Estruturas de Dados em JavaScript

Plano de 4 Aulas (45 min cada)

Aula 1 – Tipos e Fundamentos

- Os tipos representam os **valores que podem ser manipulados e representados em uma linguagem de programação**.
- Uma das características mais fundamentais de qualquer linguagem é o conjunto de tipos que ela suporta.
- O funcionamento das variáveis também é uma característica essencial.

Visão Geral e Definições

Tipos Primitivos e Objetos

- **Tipos primitivos**: números, strings (texto) e valores booleanos.
- **Valores especiais**: `null` e `undefined` também são primitivos.
- **Objetos**:
  - qualquer valor que não seja número, string, boolean, símbolo, null ou undefined.
  - Um objeto é uma coleção de propriedades, cada uma com um nome e um valor (primitivo ou outro objeto).

Tipos Primitivos e Objetos

Exemplos de objetos:

- **Objeto ordinário**: coleção não ordenada de valores nomeados.
- **Array**: coleção ordenada de valores numerados.
- **Set**: representa um conjunto de valores únicos.
- **Map**: mapeamento de chaves para valores.
- **Typed Arrays**: arrays especializados para operações com bytes e outros dados binários.

Funções e Classes

- JavaScript difere de linguagens mais estáticas porque **funções e classes são valores manipuláveis**, não apenas parte da sintaxe.
- Como qualquer objeto, funções e classes podem ser manipuladas diretamente pelos programas.

Objetos, Métodos e Mutabilidade

- Técnicamente, **apenas objetos têm métodos**, mas números, strings, booleanos e símbolos se comportam como se os tivessem.
- Tipos primitivos são **imutáveis**, enquanto objetos são **mutáveis**.
- Strings podem ser vistas como arrays de caracteres, mas **não são mutáveis**.

Conversão de Tipos

- JavaScript converte valores automaticamente entre tipos:
  - Ex.: se uma string é esperada e você fornece um número, ele será convertido para string.
  - Valores não booleanos podem ser convertidos para booleanos quando necessário.
- O operador `==` é **desencorajado**; prefira `===`, que não realiza conversões de tipo.

Variáveis e Constantes

- Permitem usar nomes para se referir a valores no programa.
- No JavaScript, **constantes e variáveis são não tipadas**.

Estilo de Programação e Memória

- JavaScript suporta **programação orientada a objetos**.
- O interpretador realiza **coleta de lixo automática** para gerenciamento de memória.

Numbers em JavaScript

Numbers em JavaScript

- Tipo numérico principal: **Number**
- Representa inteiros e aproxima números reais
- Quando aparece diretamente no programa → **numeric literal**

```
let x = 42;          // inteiro
let y = 3.14;        // ponto flutuante
```

Integer Literals

- Inteiros em **base 10** → sequência de dígitos
- Também podem ser expressos em:
  - **Binário (base 2)** → prefixo `0b`
  - **Octal (base 8)** → prefixo `0o`

```
let decimal = 123;
let bin = 0b1111;     // 15
let oct = 0o17;       // 15
```

Floating-Point Literals

- Podem ter **ponto decimal**
- Sintaxe tradicional de números reais
- Podem usar **notação exponencial**

```
let pi = 3.14159;
let exp = 1.23e4;   // 12300
```

Arithmetic em JavaScript

- Operações matemáticas via **objeto Math**
- Não gera erro em:
  - **Overflow**
  - **Underflow**
  - **Divisão por zero**

```
console.log(1 / 0);       // Infinity
console.log(-1 / 0);      // -Infinity
console.log("abc" / 2);   // NaN
console.log(Math.sqrt(-1)); // NaN
```

NaN e Funções Úteis

- `NaN` não é igual a nenhum valor, nem a si mesmo
- Funções globais:
  - `isNaN(x)`
  - `Number.isNaN(x)`
  - `Number.isFinite(x)`

```
console.log(NaN === NaN);           // false
console.log(isNaN("abc"));          // true
console.log(Number.isNaN("abc"));   // false
console.log(Number.isFinite(42));   // true
```

