Assume response must be straight forward and brief then assume you know what JavaScript String methods comes most often in problem solving like leetcode etc. then provide purpose, syntax what its takes what type of values user can provide its return type, core examples. for a method called "length" .

Here's a **straightforward breakdown** of the `length` property in JavaScript:

Purpose

Returns the number of characters in a string (including spaces and special characters).

Syntax

```
string.length
```

- `string` → any string value or variable.
- It does **not** take parameters (it's a property, not a method).

Return Type

- **Number** → the count of characters in the string.

Examples

```
console.log("hello".length);       // 5
console.log("".length);            // 0 (empty string)
console.log("hi there!".length);   // 9
console.log("💖".length);          // 2 (emoji counted as 2 code units)
```

👉 Would you like me to also cover **pitfalls** (like why `"💖".length` gives 2 instead of 1) since they often show up in tricky problems?

str.charAt(index)

JavaScript String Method: `charAt(index)`

- **Purpose**: Returns the character at a specific index in a string.

Syntax:

```
 str.charAt(index)
```

-
- **Parameters**:
  - `index` → **number** (0-based). Must be an integer.
  - If `index` is out of range → returns an empty string `""`.
- **Return type**: `string` (a single character, or `""`).

Core Examples:

```
let str = "LeetCode";

console.log(str.charAt(0));  // "L"
console.log(str.charAt(4));  // "C"
console.log(str.charAt(7));  // "e"
console.log(str.charAt(10)); // "" (out of range)
```

👉 Use when you need to safely access a single character from a string during problem solving.

Would you like me to also cover **`charCodeAt(index)`** since it often pairs with `charAt()` in coding problems?

charCodeAt(index)

`charCodeAt(index)` in JavaScript

Purpose:
Returns the Unicode (UTF-16) code of the character at a specified index in a string.

Syntax:

```
string.charCodeAt(index)
```

Parameters:

- `index` → *(Number)* Position of the character in the string (0-based).
    - If omitted or out of range → returns `NaN`.

Return Type:

- *(Number)* → Unicode code (integer between `0` and `65535`).

Examples:

```
let str = "ABC";

// Get code of 'A'
console.log(str.charCodeAt(0)); // 65

// Get code of 'C'
console.log(str.charCodeAt(2)); // 67

// Out of range
console.log(str.charCodeAt(5)); // NaN
```

👉 Mostly used in problems involving **character manipulation, ASCII conversions, frequency arrays, and hashing.**

Do you want me to also cover its relation with `String.fromCharCode()` since they're often used together in problem-solving?

at(index)

Here's a **straightforward breakdown** of JavaScript's `at(index)` string method, commonly used in problem-solving:

Purpose

Returns the character at the given index. Supports **negative indexing** (counts from the end).

Syntax

str.at(index)

Parameters

- **index** → `Number` (integer)
    - Positive → from start (`0` is first char).
    - Negative → from end (`-1` is last char).

Return Type

- `String` (the character at index).
- `undefined` if index is out of range.

Examples

```
let str = "leetcode";

// Positive index
console.log(str.at(0));    // "l"
console.log(str.at(3));    // "t"

// Negative index
console.log(str.at(-1));   // "e"
console.log(str.at(-3));   // "o"

// Out of range
console.log(str.at(20));   // undefined
console.log(str.at(-20));  // undefined
```

👉 `at(index)` is especially useful in **LeetCode problems** when accessing characters from both ends of a string.

Would you like me to also do the same **straightforward breakdown** for **array** **`.at(index)`**, since it behaves similarly and is equally common in coding problems?

slice(start, end)

String.prototype.slice(start, end)

- **Purpose**: Extracts a section of a string without modifying the original string.

Syntax:

```
str.slice(start, end)
```

-
- **Parameters**:
  - `start` *(number)* → index to begin extraction. Can be negative (counts from end).
  - `end` *(number, optional)* → index before which to stop. Can be negative. If omitted, extracts till end of string.
- **Return type**: `string` (new substring).

