Capítulo – Tipos e Números em JavaScript

1. Introdução aos Tipos

Em qualquer linguagem de programação, os **tipos de dados** são fundamentais: eles definem quais valores podem ser representados e como podem ser manipulados. Em JavaScript, a forma como os **valores** e **variáveis** funcionam é uma das suas características essenciais.

A linguagem distingue dois grandes grupos:

- **Tipos primitivos**: valores básicos, imutáveis, representando números, textos (strings), valores booleanos, além dos especiais `null`, `undefined` e `symbol`.
- **Tipos de objeto**: estruturas mais complexas que armazenam coleções de valores ou comportamentos.

Essa divisão é importante porque os primitivos são simples e imutáveis, enquanto os objetos podem crescer e ser alterados dinamicamente.

---

2. Tipos Primitivos

JavaScript define como primitivos:

- **Number**: para inteiros e números reais aproximados;
- **String**: sequências de caracteres;
- **Boolean**: valores lógicos `true` e `false`;
- **Symbol**: identificadores únicos;
- **null**: ausência de valor;
- **undefined**: valor não inicializado.

Exemplo:

```
let idade = 25;          // Number
let nome = "Maria";      // String
let ativo = true;        // Boolean
let vazio = null;        // Null
let indefinido;          // Undefined
```

Vale notar que os primitivos não possuem métodos, mas em JavaScript eles **se comportam como se tivessem**. Isso acontece porque, quando chamamos algo como `"texto".toUpperCase()`, a linguagem cria temporariamente um objeto para representar a string e disponibilizar o método.

---

3. Objetos e Estruturas de Dados

Qualquer valor que não seja primitivo é um **objeto**. Um objeto é uma coleção de propriedades, onde cada propriedade possui um nome e um valor.

Exemplo de objeto simples:

```
let pessoa = {
  nome: "Ana",
  idade: 30,
  ativo: true
};
```

Além dos objetos comuns, a linguagem disponibiliza:

- **Array**: coleção ordenada de valores.
- **Set**: conjunto de valores únicos.
- **Map**: mapeamento de chaves para valores.
- **Typed Arrays**: estruturas específicas para lidar com dados binários.

JavaScript também trata **funções e classes** como valores. Isso significa que podem ser atribuídas a variáveis, passadas como argumentos e retornadas de outras funções. Essa flexibilidade dá suporte a um estilo de programação **funcional e orientado a objetos**.

Outro aspecto importante é a **coleta de lixo automática**, que libera memória de valores que não estão mais sendo utilizados pelo programa.

---

4. Mutabilidade e Conversões de Tipo

Os **primitivos** são imutáveis: uma vez criado, um valor não pode ser alterado. Já os **objetos** são mutáveis: suas propriedades podem ser modificadas a qualquer momento.

Apesar disso, o JavaScript realiza **conversões automáticas de tipos** quando necessário. Por exemplo:

```
console.log("5" * 2);   // 10 (string foi convertida para número)
console.log(1 == "1");  // true (conversão implícita)
```

Esse comportamento pode gerar ambiguidades. Por isso, recomenda-se o uso do **operador de igualdade estrita** `===`, que não realiza conversão de tipos:

```
console.log(1 === "1"); // false
```

---

5. Números em JavaScript

5.1 O Tipo Number

O tipo **Number** representa tanto inteiros quanto números de ponto flutuante. Quando um número aparece diretamente no programa, é chamado de **numeric literal**.

Exemplos:

```
let inteiro = 42;
let real = 3.14;
```

Inteiros

- Em **decimal**: `let a = 123;`
- Em **binário**: `let b = 0b1111; // 15`
- Em **octal**: `let c = 0o17; // 15`
- Em **hexadecimal**: `let d = 0xFF; // 255`

Ponto flutuante

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

5.2 Operações Matemáticas

JavaScript suporta as operações básicas (+, -, *, /, %) e oferece funções avançadas por meio do objeto **Math**.

Exemplo:

```
console.log(Math.sqrt(16));   // 4
console.log(Math.pow(2, 3));  // 8
```

Diferente de muitas linguagens, o JavaScript não gera erros em casos como **divisão por zero** ou **overflow**. Em vez disso, retorna valores especiais:

```
console.log(1 / 0);   // Infinity
console.log(-1 / 0);  // -Infinity
console.log(0 / 0);   // NaN
```

5.3 Valores Especiais: NaN e Infinity

O valor **NaN** (Not-a-Number) é usado quando uma operação numérica não tem resultado definido. Uma particularidade é que `NaN` **não é igual a si mesmo**:

```
console.log(NaN === NaN); // false
```

Para lidar com isso:

- `isNaN(x)`: verifica se algo **pode ser convertido para NaN**;
- `Number.isNaN(x)`: verifica se o valor é **estritamente NaN**;
- `Number.isFinite(x)`: retorna `true` se o valor é finito (≠ Infinity, -Infinity, NaN).