Binary Floating-Point & Rounding Errors

- Números reais infinitos ≠ representações finitas
- Exemplo: **0.1 não é representado exatamente**
- Problema em comparações

```
console.log(0.1 + 0.2);       // 0.30000000000000004
console.log((0.3 - 0.2) === (0.2 - 0.1)); // false
```

BigInt

- Criado para representar **inteiros muito grandes**
- Literal: número seguido de `n`
- Suporte a `0b`, `0o`, `0x`

```
let big = 12345678901234567890123456789n;
let hexBig = 0x1fffffffffffffn;
console.log(big + 10n); // funciona
```

BigInt x Number

- Não pode misturar BigInt com Number em operações
- Comparações funcionam

```
console.log(10n + 20n);   // 30n
console.log(10n > 5);     // true
// console.log(10n + 5);  // Erro: não pode misturar BigInt e Number
```

JavaScript Text (Strings)

JavaScript Text (Strings)

- O tipo usado para representar **texto** em JavaScript é o **string**.
- Uma string é uma **sequência ordenada imutável** de valores de 16 bits (Unicode).
- Índices em strings (e arrays) começam do **zero** (zero-based indexing).

```
let texto = "Olá!";
console.log(texto[0]); // "O"
console.log(texto[1]); // "l"
```

Escape Sequences em Literais de String

- O caractere **barra invertida (\)** tem uso especial.
- Ele permite **escapar** caracteres que, de outra forma, teriam interpretação especial.

Exemplos:

```
let aspasSimples = 'I\'m learning JS';
let novaLinha = "Linha 1\nLinha 2";
let unicode = "\u2764"; // ❤
```

Trabalhando com Strings

Concatenação

- O operador `+` concatena strings.

```
let nome = "Ana";
let saudacao = "Olá, " + nome + "!";
console.log(saudacao); // "Olá, Ana!"
```

Comparação

- Strings podem ser comparadas com `===` e `!==`.

```
console.log("abc" === "abc"); // true
console.log("abc" !== "def"); // true
```

Propriedades e Métodos de Strings

- `length` : retorna o número de caracteres (valores de 16 bits).

```
let frase = "JavaScript";
console.log(frase.length); // 10
```

Imutabilidade

- Strings não podem ser alteradas.
- Métodos retornam **novas strings**.

```
let palavra = "casa";
let nova = palavra.replace("c", "m");
console.log(palavra); // "casa"
console.log(nova);    // "masa"
```

API de Strings

Alguns métodos úteis:

- `toUpperCase()` → converte para maiúsculas
- `toLowerCase()` → converte para minúsculas
- `slice()` → retorna parte da string
- `includes()` → verifica se contém um trecho
- `split()` → divide em array

```
let exemplo = "Aprendendo JS";
console.log(exemplo.toUpperCase()); // "APRENDENDO JS"
console.log(exemplo.includes("JS")); // true
```

Template Literals (ES6+)

- Usam **crase** (`` ` ``) em vez de aspas.
- Permitem interpolar expressões dentro de `${...}`.
- Podem conter várias linhas sem escapes.

```
let nome = "Carlos";
let idade = 25;

let msg = `Meu nome é ${nome} e tenho ${idade} anos.`;
console.log(msg);
// "Meu nome é Carlos e tenho 25 anos."
```

Template Literals

- Podem conter várias expressões.
- Suportam caracteres especiais como strings normais.
- Podem se estender em múltiplas linhas.

```
let item = "livro";
let preco = 49.90;

let nota = `
Produto: ${item}
Preço: R$${preco}
Obrigado pela compra!
`;

console.log(nota);
```

Boolean, null e undefined

Boolean

- Um **valor booleano** representa verdadeiro ou falso, ligado ou desligado, sim ou não.
- Há apenas **dois valores possíveis**:
  - `true`
  - `false`

Exemplo:

```
let ativo = true;
let inativo = false;
```

Boolean a partir de Comparações

Valores booleanos geralmente resultam de **comparações** em programas:

```
console.log(5 > 3);   // true
console.log(10 === 5); // false
```

Conversão para Boolean

Qualquer valor em JavaScript pode ser convertido para booleano.

