Strings

Adib Sakhawat
IUT CSE ‘21

Strings in JavaScript

Strings are one of the fundamental data types in JavaScript, used to represent text. They are immutable sequences of Unicode characters and come with a rich set of methods for manipulation and inspection. This guide will cover how to create and manipulate strings, along with detailed explanations and code examples of common string methods.

Creating Strings

In JavaScript, strings can be created using:

1. String Literals

You can define strings by enclosing text within single quotes `' '`, double quotes `" "`, or backticks `` ` ` ``.

```
const singleQuoteString = 'Hello, World!';
const doubleQuoteString = "Hello, World!";
const templateString = `Hello, World!`;
```

Edge Case: Quotes Inside Strings

To include quotes inside strings, you can:

- Escape them using a backslash `\\`.

  ```
  const quote = 'He said, "It\\'s a sunny day."';
  ```

- Use different types of quotes to enclose the string.

  ```
  const quote = "He said, 'It's a sunny day.'";
  ```

2. String Objects

You can also create strings using the `String` constructor.

```
const stringObject = new String('Hello, World!');
console.log(typeof stringObject); // Output: 'object'
```

Note: It's generally recommended to use string literals for simplicity and performance. String objects can introduce unexpected behavior when comparing strings.

String Immutability

Strings in JavaScript are **immutable**, meaning once created, they cannot be altered. Operations that appear to modify a string actually create a new string.

```
let greeting = 'Hello';
greeting[0] = 'J';
console.log(greeting); // Output: 'Hello' (unchanged)
```

Accessing Characters

You can access individual characters in a string using:

1. Bracket Notation

```
const word = 'JavaScript';
console.log(word[0]); // Output: 'J'
console.log(word[4]); // Output: 'S'
```

2. `charAt()` Method

```
console.log(word.charAt(0)); // Output: 'J'
console.log(word.charAt(4)); // Output: 'S'
```

Edge Case: Out-of-Bounds Indices

```
console.log(word[100]);        // Output: undefined
console.log(word.charAt(100)); // Output: '' (empty string)
```

Common String Properties and Methods

1. `length` Property

Returns the length of the string.

```
const text = 'Hello, World!';
console.log(text.length); // Output: 13
```

2. `indexOf()` Method

Returns the index of the **first occurrence** of a specified value. Returns `-1` if not found.

```
const text = 'Hello, World!';
console.log(text.indexOf('o'));        // Output: 4
console.log(text.indexOf('World'));    // Output: 7
console.log(text.indexOf('world'));    // Output: -1 (case-s
ensitive)
```

Using `indexOf()` with Starting Position

```
console.log(text.indexOf('o', 5)); // Output: 8 (search sta
rts at index 5)
```

Edge Case: Non-Existent Substring

```
console.log(text.indexOf('z')); // Output: -1
```

3. `lastIndexOf()` Method

Returns the index of the **last occurrence** of a specified value.

```
const text = 'Hello, World!';
console.log(text.lastIndexOf('o')); // Output: 8
```

4. `slice()` Method

Extracts a section of a string and returns it as a new string.

```
const text = 'Hello, World!';
const slicedText = text.slice(7, 12);
console.log(slicedText); // Output: 'World'
```

Parameters:

- `start`: Zero-based index at which to begin extraction.
- `end` (optional): Zero-based index before which to end extraction. If omitted, extracts to the end of the string.

Using Negative Indices

Negative indices count from the end of the string.

```
const lastWord = text.slice(-6, -1);
console.log(lastWord); // Output: 'World'
```

Edge Cases:

- If `start` is greater than `end`, `slice()` returns an empty string.

  ```
  console.log(text.slice(12, 7)); // Output: ''
  ```

- If `start` or `end` are out of bounds, they are clamped to the string's length.

  ```
  console.log(text.slice(7, 50)); // Output: 'World!'
  ```

---

5. `substring()` Method

Similar to `slice()`, but doesn't accept negative indices.

```
const text = 'Hello, World!';
const subText = text.substring(7, 12);
console.log(subText); // Output: 'World'
```

Differences from `slice()`:

- If `start` is greater than `end`, `substring()` swaps them.

  ```
  console.log(text.substring(12, 7)); // Output: 'World'
  ```

- Negative values are treated as `0`.

  ```
  console.log(text.substring(-5, 5)); // Output: 'Hello'
  ```

---

6. `substr()` Method

Extracts a substring starting from a specified index for a given length.

```
const text = 'Hello, World!';
const subText = text.substr(7, 5);
console.log(subText); // Output: 'World'
```

Parameters:

- `start`: The starting index.
- `length`: Number of characters to extract.