Examples:

```
let text = "LeetCode";
```

```
console.log(text.slice(0, 4));    // "Leet"
console.log(text.slice(4));       // "Code"
console.log(text.slice(-4));      // "Code"
console.log(text.slice(1, -1));   // "eetCod"
```

-

Would you like me to also cover **edge cases** (like when `start > end` or indices out of range) for this method?

substring(start, end)

Got it ✅ Here's a **straightforward breakdown** of **`substring(start, end)`** in JavaScript:

Purpose

Extracts characters from a string, starting at a given index up to (but not including) another index.

Syntax

```
string.substring(start, end)
```

Parameters

- **`start`** → integer (0 or positive), starting index (inclusive).
- **`end`** *(optional)* → integer (0 or positive), ending index (exclusive). If omitted, goes to end of string.

👉 Negative or `NaN` values are treated as `0`.
👉 If `start > end`, JavaScript swaps them.

Return Type

- Returns a **new string** (substring of the original).

Examples

```
let str = "leetcode";

// Basic usage
console.log(str.substring(0, 4));   // "leet"

// Without end → goes to end of string
console.log(str.substring(4));      // "code"

// If start > end → swapped automatically
console.log(str.substring(4, 2));   // "et"

// If start is negative → treated as 0
console.log(str.substring(-3, 2));  // "le"
```

Do you want me to also cover `slice(start, end)` **vs** `substring(start, end)` since they often confuse people in problem-solving?

substr(start, length)

`substr(start, length)` in JavaScript

Purpose:
 Extracts a portion of a string starting at a given index and of a specified length.

Syntax:

```
string.substr(start, length)
```

Parameters:

- `start` → (Number) Index at which to begin extraction. Can be negative (counts from the end).

- `length` → (Number, optional) Number of characters to extract. If omitted, extracts till end of string.

Return Type:

- Returns a **new string** containing the extracted part.

Core Examples:

```
let str = "LeetCode";

// Extract from index 0, length 4
console.log(str.substr(0, 4));   // "Leet"

// Extract from index 4, length 4
console.log(str.substr(4, 4));   // "Code"

// Negative start → counts from end
console.log(str.substr(-4, 2));  // "Co"

// Omitting length → till end
console.log(str.substr(4));      // "Code"
```

👉 Note: `substr()` is **deprecated**, prefer `substring()` or `slice()` in modern code, but it still works in many environments and appears in some problems.

Do you want me to also give you a quick comparison table (`substr` vs `substring` vs `slice`) for clarity?

indexOf(searchValue, fromIndex)

`String.prototype.indexOf(searchValue, fromIndex)`

- **Purpose**: Finds the first occurrence index of a substring in a string.

Syntax:

```
str.indexOf(searchValue, fromIndex)
```

-

- **Parameters**:
  - `searchValue` *(string | char)* → Substring/character to search.
  - `fromIndex` *(number, optional)* → Position to start searching from (default = `0`).
- **Return Type**:
  - `number` → Index of first match.
  - `-1` → If not found.

Examples:

```
 "hello".indexOf("e");       // 1
"hello".indexOf("l");       // 2
"hello".indexOf("l", 3);     // 3
"hello".indexOf("z");       // -1
"javascript".indexOf("script"); // 4
```

-

Would you like me to also cover the **edge cases** (like empty string searches, negative `fromIndex`)?

lastIndexOf(searchValue, fromIndex)

Got it ✅ Here's a **straightforward breakdown** of `lastIndexOf(searchValue, fromIndex)` in JavaScript:

Purpose

Finds the **last (rightmost) occurrence** of a substring inside a string.

Syntax

str.lastIndexOf(searchValue, fromIndex)

Parameters

- **searchValue** *(string)* → The substring to search for.
- **fromIndex** *(optional, number)* → Position in the string to start searching **backward**.
  - Default = `str.length - 1` (end of string).
  - If negative → treated as `0`.

