FMX Academy

INFORMAÇÕES		
MÓDULO	AULA	INSTRUTOR
04 JavaScript Essencial e Consumo de APIs	Aula 02 – Operadores, Condicionais e Loops	Pedro Henrique Amadio

CONTEÚDO

1. Operadores Aritméticos

Operadores aritméticos realizam cálculos matemáticos básicos.

```
let a = 10;
let b = 3;

console.log(a + b); // 13
console.log(a - b); // 7
console.log(a * b); // 30
console.log(a / b); // 3.3333
console.log(a % b); // 1
console.log(a ** b); // 1000
```

O que cada um faz?

- `+` → Soma;
- `-` → Subtração;
- `*` → Multiplicação;
- `/` → Divisão;
- `%` → Seria o 'Mod' de outras linguagens de programação, o resto da divisão;
- `**` → Representa exponenciação;

2. Operadores Relacionais

Operadores relacionais comparam dois valores e retornam um booleano (true/false). Operadores lógicos combinam ou negam expressões booleanas para formar condições mais complexas.

- `>` → Maior que
- `>=` → Maior ou igual a
- `<` → Menor que
- `<=` → Menor ou igual a
- `==` → Comparação (somente do valor, reprimindo o tipo (evitar))
- `!=` → Diferença (somente do valor, reprimindo o tipo (evitar))
- `===` → Igualdade estrita (tanto valor, quanto tipo (preferido))
- `!==` → Diferença estrita (tanto valor, quanto tipo (preferido))

Exemplos

```
console.log(10 > 5);           // true
console.log(3 <= 2);           // false
console.log("5" == 5);         // true  (coerção de tipo)
console.log("5" === 5);        // false (tipos diferentes)
console.log("a" < "b");        // true  (ordem lexicográfica de strings)
```

Mas por que evitar "==" ?

```
console.log(0 == false);   // true  (coerção!)
console.log(0 === false);  // false

console.log("" == 0);      // true  (coerção!)
console.log("" === 0);     // false

console.log(null == undefined);   // true  (exceção histórica)
console.log(null === undefined);  // false
```

Tratativas com NaN

```
console.log(NaN === NaN);  // false
console.log(Number.isNaN(NaN)); // true (forma correta de checar NaN)
```

Comparações com tipos mistos:

```
console.log(NaN === NaN);  // false
console.log(Number.isNaN(NaN)); // true (forma correta de checar NaN)
```

Atalhos para negação e dupla negação

- ! → Negação
- !! → Dupla negação

Exemplo

```
console.log(!true);  // false
console.log(!!true); // true
```

Truthy & Falsy (coerção para Boolean)

- **Falsy**
  - False
  - 0
  - -0
  - “”
  - Null
  - Undefined
  - NaN

- **Truthy: Todos os demais, incluindo**

- []
- {}
- “0”
- “false”

```
console.log(Boolean(""));      // false
console.log(Boolean("teste")); // true
console.log(Boolean(0));       // false
console.log(Boolean(1));       // true
```

3. Operadores Lógicos

Operadores lógicos permitem combinar ou inverter valores booleanos “true” ou “false” para criar condições compostas. Eles são usados para tomar decisões em instruções condicionais e loops, retornando um resultado verdadeiro ou falso com base nas condições avaliadas.

- && → (E/AND), verdadeiro se todas as condições forem verdadeiras;
- || → (OU/OR), verdadeiro se pelo menos uma condição for verdadeira;
- ! → (NÃO/NOT), inverte o valor boolean;

- **Short-Circuit (Curto-Circuito)**
  Significa que o **JavaScript interrompe a avaliação de uma expressão lógica assim que o resultado já é conhecido.**
  Em outras palavras, **ele não precisa avaliar o resto da expressão se já souber o** resultado final.

  - **Operador &&:**
    O && só retorna true se **todos** os valores forem verdadeiros.
    Logo, se o primeiro valor for falso, o JavaScript **nem olha o segundo** – pois já sabe que o resultado será false.

    ```
    false && console.log("Isso nunca será executado");
    ```

    - Aqui o console.log **não roda**, porque o primeiro valor (false) já torna a expressão falsa.

    ```
    let logado = false;
    logado && console.log("Bem-vindo!"); // nada acontece
    ```

  - Operador || (OU lógico):
    O || retorna true se **pelo menos um** valor for verdadeiro.
    Então, se o primeiro já for verdadeiro, o segundo **nem é avaliado**.

    ```
    true || console.log("Isso também nunca será executado");
    ```

    - O resultado de (true || ...) é true imediatamente, e o resto é ignorado.