- Falsy (avaliam como falso):
  - `false`, `0`, `-0`, `""`, `null`, `undefined`, `NaN`
- Truthy (avaliam como verdadeiro):
  - Qualquer outro valor (incluindo objetos e arrays)

```
console.log(Boolean(""));   // false
console.log(Boolean("JS")); // true
console.log(Boolean([]));   // true
```

Operadores Booleanos

- `&&` → AND
- `||` → OR
- `!` → NOT

Exemplos:

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

Método toString()

Valores booleanos podem ser convertidos para string com `toString()`:

```
let valor = true;
console.log(valor.toString()); // "true"
```

`null` e `undefined`

`null`

- `null` é uma **palavra-chave** que indica **ausência de valor**.
- Geralmente usado quando queremos dizer que algo está **vazio** ou **não tem valor**.

```
let usuario = null;
console.log(usuario); // null
```

`undefined`

- `undefined` representa uma ausência **mais profunda** de valor.
- Situações comuns:
  - Variáveis declaradas, mas **não inicializadas**
  - Funções que não retornam valor
  - Parâmetros não passados em funções

```
let x;
console.log(x); // undefined

function teste() {}
console.log(teste()); // undefined
```

`null` vs `undefined`

- Ambos indicam **ausência de valor**.
- `==` → considera `null` e `undefined` iguais
- `===` → distingue os dois

Exemplo:

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

Valores Primitivos Imutáveis e Objetos Mutáveis

Diferença Fundamental

- Em JavaScript há uma distinção clara entre:
  - **Primitivos**: `undefined`, `null`, `boolean`, `number`, `string`, `symbol`, `bigint`
  - **Objetos**: arrays, funções, objetos literais etc.
- **Primitivos são imutáveis**
- **Objetos são mutáveis**

Strings e Imutabilidade

- Strings são primitivas e **imutáveis**.
- Métodos de string retornam **novas strings**, não alteram a original.

```
let palavra = "Java";
let nova = palavra.toUpperCase();

console.log(palavra); // "Java"
console.log(nova);    // "JAVA"
```

Comparação de Primitivos

📈 **Primitivos** são comparados **por valor**.

```
console.log(5 === 5);       // true
console.log("abc" === "abc"); // true
console.log("abc" === "def"); // false
```

Comparação de Strings

- Strings só são iguais se:
  - Tiverem o mesmo comprimento
  - Cada caractere for igual em cada posição

```
console.log("ola" === "ola"); // true
console.log("ola" === "olá"); // false
```

Objetos são Mutáveis

📈 Objetos podem ter seus valores **alterados após a criação**.

```
let pessoa = { nome: "Ana" };
pessoa.nome = "Maria";

console.log(pessoa); // { nome: "Maria" }
```

Comparação de Objetos

- Objetos são comparados **por referência** e **não por valor**.
- Mesmo que dois objetos tenham propriedades iguais, eles não são iguais.

```
let obj1 = { a: 1 };
let obj2 = { a: 1 };

console.log(obj1 === obj2); // false
```

- Objetos iguais

```
let obj1 = { a: 1 };
obj2 = obj1;
console.log(obj1 === obj2); // true
```

Arrays

📈 Comparação de arrays diferentes

```
let arr1 = [1, 2, 3];
let arr2 = [1, 2, 3];

console.log(arr1 === arr2); // false
```

📈 Comparação de arrays iguais

```
let arr1 = [1, 2, 3];
let arr2 = arr1;

console.log(arr1 === arr2); // true
```

Referências de Objetos

- Variáveis armazenam uma **referência** ao objeto, e não o objeto em si.
- Atribuir um objeto a outra variável copia apenas a **referência**.

```
let a = { valor: 10 };
let b = a;
b.valor = 20;
console.log(a.valor); // 20
```

- O resultado abaixo é `true`

```
let a = { valor: 10 };
let b = a;
console.log(a === b) //true
```

Cópia de Objetos e Arrays

↗ Para criar **cópias reais**, é necessário copiar manualmente ou usar métodos como:

```
// Objetos
let original = { x: 1, y: 2 };
let copia = { ...original };

console.log(original === copia); // false

// Arrays
let nums = [1, 2, 3];
let clone = [...nums];

console.log(nums === clone); // false
```