Return Type

- **Number**
  - Index of the last occurrence.
  - `-1` if not found.

Examples

```
let text = "hello world hello";

// Last occurrence of "hello"
console.log(text.lastIndexOf("hello"));
// 12

// Search backward from index 10
console.log(text.lastIndexOf("hello", 10));
// 0

// Not found
console.log(text.lastIndexOf("bye"));
// -1
```

👉 Would you like me to also cover **common pitfalls** (like case-sensitivity, empty string behavior, etc.)?

includes(searchString, position)

Method: `includes(searchString, position)`

Purpose:
Checks if a string contains another string.

Syntax:

str.includes(searchString, position)

Parameters:

- `searchString` *(string)* → The substring to search for.
- `position` *(number, optional)* → The index to start searching from. Default is `0`.

Return Type:

- `boolean` → `true` if found, `false` otherwise.

Examples:

```
"hello world".includes("world");      // true
"hello world".includes("world", 7);   // false
"abc".includes("a");                   // true
"abc".includes("d");                   // false
```

Would you like me to make the same structured notes for **all most-used String methods** (like `substring`, `slice`, `indexOf`, `toLowerCase`, etc.) in LeetCode-style problems?

startsWith(searchString, position)

Method: `startsWith(searchString, position)`

- **Purpose:** Checks if a string starts with the given `searchString` from an optional `position`.
- **Syntax:**

```
str.startsWith(searchString[, position])
```

- **Parameters:**
  - `searchString` → **string** to search for. **Required**.
  - `position` → **integer** index to start checking from. **Optional**, default = 0.
- **Return Type:** `boolean` (`true` or `false`)
- **Core Examples:**

```
let str = "JavaScript";

// Check start
console.log(str.startsWith("Java")); // true

// Check start from index 4
console.log(str.startsWith("Script", 4)); // true

// Check with non-matching string
console.log(str.startsWith("script")); // false (case-sensitive)
```

This method is **case-sensitive** and widely used for prefix checks in problem-solving.

endsWith(searchString, length)

Method: `endsWith(searchString, length)`

- **Purpose:** Checks if a string ends with the specified substring (`searchString`). Optional `length` limits the string length to consider.
- **Syntax:**

```
str.endsWith(searchString[, length])
```

- **Parameters:**

    - `searchString` → **string** to search for at the end.
    - `length` → **optional number**, considers the string up to this length.
- **Return Type:** `boolean` (`true` or `false`)
- **Examples:**

```
let str = "Hello World";

str.endsWith("World");       // true
str.endsWith("Hello");       // false
str.endsWith("Hello", 5);    // true
str.endsWith("World", 11);   // true
str.endsWith("World", 10);   // false
```

This is commonly used in LeetCode for string suffix checks.

toLowerCase()

Method: `toLowerCase()`

- **Purpose:** Converts all characters in a string to lowercase.
- **Syntax:** `string.toLowerCase()`
- **Parameters:** None
- **Acceptable Values:** Works on any string.
- **Return Type:** `string` (new lowercase string)

Core Examples:

```
let str = "Hello World";
console.log(str.toLowerCase()); // "hello world"

console.log("JAVASCRIPT".toLowerCase()); // "javascript"
```

toUpperCase()

Method: `.toUpperCase()`

- **Purpose:** Converts all characters in a string to uppercase.
- **Syntax:** `str.toUpperCase()`
- **Parameters:** None
- **Value Types:** Called on a string; does not modify non-string values directly.
- **Return Type:** `String` (uppercase version of original string)
- **Examples:**

```
let text = "hello world";
console.log(text.toUpperCase()); // "HELLO WORLD"

console.log("JavaScript".toUpperCase()); // "JAVASCRIPT"
```

This is often used in LeetCode for case-insensitive comparisons or normalization.

trim()