    ```
    let nome = "" || "Visitante";
    console.log(nome); // "Visitante"
    ```

    - Se o primeiro valor ("") for *falsy*, o JavaScript “pula para o próximo”.

- **Cuidado comum**
  Por causa do short-circuit, valores *falsy* como 0, "", ou null podem causar comportamentos inesperados:

```
let porta = 0 || 3000;
console.log(porta); // 3000 (mas 0 era válido!)
```

- Forma correta nesses casos (com operador *nullish* ??):

```
let porta = 0 ?? 3000;
console.log(porta); // 0 (mantém o valor válido)
```

4. Estruturas Condicionais

O programa avalia condições de cima para baixo. Assim que uma for verdadeira, as demais são ignoradas. (if → else if → else)

Exemplo base:

```
const nota = Number(prompt("Nota:"));
if (nota >= 7) {
  console.log("Aprovado");
} else if (nota >= 5) {
  console.log("Recuperação");
} else {
  console.log("Reprovado");
}
```

- **Ordem das Condições e Tratamento de Intervalos**
  Ao trabalhar com faixas/intervalos (idade, notas, preços), ordene da condição mais restritiva/alta prioridade para a mais abrangente. Isso evita sobreposição e resultados incorretos.

```
const salario = Number(prompt("Salário:"));
if (salario >= 20000) {
  console.log("Faixa A");
} else if (salario >= 10000) {
  console.log("Faixa B");
} else if (salario >= 3000) {
  console.log("Faixa C");
} else {
  console.log("Faixa D");
}
```

- **Evitando Aninhamento Profundo (if dentro de if)**
  Aninhamentos profundos dificultam a leitura. Prefira combinar condições com operadores lógicos, extrair subcondições para variáveis, ou reordenar a lógica.
    - Exemplo “aninhamento profundo”:

```
const idade = Number(prompt("Idade:"));
const temCNH = prompt("CNH? (s/n)") === "s";
if (idade >= 18) {
  if (temCNH) {
    console.log("Pode dirigir");
  } else {
    console.log("Maior de idade, sem CNH");
  }
} else {
  console.log("Menor de idade");
}
```

- Versão mais clara (menos níveis):

```
const idade = Number(prompt("Idade:"));
const temCNH = prompt("CNH? (s/n)") === "s";
if (idade < 18) {
  console.log("Menor de idade");
} else if (!temCNH) {
  console.log("Maior de idade, sem CNH");
} else {
  console.log("Pode dirigir");
}
```

- **Operador Ternário (?:)**
  Sintaxe: condição ? valorSeVerdadeiro : valorSeFalso. Use-o para expressões curtas de atribuição e evite ternários encadeados.

  Exemplos:

```
const status = (nota >= 7) ? "Aprovado" : "Reprovado";
const nome = prompt("Nome:") || "Visitante"; // OU lógico (valor padrão)
const maior = (a > b) ? a : b;
```

- **Early Exit (Saída Antecipada)**
  Encerra a execução de um bloco o mais cedo possível quando uma condição crítica não é atendida. Isso reduz aninhamento e melhora a legibilidade, colocando validações/erros no início.

  - Prévia com função (guard clause) – veremos na próxima aula:

```
function processarCompra(estoque, quantidade) {
  if (quantidade <= 0) {
    console.log("Quantidade inválida.");
    return; // EARLY EXIT
  }
  if (estoque < quantidade) {
    console.log("Estoque insuficiente.");
    return; // EARLY EXIT
  }
  console.log("Compra realizada!");
}
```

5. Estruturas de Repetição

Loops (ou laços de repetição) permitem executar um bloco de código várias vezes, enquanto uma condição for verdadeira ou até percorrer todos os elementos de uma lista (array).

- **for (tradicional)**
  Usado quando você **sabe quantas vezes** quer repetir o código.

```
for (let i = 1; i <= 5; i++) {
  console.log("Contagem: " + i);
}
```

  - `i` é o contador.
  - Atualiza a cada iteração (`i++`)

- **while**
  Executa **enquanto** a condição for verdadeira.
  Use while quando a repetição depender de uma condição lógica.

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

  Cuidado: se a condição nunca for falsa, cria um **loop infinito.**

- **do...while**
  Executa **ao menos uma vez**, mesmo que a condição seja falsa.

```
let numero = 0;
do {
  console.log("Executou ao menos uma vez!");
} while (numero > 0);
```

- **Loops em Arrays**
  Vamos usar o array abaixo como exemplo

```
const frutas = ["maçã", "banana", "laranja"];
```

- **for (tradicional e com índice)**

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

  - Permite **acessar o índice** facilmente (i).
  - É o **mais performático** em grandes listas (porque evita funções internas).

