

JavaScript learning path

Alright 🚀 — let's move on to the **third primitive**: `String`.

This one looks simple at first glance, but under the hood it's **one of the richest and trickiest primitives in JS** because of encodings, immutability, and hidden optimizations.

◆ Primitive 3: String

✓ What is a String?

- A **sequence of characters** (textual data) inside:
 - **single quotes** `'hello'`
 - **double quotes** `"hello"`
 - **backticks** ``hello`` (template literals, ES6+).
- Technically, each character is a **UTF-16 code unit** (not always a single symbol — think emojis 🚀).

js

```
let str1 = "Hello";
let str2 = 'World';
let str3 = `Hello ${str2}`; // Template literal with interpolation
```

⚙ Internal Representation

- Stored as **UTF-16 encoded values**.
- Each element is a **16-bit unit**, but some characters (like emojis, non-Latin scripts) require **2 units** (called *surrogate pairs*).

js

```
console.log("😊".length); // 2, not 1
```

⚠ Interview trap: **String length** ≠ **number of visible characters**.

🌟 Key Properties

1. Immutability

- Strings cannot be changed in place.

```
js

let s = "hello";
s[0] = "H";
console.log(s); // "hello", unchanged
```

👉 Every operation (`slice` , `concat` , `replace`) creates a **new string**.

2. Indexed access

```
js

let word = "JS";
console.log(word[0]); // "J"
console.log(word.charAt(1)); // "S"
```

3. Length

```
js

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

🔧 Common Operations

```
js

let msg = "JavaScript";

// Substring
console.log(msg.slice(0, 4)); // "Java"

// Replace
console.log(msg.replace("Java", "Type")); // "TypeScript"

// Upper/lower case
console.log(msg.toUpperCase()); // "JAVASCRIPT"

// Concatenation
```

```
console.log("Hello" + " " + "World"); // "Hello World"
console.log(`Hello ${"World"}`); // "Hello World" (template literal)
```

⚡ Performance & Memory

- Since strings are **immutable**, repeated concatenation (`+=`) can be costly in large loops.
- Use `Array.join()` or **template literals** in performance-critical code.

```
js

// Slow for very large strings
let str = "";
for (let i = 0; i < 100000; i++) {
  str += i;
}

// Better
let arr = [];
for (let i = 0; i < 100000; i++) {
  arr.push(i);
}
let result = arr.join("");
```

⚠ Tricky Cases

1. Unicode quirks

```
js

console.log("❤️".length); // 2
console.log([... "❤️"].length); // 1 (spread handles Unicode properly)
```

2. Comparisons

- Strings are compared **lexicographically by UTF-16 code units**.

```
js

console.log("Z" > "a"); // false (90 vs 97 in UTF-16)
```

3. Falsy string

```
js
```

```
console.log(Boolean("")); // false
```

Mini-Exercises

1. What will this print?

```
js
```

```
console.log("hello".toUpperCase()[1]);
```

2. What is the output?

```
js
```

```
console.log("❤️".length);  
console.log([... "❤️"].length);
```

3. Predict:

```
js
```

```
console.log("2" + 2);  
console.log("2" - 2);
```

👉 Do you want me to solve these String exercises now with explanations + edge cases, or should I first dive deeper into String internals (Unicode, surrogate pairs, normalization) before solving?

Excellent 🙌 — let's zoom in on why

```
js
```

```
"❤️".length // → 2
```

instead of the expected `1`.

🔍 Why "💖".length === 2?

1. Strings in JavaScript are UTF-16 encoded

- Each character is stored as **16 bits (2 bytes)**.
- Most common characters (A–Z, a–z, numbers, many symbols) fit in **one 16-bit unit**.

✅ Example:

```
js
console.log("A".length); // 1
```

Because "A" = 0x0041 (1 code unit).

2. Unicode characters outside the Basic Multilingual Plane (BMP)

- Unicode has more than **1 million characters** (Chinese, emojis, ancient scripts, etc.).
- UTF-16 represents these *special* characters using a **pair of 16-bit units** → called a **surrogate pair**.

💖 = Unicode code point U+1F496

- In UTF-16, this is stored as:
 - **High surrogate:** 0xD83D
 - **Low surrogate:** 0xDC96
- So "💖" = 2 code units.

3. JavaScript's .length counts code units, not characters

```
js
console.log("💖".length); // 2
```

It sees *two 16-bit units*, so it returns 2.

4. Correct way to count actual characters

We want **Unicode code points** (what humans see as characters), not code units.

Solutions:

```
js
```

```
console.log([... "💖"].length); // 1 (spread into array of code points)
console.log(Array.from("💖").length); // 1
```

👉 These use **iterators** under the hood, which are Unicode-aware.

⚡ Interview-Level Brain-Bender

```
js
```

```
console.log("💖"[0]);
console.log("💖"[1]);
```

What happens?

- `"💖"[0]` → `"\uD83D"` (half of the surrogate pair, meaningless symbol)
- `"💖"[1]` → `"\uDC96"` (other half, also broken)

💡 That's why naive string manipulation can break emojis.

✅ Key Takeaways

- `.length` counts **UTF-16 code units**, not real characters.
 - Emojis & rare symbols often = 2 units.
 - Use `Array.from(str)` or spread (`[...str]`) to work with full Unicode safely.
-

👉 Do you want me to now **solve the pending String exercises step by step** (including this one), or first do a **deep dive into Unicode internals** (BMP, surrogate pairs, grapheme clusters) before moving on?

Great observation 👍 and this is exactly the contrast that makes `"💖".length === 2` confusing. Let's carefully break down why `"JavaScript".length` is `10`, but `"💖".length` is `2`.

🔍 Why `"JavaScript".length === 10`

1. ASCII vs Unicode

- `"JavaScript"` is made of plain ASCII characters (J, a, v, a, S, c, r, i, p, t).
- ASCII characters are part of the Basic Multilingual Plane (BMP).
- Each one fits neatly into one 16-bit UTF-16 code unit.

👉 `"JavaScript"` has 10 characters → 10 code units → length = 10

```
js
console.log("JavaScript".length); // 10
```

2. Emojis / Special Unicode (💖 example)

- Emojis like `"💖"` live outside the BMP.
- UTF-16 represents them using two code units (surrogate pair).
- So `"💖".length` counts 2 code units, not 1 visible symbol.

```
js
console.log("💖".length); // 2
```

⚡ Quick Comparison

String	Visible characters	Code units (UTF-16)	<code>.length</code>
<code>"JavaScript"</code>	10	10	10
<code>"💖"</code>	1	2 (surrogate pair)	2

String	Visible characters	Code units (UTF-16)	<code>.length</code>
"👍👍"	2	4 (2 each)	4

✓ How to Get Real Character Count (Unicode-Aware)

js

```
console.log([...]"JavaScript"].length); // 10
console.log([...]"❤️"].length); // 1
console.log([...]"👍👍"].length); // 2
```

👉 Spread (`...`) or `Array.from` iterates over **full code points**, not just 16-bit units.

🧠 Interview Trap

js

```
console.log("❤️".split(" ").length);
```

What do you think?

- Output: `2` (because `.split(" ")` splits by code units, not Unicode code points).
- To split properly:

js

```
console.log([...]"❤️"].length); // 1
```

✓ So:

- `"JavaScript"` uses only **BMP ASCII** letters → **1 unit per character** → **length = characters count**.
- `"❤️"` is outside BMP → stored as **2 units** → `.length = 2`.

👉 Do you want me to now **deep dive into Unicode internals** (BMP, surrogate pairs, grapheme clusters), or should I **go back and solve all the String exercises step by step** (with tricky cases)?