`trim()`

- **Purpose:** Removes whitespace from both ends of a string.
- **Syntax:** `string.trim()`
- **Parameters:** None (does not take any arguments).
- **Return Type:** `string`
- **Example:**

```
let str = "   Hello World!   ";
let result = str.trim();
console.log(result); // "Hello World!"
```

- **Notes:** Only removes spaces, tabs, and other whitespace at the start and end; does not affect spaces between words.

trimStart() / trimEnd()

Here's a concise overview for `trimStart()` and `trimEnd()` in JavaScript:

Purpose:

Removes whitespace from the **start** (`trimStart()`) or **end** (`trimEnd()`) of a string.

Syntax:

```
str.trimStart()
str.trimEnd()
```

- **Parameters:** None
- **Accepts:** Only a string object
- **Return type:** String (new string with whitespace removed)

Core Examples:

```
let str1 = "   Hello World   ";
console.log(str1.trimStart()); // "Hello World   "
console.log(str1.trimEnd());   // "   Hello World"
```

If you want, I can make a **tiny reference table for all common JS string methods** like this for quick LeetCode use. Do you want me to do that?

replace(searchValue, newValue)

Method: `replace(searchValue, newValue)`

- **Purpose:** Replaces the first occurrence of `searchValue` in a string with `newValue`.
- **Syntax:**

```
string.replace(searchValue, newValue)
```

- **Parameters:**
    1. `searchValue` — string or regular expression (pattern to match).
    2. `newValue` — string or function (replacement value).
- **Return Type:** `string` (a new string, original string unchanged).
- **Examples:**

```
let str = "hello world";
console.log(str.replace("world", "JavaScript")); // "hello JavaScript"

let str2 = "foo foo foo";
console.log(str2.replace(/foo/, "bar")); // "bar foo foo" (only first match)

let str3 = "abc123";
console.log(str3.replace(/\d/, "9")); // "abc923"
```

This is commonly used in LeetCode or JS challenges for string transformations.

replaceAll(searchValue, newValue)

Method: `replaceAll(searchValue, newValue)`

- **Purpose:** Replaces **all occurrences** of `searchValue` in a string with `newValue`.
- **Syntax:**

```
str.replaceAll(searchValue, newValue)
```

- **Parameters:**
  1. `searchValue` – string or RegExp (global) to find.
  2. `newValue` – string or function to replace matches.
- **Return Type:** `String` (new string; original string unchanged).
- **Examples:**

```
let text = "apple banana apple";
console.log(text.replaceAll("apple", "orange"));
// Output: "orange banana orange"

let str = "123-456-123";
console.log(str.replaceAll(/\d{3}/g, "XXX"));
// Output: "XXX-XXX-XXX"
```

It's most useful in coding problems for **replacing multiple substrings efficiently**.

split(separator, limit)

Here's a concise breakdown for the JavaScript `split()` method:

Purpose:
Splits a string into an array of substrings based on a separator.

Syntax:

```
string.split(separator, limit)
```

Parameters:

- `separator` (optional) → String or RegExp used to determine where to split. If omitted, the entire string is returned as a single element array.
- `limit` (optional) → Integer specifying the maximum number of splits.

Return Type:

- Array of strings.

Examples:

```
let str = "apple,banana,cherry";

// Split by comma
console.log(str.split(","));
// Output: ["apple", "banana", "cherry"]

// Split by comma with limit
console.log(str.split(",", 2));
// Output: ["apple", "banana"]

// Split by empty string
console.log(str.split(""));
// Output: ["a","p","p","l","e",",","b","a","n","a","n","a",",","c","h","e","r","r","y"]

// No separator
console.log(str.split());
// Output: ["apple,banana,cherry"]
```

If you want, I can make a **quick reference table** of `split()` with **all edge cases and common patterns** used in problem solving. Do you want me to do that?

concat(...strings)

Method: `concat(...strings)`

- **Purpose:** Combines the calling string with one or more strings and returns a new string.
- **Syntax:**

```
str.concat(string1, string2, ..., stringN)
```

- **Parameters:**

    - `string1, string2, ..., stringN` → strings (or values converted to strings) to append.