- **for...of**
  Itera diretamente sobre os valores.

```
for (const fruta of frutas) {
  console.log(fruta);
}
```

  - Simples e legível.
  - Não fornece índice (somente o valor).

- **for...in**
  Itera sobre as **chaves ou índices**.

```
for (const i in frutas) {
  console.log(i, frutas[i]);
}
```

  - Mais útil em **objetos**, não em arrays.
  - Em arrays, pode iterar sobre propriedades extras, se o array tiver alguma.
  - Evite para arrays grandes (pode ser mais lento e confuso).

- **.forEach()**
  Executa uma função para **cada elemento do array**.

```
frutas.forEach((item, indice) => {
  console.log(`Fruta ${indice}: ${item}`);
});
```

  - Ideal para percorrer arrays de forma **declarativa e legível**.
  - Não permite break ou continue (use return dentro da função apenas para pular internamente).
  - Levemente mais lento que o for clássico, pois cria uma função callback a cada iteração.

- **.map()**
  Cria **um novo array transformado**.

```
const maiusculas = frutas.map(item => item.toUpperCase());
console.log(maiusculas);
```

  - Retorna um **novo array** com base no retorno da função.
  - Não altera o array original.

- Use apenas quando quiser **gerar um novo array**, não para simples iteração.

- **.filter()**
  Filtra elementos com base em uma condição.

```
const maiusculas = frutas.map(item => item.toUpperCase());
console.log(maiusculas);
```

  - Cria um novo array apenas com os elementos que **passam no teste.**

- **Comparando Performance**

Método	Velocidade (≈)	Pode quebrar (usar "break")?	Uso Ideal
for	Mais rápido	Sim	Processamento pesado ou arrays grandes
for...of	Rápido	Sim	Iteração simples sobre valores
forEach()	Intermediária	Não	Loops declarativos curtos
map()	Intermediária	Não	Transformar arrays
while	Rápido	Sim	Condição de parada dinâmica
do...while	Razoável	Sim	Executar ao menos uma vez
for...in	Mais lento	Sim	Iterar propriedades de objetos, não arrays

6. Atividade Prática

Construa uma lista de perguntas e use um laço de repetição para solicitar as respostas do usuário.

```
const perguntas = [
  { title: "Qual seu nome?", default: "" },
  { title: "Qual sua idade?", default: "" },
  { title: "Qual cidade você mora?", default: "" },
  { title: "Qual sua linguagem de programação favorita?", default:
"" }
];

const respostas = [];

for (let i = 0; i < perguntas.length; i++) {
  const resposta = prompt(perguntas[i].title, perguntas[i].default);
  respostas.push({
    title: perguntas[i].title,
    data: resposta
  });
}

console.log("Respostas finais:");
console.log(respostas);
```

7. Desafio para Casa

Monte um script que peça nome e idade de 3 pessoas, armazenando em um array, e exiba apenas quem for maior de idade.

Dica (Para deixar mais fácil vai):

- const pessoas = []; // um array vazio para começar
- Dentro do loop, use o prompt() para perguntar **nome** e **idade.**
  - const nome = prompt("Nome da pessoa:");
  - const idade = Number(prompt("Idade:"));
- Depois, salve essas informações como **um objeto dentro do array**:
  - pessoas.push({ nome: nome, idade: idade });
- Para filtrar os maiores de idade use o .filter()
  - pessoas.filter(pessoa => pessoa.idade >= 18);
- Use o console.log para exibir o resultado

8. Exercícios de Fixação

1. Qual dos operadores abaixo é usado para verificar igualdade estrita (valor e tipo)?
   a) =
   b) ==
   c) ===
   d) !=

2. O que o operador lógico '&&' representa em JavaScript?
   a) OU lógico (retorna true se pelo menos uma condição for verdadeira)
   b) E lógico (retorna true apenas se todas as condições forem verdadeiras)
   c) Negação lógica (inverte o valor booleano)
   d) Soma lógica de valores booleanos

3. Qual o resultado da expressão: console.log(5 + '5') ?
   a) 10
   b) '55'
   c) 55 (number)
   d) undefined

4. Em um loop 'for', o que acontece se esquecermos de atualizar o contador (ex: i++)?
   a) O loop executa apenas uma vez
   b) O programa ignora o loop
   c) O loop entra em repetição infinita
   d) O programa gera erro de sintaxe

5. Sobre o método '.filter()', assinale a alternativa correta:
   a) Modifica o array original
   b) Retorna um único valor booleano
   c) Cria um novo array com os elementos que passam no teste
   d) Interrompe o loop ao encontrar o primeiro resultado