Note: `substr()` is considered a legacy feature and may not be supported in all environments. Prefer using `slice()` or `substring()`.

---

7. `replace()` Method

Returns a new string with some or all matches of a pattern replaced by a replacement.

```
const text = 'Hello, World!';
const newText = text.replace('World', 'JavaScript');
console.log(newText); // Output: 'Hello, JavaScript!'
```

Parameters:

- `pattern`: The substring or RegExp to match.
- `replacement`: The string to replace the matches with.

Replacing All Occurrences:

- Use a global regular expression.

```
const text = 'apple, banana, apple';
const newText = text.replace(/apple/g, 'orange');
console.log(newText); // Output: 'orange, banana, orang
e'
```

Using a Function as Replacement:

```
const text = 'Hello, World!';
const newText = text.replace(/(\\w+), (\\w+)!/, (match, p1,
p2) => {
  return `Hi, ${p2} and ${p1}!`;
});
console.log(newText); // Output: 'Hi, World and Hello!'
```

8. `concat()` Method

Concatenates two or more strings.

```
const greeting = 'Hello';
const name = 'Alice';
const message = greeting.concat(', ', name, '!');
console.log(message); // Output: 'Hello, Alice!'
```

Note: The `+` operator is more commonly used for string concatenation.

```
const message = greeting + ', ' + name + '!';
```

9. `toUpperCase()` and `toLowerCase()` Methods

Convert the string to uppercase or lowercase.

```
const text = 'Hello, World!';
console.log(text.toUpperCase()); // Output: 'HELLO, WORLD!'
console.log(text.toLowerCase()); // Output: 'hello, world!'
```

10. `trim()` Method

Removes whitespace from both ends of a string.

```
const text = '   Hello, World!   ';
console.log(text.trim()); // Output: 'Hello, World!'
```

Related Methods:

- `trimStart()` / `trimLeft()`: Trims whitespace from the start.
- `trimEnd()` / `trimRight()`: Trims whitespace from the end.

---

11. `split()` Method

Splits a string into an array of substrings.

```
const text = 'apple, banana, cherry';
const fruits = text.split(', ');
console.log(fruits); // Output: ['apple', 'banana', 'cherr
y']
```

Parameters:

- `separator`: Specifies the character or regular expression to use for splitting.
- `limit` (optional): An integer that limits the number of splits.

Edge Case: Splitting into Individual Characters

```
const word = 'JavaScript';
const letters = word.split('');
console.log(letters);
// Output: ['J', 'a', 'v', 'a', 'S', 'c', 'r', 'i', 'p',
't']
```

---

12. `includes()` Method

Determines whether a string contains a specified substring.

```
const text = 'Hello, World!';
console.log(text.includes('World')); // Output: true
console.log(text.includes('world')); // Output: false (case
-sensitive)
```

13. `startsWith()` and `endsWith()` Methods

Check if a string starts or ends with a specified substring.

```
const filename = 'document.pdf';
console.log(filename.endsWith('.pdf'));   // Output: true
console.log(filename.startsWith('doc'));  // Output: true
```

Parameters:

- `searchString`: The substring to search for.
- `position` (optional): For `startsWith()`, the position in the string at which to begin searching.

14. `repeat()` Method

Returns a new string with a specified number of copies of the original string.

```
const pattern = '*';
console.log(pattern.repeat(5)); // Output: '*****'
```

String Templates (Template Literals)

Template literals allow embedded expressions and multi-line strings using backticks `` ` ``.

```
const name = 'Alice';
const greeting = `Hello, ${name}!`;
console.log(greeting); // Output: 'Hello, Alice!'
```

Multi-line Strings:

```
const message = `This is a
multi-line
string.`;
```

```
console.log(message);
// Output:
// 'This is a
// multi-line
// string.'
```

Expressions in Templates:

```
const a = 5;
const b = 10;
console.log(`The sum of a and b is ${a + b}.`); // Output:
'The sum of a and b is 15.'
```

String Comparison

Strings can be compared using relational operators.

```
console.log('apple' === 'apple'); // Output: true
console.log('apple' === 'Apple'); // Output: false (case-se
nsitive)
console.log('apple' > 'banana');  // Output: false (lexicog
raphical order)
```

Escaping Special Characters

Certain characters need to be escaped with a backslash `\\` when included in strings.