- **Return Type:** `String` → new string with concatenated values.
- **Examples:**

```
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(" ", str2)); // "Hello World"

console.log("A".concat("B", "C")); // "ABC"

console.log("123".concat(456)); // "123456" (number converted to string)
```

This method **does not modify the original string**.

repeat(count)

`repeat(count)`

- **Purpose:** Returns a new string containing the original string repeated `count` times.
- **Syntax:**

```
str.repeat(count)
```

- **Parameters:**
    - `count` → A non-negative integer indicating how many times to repeat the string.
- **Value Type:** Number (integer ≥ 0)
- **Return Type:** String
- **Core Examples:**

```
"abc".repeat(3);     // "abcabcabc"
"hi".repeat(0);      // ""
```

```
"xyz".repeat(1);     // "xyz"
```

This is often used in formatting, generating repeated patterns, or padding output in coding challenges.

match(regex)

Method: `match()`

- **Purpose:** Searches a string for matches against a regular expression and returns the matches.
- **Syntax:** `str.match(regex)`
- **Parameter:**
  - `regex` → A `RegExp` object (can include flags like `g`, `i`, `m`).
- **Accepts:** Regular expressions.
- **Return Type:**
  - Array of matches if found, or `null` if no match.
  - With `g` flag → returns all matches as an array.
  - Without `g` flag → returns an array with match details and capturing groups.

Core Examples:

```
let str = "hello world 123";

// With global flag
console.log(str.match(/\d/g)); // ["1", "2", "3"]

// Without global flag
console.log(str.match(/\d/)); // ["1", index: 12, input: "hello world 123", groups: undefined]

// Using capturing group
console.log(str.match(/(hello) world/)); // ["hello world", "hello", index: 0, input: "hello world 123",
groups: undefined]
```

matchAll(regex)

Here's a concise breakdown of the JavaScript `matchAll()` method:

Purpose:
Returns an iterator of all matches of a **regular expression** in a string, including capturing groups. Useful when you need all matches with details.

Syntax:

```
string.matchAll(regex)
```

Parameters:

- `regex` — A `RegExp` object **with the global flag `g`**.
  - Without `g`, it throws a `TypeError`.

Return Type:

- `Iterator` of `RegExpMatchArray` objects (can be converted to array using `Array.from()`).

Example:

```
const str = "test1 test2 test3";
const regex = /test(\d)/g;

const matches = str.matchAll(regex);

for (const match of matches) {
  console.log(match[0], match[1]);
}
// Output:
// test1 1
// test2 2
// test3 3
```

Notes:

- Each match object contains the full match at `[0]` and capturing groups at `[1..n]`.
- To get all matches at once:

```
console.log([...str.matchAll(regex)]);
```

If you want, I can make a **mini cheat sheet** for all frequently used JavaScript string methods in the same style, optimized for problem solving like LeetCode. Do you want me to do that?

search(regex)

Method: `search(regex)`

- **Purpose:** Finds the index of the first match of a regular expression in a string.
- **Syntax:** `string.search(regexp)`
- **Parameters:**
  - `regexp` → A `RegExp` object or a string (if a string, it is converted to a `RegExp`).
- **Return Type:** `Number` → index of the first match, or `-1` if no match is found.
- **Examples:**

```
let str = "Hello World";
console.log(str.search(/World/)); // 6
console.log(str.search("Hello")); // 0
console.log(str.search(/abc/));   // -1
```

- **Notes:** Only finds the first occurrence; does not return the match itself.

padStart(targetLength, padString)

Method: `padStart(targetLength, padString)`

- **Purpose:** Pads the current string from the **start** with a given string until the resulting string reaches the specified length.
- **Syntax:**

```
str.padStart(targetLength, padString)
```