5.4 Ponto Flutuante e Erros de Arredondamento

JavaScript segue o padrão de representação binária em ponto flutuante (IEEE 754). Isso significa que nem todos os números reais podem ser representados com exatidão.

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

Esses erros são pequenos, mas tornam comparações delicadas:

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

Esse problema não é exclusivo do JavaScript, mas de qualquer linguagem que use representação binária de ponto flutuante.

---

6. BigInt

O tipo **BigInt** foi introduzido para lidar com inteiros de tamanho arbitrário, superando as limitações de `Number`.

Exemplo de literal BigInt:

```
let big = 1234567890123456789012345678901234567890n;
let hexBig = 0x1fffffffffffffn;
```

Com BigInt, operações aritméticas funcionam normalmente, mas não é possível misturar com valores do tipo Number:

```
console.log(10n + 20n);   // 30n
console.log(10n > 5);     // true
// console.log(10n + 5);  // Erro
```

Um detalhe importante: funções do objeto **Math** não aceitam BigInt.

---

7. Variáveis e Constantes

Em JavaScript, variáveis e constantes são **não tipadas**, o que significa que o mesmo identificador pode armazenar valores de diferentes tipos em momentos distintos.

```
let x = 42;
x = "agora sou string";
```

O uso de `const` e `let` ajuda a tornar o código mais previsível:

- `const`: cria constantes cujo valor não pode ser reatribuído;
- `let`: cria variáveis de escopo de bloco.

---

8. Conclusão

O sistema de tipos em JavaScript é ao mesmo tempo **flexível e desafiador**. Os primitivos garantem simplicidade, enquanto os objetos oferecem estruturas complexas e reutilizáveis. A forma como números são tratados — incluindo `NaN`, `Infinity`, arredondamentos e BigInt — mostra como a linguagem privilegia a **praticidade sobre a rigidez**, o que a torna extremamente poderosa em aplicações web e sistemas dinâmicos.

Aqui está um **resumo em estilo de capítulo de apostila** sobre **Strings em JavaScript**, baseado no conteúdo que você passou:

Capítulo: Manipulação de Texto em JavaScript

O **tipo de dado para representar texto** em JavaScript é a **string**. Uma string é uma **sequência ordenada e imutável** de valores de 16 bits, onde cada valor geralmente corresponde a um caractere Unicode. Assim como os arrays, as strings usam **indexação baseada em zero**, ou seja, o primeiro caractere ocupa a posição `0`.

Exemplo:

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

---

1. Escape Sequences em Literais de String

Em strings, o caractere **barra invertida (\)** é usado para criar **sequências de escape**. Essas sequências permitem representar caracteres especiais dentro de uma string.

Alguns exemplos comuns:

- `\'` → aspas simples

- `\"` → aspas duplas
- `\n` → nova linha
- `\t` → tabulação
- `\uXXXX` → caractere Unicode

Exemplo:

```
let exemplo1 = 'I\'m learning JS';
let exemplo2 = "Linha 1\nLinha 2";
let exemplo3 = "\u2764"; // ♡
```

---

2. Operações Básicas com Strings

Concatenação

O operador `+` pode ser usado para **juntar strings**:

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

Comparação

Strings podem ser comparadas com os operadores de igualdade estrita (`===`) e diferença (`!==`):

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

---

3. Propriedades e Imutabilidade

Propriedade `.length`

Indica o número de caracteres da string.

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

Imutabilidade

Strings são **imutáveis** em JavaScript. Isso significa que, ao aplicar métodos como `replace()` ou `toUpperCase()`, uma **nova string** é criada, sem modificar a original.