- **Newline:** `\\n`
- **Tab:** `\\t`
- **Backslash:** `\\\\`
- **Single Quote:** `\\'`
- **Double Quote:** `\\"`

```
const text = 'She said, "It\\'s a beautiful day."\\nLet\\'s
go outside.';
console.log(text);
```

```
// Output:
// She said, "It's a beautiful day."
// Let's go outside.
```

String and Unicode

JavaScript strings are sequences of UTF-16 code units.

Accessing Code Units

```
const emoji = '😊';
console.log(emoji.charCodeAt(0)); // Output: 55357
console.log(emoji.charCodeAt(1)); // Output: 56842
```

Edge Case: Characters Represented by Surrogate Pairs

Some characters (like emojis) are represented using two UTF-16 code units.

Using `codePointAt()`

For full Unicode code points.

```
console.log(emoji.codePointAt(0)); // Output: 128522
```

Regular Expressions with Strings

Strings can be manipulated using regular expressions.

Using `match()` Method

```
const text = 'The rain in SPAIN stays mainly in the plai
n.';
const matches = text.match(/ain/g);
console.log(matches); // Output: ['ain', 'ain', 'ain']
```

Using `search()` Method

Returns the index of the first match.

```
const index = text.search(/ain/);
console.log(index); // Output: 5
```

Using `replace()` with Regular Expressions

```
const newText = text.replace(/ain/g, '***');
console.log(newText);
// Output: 'The r*** in SPAIN stays m***ly in the pl***.'
```

Edge Cases and Best Practices

1. String Object vs. String Literal

Avoid using `new String()` as it creates a string object, not a primitive string.

```
const strLiteral = 'Hello';
const strObject = new String('Hello');

console.log(typeof strLiteral); // Output: 'string'
console.log(typeof strObject);  // Output: 'object'

console.log(strLiteral === 'Hello'); // Output: true
console.log(strObject === 'Hello');  // Output: false
```

2. Immutability and Performance

Since strings are immutable, methods like `replace()`, `slice()`, etc., return new strings. Be mindful of performance when performing many string operations.

3. Concatenating Large Strings

For concatenating many strings, especially in loops, consider using arrays and `join()`.

```
let longString = '';
for (let i = 0; i < 1000; i++) {
  longString += 'a';
}
```

```
// Alternatively
const array = new Array(1000).fill('a');
const longString = array.join('');
```

4. Handling Non-BMP Characters

Characters outside the Basic Multilingual Plane (BMP), like some emojis, require special handling.

```
const text = '𠮷';
console.log(text.length);          // Output: 2
console.log(text.charAt(0));       // Output: '\\uD842'
console.log(text.charAt(1));       // Output: '\\uDFB7'
console.log(text.codePointAt(0));  // Output: 134071
```

Use libraries like ES6 `String` methods for better support.

Converting Other Types to Strings

Using `String()` Function

```
const num = 123;
const str = String(num);
console.log(str);          // Output: '123'
console.log(typeof str);   // Output: 'string'
```

Using `toString()` Method

```
const bool = true;
const str = bool.toString();
console.log(str);          // Output: 'true'
console.log(typeof str);   // Output: 'string'
```

Edge Case: `null` and `undefined`

```
console.log(String(null));       // Output: 'null'
console.log(String(undefined));  // Output: 'undefined'
```

```
// Using toString() directly will cause an error
// console.log(null.toString()); // Error
// console.log(undefined.toString()); // Error
```

Template Literals vs. String Concatenation

Template literals provide a cleaner syntax for embedding expressions.

String Concatenation

```
const name = 'Bob';
const age = 25;
const message = 'My name is ' + name + ' and I am ' + age +
' years old.';
```

Template Literal

```
const message = `My name is ${name} and I am ${age} years o
ld.`;
```

Advantages of Template Literals:

- Easier to read and write.
- Support multi-line strings.
- Allow expressions and function calls inside `${}`.