- **Parameters:**
  - `targetLength` (Number) – Total length of the resulting string.
  - `padString` (String, optional) – String to pad with (defaults to a space `" "`).
- **Accepted Values:**
  - `targetLength`: positive integer ≥ 0
  - `padString`: any string (empty string allowed, repeated as needed)
- **Return Type:** `String`
- **Examples:**

```
"5".padStart(3, "0");      // "005"
"abc".padStart(6, "*");    // "***abc"
"123".padStart(2);         // "123"  (no padding, already ≥ targetLength)
```

If you want, I can make a **super compact table of all frequently used JS String methods** for LeetCode-style problems. It'll be a quick reference.

padEnd(targetLength, padString)

Method: `padEnd(targetLength, padString)`

- **Purpose:** Pads the current string at the end with a given string until it reaches the specified length.
- **Syntax:**

```
str.padEnd(targetLength[, padString])
```

- **Parameters:**
    1. `targetLength` (Number) – The length of the resulting string after padding.
    2. `padString` (String, optional) – The string to pad with. Defaults to `' '`.
- **Accepts:** Numbers (for length), strings (for padding).
- **Return Type:** `String` – New string with padding applied.
- **Examples:**

```
'abc'.padEnd(6);       // "abc   "
'abc'.padEnd(6, '0');  // "abc000"
'abc'.padEnd(2);       // "abc" (no change, length already ≥ 2)
```

If you want, I can make a **quick table of 10 most common JS string methods with purpose, syntax, parameters, and examples** for LeetCode-style problems. It's super handy. Do you want me to do that?

localeCompare(compareString)

Here's a concise breakdown for `localeCompare()` in JavaScript:

- **Purpose:** Compares two strings in a locale-sensitive way.
- **Syntax:**

```
str.localeCompare(compareString, [locale, [options]])
```

- **Parameters:**
    - `compareString` (required) – string to compare with `str`.
    - `locale` (optional) – string or array of strings representing locale(s).

    - `options` (optional) – object with comparison options (`sensitivity`, `ignorePunctuation`, etc.).

- **Accepted Values:** Strings for comparison; optional locale strings like `"en-US"`.
- **Return Type:** `number`
    - `< 0` → `str` comes before `compareString`
    - `0` → strings are equivalent
    - `> 0` → `str` comes after `compareString`

Core Examples:

```
"apple".localeCompare("banana"); // -1 (apple < banana)
"apple".localeCompare("apple");  // 0  (equal)
"banana".localeCompare("apple"); // 1  (banana > apple)

// With locale and options
"résumé".localeCompare("resume", 'fr', { sensitivity: 'base' }); // 0 (accent ignored)
```

If you want, I can make a **mini table with all key string methods in JS like this**, ready for quick LeetCode reference. Do you want me to do that?

normalize(form)

Here's a concise overview of the JavaScript `String.prototype.normalize()` method:

Purpose:
 Normalizes Unicode characters in a string to a standard form (useful for comparing strings with accented characters).

Syntax:

```
str.normalize([form])
```

Parameters:

- `form` (optional) — a string specifying the normalization form. Can be:
  - `"NFC"` (default) – Canonical Composition
  - `"NFD"` – Canonical Decomposition
  - `"NFKC"` – Compatibility Composition
  - `"NFKD"` – Compatibility Decomposition

Accepts:

- Only strings (`str`) and optional form string.

Returns:

- A **new string** in normalized form.

Core Examples:

```
let s1 = 'é';
let s2 = 'e\u0301'; // e + combining acute accent

console.log(s1 === s2); // false

console.log(s1.normalize() === s2.normalize()); // true
console.log(s1.normalize('NFD')); // e +´
console.log(s1.normalize('NFC')); // é
```

This method is particularly useful in problems involving **string equality** where Unicode characters might be composed differently.