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

---

4. Principais Métodos de Strings

JavaScript oferece uma **API rica** para trabalhar com strings. Alguns métodos importantes:

- `toUpperCase()` → converte para maiúsculas
- `toLowerCase()` → converte para minúsculas
- `slice(início, fim)` → extrai parte da string
- `includes(substring)` → verifica se contém determinado texto
- `split(delimitador)` → divide a string em um array

Exemplo:

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

---

5. Template Literals (ES6+)

Com a introdução do **ES6**, JavaScript passou a suportar **template literals**, que utilizam **crases (`)** em vez de aspas. Essa forma de declaração oferece **três grandes vantagens**:

1. **Interpolação de expressões** com `${...}`:

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

2. **Suporte a múltiplas linhas** sem necessidade de `\n`:

   ```
   let msg = `Linha 1
   Linha 2
   Linha 3`;
   ```

3. **Uso combinado com escape sequences**:

   ```
   ```

```
let preco = 49.90;
let nota = `
Produto: Livro
Preço: R$${preco}
Obrigado pela compra!
`;
console.log(nota);
```

Conclusão

Strings em JavaScript são fundamentais para a manipulação de texto em programas. Principais pontos a reter:

- São **imutáveis** e usam **indexação baseada em zero**.
- Aceitam **sequências de escape** para representar caracteres especiais.
- Possuem diversas operações nativas para **concatenação, comparação e transformação**.
- A partir do ES6, os **template literals** facilitaram a interpolação de variáveis e a criação de strings multilinha.

Capítulo: Boolean, null e undefined em JavaScript

1. O tipo Boolean

O tipo **boolean** é um dos mais simples e fundamentais de JavaScript. Ele representa apenas dois valores possíveis:

- `true` → verdadeiro, ligado, sim.
- `false` → falso, desligado, não.

Esses valores são amplamente utilizados para **controle de fluxo** e **avaliação de condições**.

Exemplo:

```
let ativo = true;
let inativo = false;
console.log(ativo);   // true
console.log(inativo); // false
```

1.1 Booleanos em comparações

Valores booleanos geralmente são o **resultado de expressões de comparação**:

```
console.log(5 > 3);     // true
console.log(10 === 5);  // false
console.log("JS" !== "Java"); // true
```

1.2 Conversão de valores para Boolean

Qualquer valor em JavaScript pode ser convertido implicitamente para um booleano. Chamamos esses valores de **truthy** (avaliados como verdadeiro) ou **falsy** (avaliados como falso).

- **Falsy**: `false`, `0`, `-0`, `""` (string vazia), `null`, `undefined`, `NaN`.
- **Truthy**: todos os outros valores (incluindo arrays e objetos).

Exemplo:

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

1.3 Operadores Booleanos

JavaScript possui operadores lógicos que retornam valores booleanos:

- `&&` (AND): verdadeiro apenas se **ambos** os operandos forem verdadeiros.
- `||` (OR): verdadeiro se **pelo menos um** dos operandos for verdadeiro.
- `!` (NOT): inverte o valor lógico.

Exemplo:

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

1.4 Conversão para string

O tipo boolean possui o método `toString()`, que converte `true` em `"true"` e `false` em `"false"`:

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

---

2. O valor null

O valor especial `null` é uma **palavra-chave da linguagem** que representa a ausência intencional de valor. Ele é frequentemente utilizado quando queremos indicar que uma variável foi **deliberadamente esvaziada**.

Exemplo:

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

Aqui, `usuario` não aponta para nenhum objeto ou valor válido.

---

3. O valor undefined

Diferente de `null`, o valor **`undefined`** representa uma ausência **mais profunda** de valor, geralmente associada a algo não inicializado.

Situações em que `undefined` aparece:

- Variáveis declaradas mas não inicializadas.
- Parâmetros de função não passados.
- Funções que não possuem retorno explícito.

Exemplo:

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

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

function soma(a, b) {
  return a + b;
}
console.log(soma(5)); // NaN (porque b é undefined)
```

---

4. null vs undefined

Apesar de diferentes, **ambos indicam ausência de valor**:

- `null`: ausência intencional (escolha do programador).
- `undefined`: ausência implícita (valor não definido pela linguagem).

Comparação:

```
console.log(null == undefined);  // true (considerados iguais pelo ==)
console.log(null === undefined); // false (estritamente diferentes)
```

Tanto `null` quanto `undefined` são considerados **falsy**:

