assíncronas	Simplicidade e promessa embutida para melhor gerenciamento de requisições	Quando não há necessidade de fazer requisições HTTP
XMLHttpRequest para requisições assíncronas	async/await com fetch	Para simplificar a escrita de código assíncrono e melhorar a legibilidade	Requisições HTTP assíncronas	Simplificar escrita de código assíncrono e melhorar legibilidade	Quando não há necessidade de fazer requisições HTTP