If you want, I can make a **mini cheat-sheet** for all commonly used JS string methods for LeetCode-style problems. Do you want me to do that?

valueOf()

Here's a concise overview for **`valueOf()`** in JavaScript:

- **Purpose:** Returns the primitive value of a String object.

Syntax:

```
 str.valueOf()
```

-
- **Parameters:** None.
- **Input Type:** Called on a String object or string literal.
- **Return Type:** Primitive string (`string`).

Core Examples:

```
let strObj = new String("Hello");
console.log(strObj.valueOf()); // "Hello"

let strLiteral = "World";
console.log(strLiteral.valueOf()); // "World"
```

It's mostly used internally; explicit use is rare since strings are usually already primitives.

toString()

Method: `toString()`

- **Purpose:** Converts a value (number, array, object, etc.) to a string.
- **Syntax:**

```
value.toString([radix])
```

- **Parameters:**
  - `radix` (optional, for numbers): An integer between 2 and 36 that represents the base of the number system.
- **Accepts:** Numbers, Booleans, Arrays, Objects, etc.

- **Returns:** String representation of the value.

Core Examples:

```
// Number to string
let num = 123;
console.log(num.toString()); // "123"

// Number to binary string
console.log(num.toString(2)); // "1111011"

// Boolean to string
let bool = true;
console.log(bool.toString()); // "true"

// Array to string
let arr = [1,2,3];
console.log(arr.toString()); // "1,2,3"

// Object to string
let obj = {a:1};
console.log(obj.toString()); // "[object Object]"
```

If you want, I can create a **tiny table of all JS string methods used in LeetCode with purpose + examples** for super quick reference. Do you want me to do that?

fromCharCode(...codes)

Method: `String.fromCharCode(...codes)`

- **Purpose:** Converts one or more Unicode values (UTF-16 code units) into a string.
- **Syntax:**

```
String.fromCharCode(num1, num2, ..., numN)
```

- **Parameters:**

    - `num1, num2, ...` → Numbers representing UTF-16 code units. Can be integers or expressions that resolve to numbers.

- **Return Type:** `string`
- **Core Examples:**

```
// Single character
console.log(String.fromCharCode(65)); // "A"

// Multiple characters
console.log(String.fromCharCode(72, 101, 108, 108, 111)); // "Hello"

// Using expressions
console.log(String.fromCharCode(65 + 1)); // "B"
```

This method is often used to generate characters dynamically from Unicode values.

If you want, I can make a **one-page reference for all top JS string methods used in coding problems**. Do you want me to do that?

fromCodePoint(...codes)

`String.fromCodePoint(...codes)`

- **Purpose:** Creates a string from one or more Unicode code points. Useful for handling characters outside the Basic Multilingual Plane.
- **Syntax:**

```
String.fromCodePoint(codePoint1, codePoint2, ..., codePointN)
```

- **Parameters:**
    - `codePoint1, codePoint2, ...` → One or more numbers representing Unicode code points (integers between 0 and 0x10FFFF).
- **Return Type:**

   - `string` → A string containing the specified characters.

- **Core Examples:**

```
console.log(String.fromCodePoint(65));       // "A"
console.log(String.fromCodePoint(9731, 9733, 9842)); // "☃★♲"
console.log(String.fromCodePoint(0x1F600));  // "😀" (emoji)
```

This method is static and called on `String`, not a string instance.

raw(template, ...substitutions)

Here's a concise breakdown for `String.raw()` in JavaScript:

Purpose:
 Creates a raw string from a template literal, preserving backslashes (`\`) without interpreting escape sequences. Often used in regex patterns or file paths.

Syntax:

```
String.raw(template, ...substitutions)
```

Parameters:

- `template`: A template literal object (usually from a tagged template).
- `...substitutions`: Values to replace `${}` placeholders in the template.

Accepted Types:

- `template`: Object with a `raw` property (array of strings).
- `substitutions`: Any value (numbers, strings, expressions).

Return Type:

- `string` — raw string with substitutions applied.