```
if (!null) console.log("null é falsy");
if (!undefined) console.log("undefined é falsy");
```

---

5. Resumo do Capítulo

- **Boolean**: representa `true` ou `false`.
    - Usado em comparações, condições e operadores lógicos.
    - Truthy vs Falsy: todo valor em JS pode ser convertido implicitamente para boolean.
- **null**: ausência intencional de valor.
    - Usado quando queremos "zerar" uma variável ou indicar vazio.
- **undefined**: ausência implícita de valor.
    - Variáveis não inicializadas, funções sem retorno e parâmetros ausentes assumem `undefined`.
- **Comparação**:
    - `null == undefined` → `true`
    - `null === undefined` → `false`

Capítulo – Valores Primitivos Imutáveis e Objetos Mutáveis

No JavaScript, existe uma distinção essencial entre **valores primitivos** e **objetos**. Essa diferença impacta diretamente na forma como os dados são manipulados, armazenados na memória e comparados.

---

1. Valores Primitivos Imutáveis

Os valores primitivos em JavaScript são:

- **undefined**
- **null**
- **boolean**
- **number**
- **string**
- **symbol**
- **bigint**

Esses valores são **imutáveis**, ou seja, não podem ser alterados depois de criados. Se você aplicar um método em uma string, por exemplo, o valor original não será modificado; em vez disso, será retornado um **novo valor**.

📌 **Exemplo:**

```
let nome = "JavaScript";
let novoNome = nome.toUpperCase();

console.log(nome);      // "JavaScript" (valor original imutável)
console.log(novoNome);  // "JAVASCRIPT" (nova string criada)
```

Isso significa que **qualquer transformação gera um novo valor**, sem modificar o anterior.

Outra característica importante:

- Valores primitivos são **comparados por valor**. Isso quer dizer que dois primitivos são iguais se possuírem exatamente o mesmo valor.

📌 **Exemplo:**

```
let a = 10;
let b = 10;
let c = 20;

console.log(a === b); // true (valores iguais)
console.log(a === c); // false (valores diferentes)

let str1 = "abc";
let str2 = "abc";
console.log(str1 === str2); // true (mesmo conteúdo)
```

---

2. Objetos Mutáveis

Diferente dos primitivos, os **objetos** em JavaScript são **mutáveis**. Isso significa que, após a criação, seus valores internos (propriedades e elementos) podem ser alterados.

📌 **Exemplo:**

```
let pessoa = { nome: "Ana", idade: 25 };
pessoa.idade = 26;  // alteração permitida

console.log(pessoa); // { nome: "Ana", idade: 26 }
```

Os objetos não são comparados pelo valor de suas propriedades, mas sim pela **referência na memória**. Assim, dois objetos distintos com os mesmos valores **não são iguais**.

📌 **Exemplo:**

```
let obj1 = { x: 1, y: 2 };
let obj2 = { x: 1, y: 2 };
```

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

let obj3 = obj1;
console.log(obj1 === obj3); // true (mesma referência)
```

Isso ocorre porque, quando você atribui um objeto a uma variável, na verdade está atribuindo a **referência** para aquele objeto na memória, e não uma cópia independente.

---

3. Objetos como Tipos de Referência

Por serem mutáveis e manipulados por referência, os objetos são chamados também de **tipos de referência**. Quando uma variável recebe um objeto ou um array, ela não contém o valor em si, mas apenas um **ponteiro** que aponta para a localização do objeto na memória.

📌 **Exemplo:**

```
let lista1 = [1, 2, 3];
let lista2 = lista1; // lista2 aponta para o mesmo array

lista2.push(4);

console.log(lista1); // [1, 2, 3, 4] (alterado também)
console.log(lista2); // [1, 2, 3, 4]
```

Se desejamos criar uma **cópia independente**, precisamos explicitamente copiar os elementos ou propriedades, usando técnicas como:

- **Operador spread (`...`)**
- **Métodos específicos (`slice`, `Object.assign`, etc.)**

📌 **Exemplo:**

```
let original = [1, 2, 3];
let copia = [...original]; // cópia independente

copia.push(4);

console.log(original); // [1, 2, 3]
console.log(copia);    // [1, 2, 3, 4]
```

---

4. Comparação entre Primitivos e Objetos

Característica	Primitivos	Objetos

Característica	Primitivos	Objetos
Mutabilidade	Imutáveis	Mutáveis
Comparação	Por valor	Por referência
Armazenamento	Valor direto	Referência (ponteiro)
Exemplo de comportamento	`str.toUpperCase()` cria nova string	Alterar `obj.propriedade` muda o mesmo objeto

Conclusão

A distinção entre **valores primitivos imutáveis** e **objetos mutáveis** é fundamental para compreender o comportamento do JavaScript. Enquanto os primitivos são simples, imutáveis e comparados diretamente, os objetos são complexos, mutáveis e manipulados por referência. Essa diferença impacta operações de cópia, comparação e manipulação de dados, e deve ser sempre considerada no desenvolvimento de aplicações.