Core Examples:

```
// Basic usage
console.log(String.raw`Line1\nLine2`);
// Output: Line1\nLine2

// With substitutions
let name = 'Alice';
console.log(String.raw`Hello\n${name}!`);
// Output: Hello\nAlice!

// Using String.raw() as function
console.log(String.raw({raw: ['foo','bar']}, 42));
// Output: foo42bar
```

It's mostly used to preserve exact formatting in strings.

If you want, I can make a **mini table of top JS string methods like this** for LeetCode-style usage. Do you want me to do that?

codePointAt(index)

`codePointAt(index)`

- **Purpose:** Returns the Unicode code point value of the character at the specified position in a string.

Syntax:

```
str.codePointAt(index)
```

-
- **Parameters:**
    - `index` (Number) — Position of the character (0-based). Must be within `[0, str.length - 1]`.
- **Acceptable Values:** Integer index within string length.
- **Return Type:** Number (Unicode code point) or `undefined` if index is out of range.

Examples:

```
 let str = 'Hello';
console.log(str.codePointAt(0)); // 72 ('H')
console.log(str.codePointAt(4)); // 111 ('o')
console.log('💖'.codePointAt(0)); // 128150 (surrogate pair)
console.log(str.codePointAt(10)); // undefined (out of range)
```

-

Straightforward, used for Unicode-aware character processing.

anchor(name), big(), blink(), bold(), fixed(), fontcolor(), fontsize(), italics(), link(url), small(), strike(), sub(), sup()

Here’s a concise summary for the requested **JavaScript string methods** (mostly legacy HTML formatting methods):

Method	Purpose	Syntax	Parameters	Return Type	Example
`anchor(name)`	Creates an `<a>` HTML anchor	`str.anchor(name)`	`name` → string	string	`"hello".anchor("top")` → `<a name="top">hello</a>`
`big()`	Makes text appear bigger (`<big>`)	`str.big()`	none	string	`"text".big()` → `<big>text</big>`
`blink()`	Makes text blink (`<blink>`)	`str.blink()`	none	string	`"text".blink()` → `<blink>text</blink>`

`bold()`	Makes text bold (`<b>`)	`str.bold()`	none	string	`"text".bold()` → `<b>text</b>`
`fixed()`	Displays text in monospace (`<tt>`)	`str.fixed()`	none	string	`"text".fixed()` → `<tt>text</tt>`
`fontcolor(color)`	Sets text color (`<font color="...">`)	`str.fontcolor(color)`	`color` → string (color name or hex)	string	`"text".fontcolor("red")` → `<font color="red">text</font>`
`fontsize(size)`	Sets font size (`<font size="...">`)	`str.fontsize(size)`	`size` → number/string	string	`"text".fontsize(5)` → `<font size="5">text</font>`
`italics()`	Makes text italic (`<i>`)	`str.italics()`	none	string	`"text".italics()` → `<i>text</i>`
`link(url)`	Creates a hyperlink (`<a href="...">`)	`str.link(url)`	`url` → string	string	`"Google".link("https://google.com")` → `<a href="https://google.com">Google</a>`
`small()`	Makes text smaller (`<small>`)	`str.small()`	none	string	`"text".small()` → `<small>text</small>`
`strike()`	Strikethrough text (`<strike>`)	`str.strike()`	none	string	`"text".strike()` → `<strike>text</strike>`

`sub()`	Subscript text (`<sub>`)	`str.sub()`	none	string	`"text".sub()` → `<sub>text</sub>`
`sup()`	Superscript text (`<sup>`)	`str.sup()`	none	string	`"text".sup()` → `<sup>text</sup>`

✅ **Notes:**

- These methods **return a new string** with HTML tags.
- Mostly **deprecated** in modern JS/HTML; used primarily for legacy or quick HTML embedding.

If you want, I can make a **1-line "cheat sheet" with syntax + return** for all of these that's LeetCode-ready. Do you want me to do that?