JavaScript Basics

**String**

The String object is used to represent and manipulate a sequence of characters.

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their length, to build and concatenate them using the + and += string operators, checking for the existence or location of substrings with the indexOf() method, or extracting substrings with the substring() method.

Different ways to create string in js

const string1 = "A string primitive";

const string2 = 'Also a string primitive';

const string3 = `Yet another string primitive`;

const string4 = new String("A String object");

how to use variable in between the strings

console.log(` My Name is ${Name}`);

**String() constructor**

The String() constructor creates [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) objects. When called as a function, it returns primitive values of type String.

[Syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/String#syntax)

new String(thing)

String(thing)

const a = new String("Hello world"); // a === "Hello world" is false

const b = String("Hello world"); // b === "Hello world" is true

a instanceof String; // is true

b instanceof String; // is false

typeof a; // "object"

typeof b; // "string"

**String Method**

### [JavaScript String length](https://www.programiz.com/javascript/library/string/length)

Returns the number of characters in a string

### [JavaScript String replace()](https://www.programiz.com/javascript/library/string/replace)

replace a substring/pattern in the string

### [JavaScript String indexOf()](https://www.programiz.com/javascript/library/string/indexof)

Returns the first index of occurrence of a value

### [JavaScript String lastIndexOf()](https://www.programiz.com/javascript/library/string/lastindexof)

Returns the last index of occurrence of a value

### [Javascript String startsWith()](https://www.programiz.com/javascript/library/string/startswith)

Checks if a string begins with a specified string

### [Javascript String endsWith()](https://www.programiz.com/javascript/library/string/endswith)

Checks if a string ends with a specified string

### [Javascript String toUpperCase()](https://www.programiz.com/javascript/library/string/touppercase)

Returns uppercase of a string

### [Javascript String toLowerCase()](https://www.programiz.com/javascript/library/string/tolowercase)

Returns lowercase representation of a string

### [Javascript String includes()](https://www.programiz.com/javascript/library/string/includes)

Checks if given string is found inside a string

### [JavaScript String repeat()](https://www.programiz.com/javascript/library/string/repeat)

Returns a string by repeating it given times

### [JavaScript String charAt()](https://www.programiz.com/javascript/library/string/charat)

Returns character at a specified index in string

### [JavaScript String charCodeAt()](https://www.programiz.com/javascript/library/string/charcodeat)

Returns Unicode of the character at given index

### [JavaScript String fromCharCode()](https://www.programiz.com/javascript/library/string/fromcharcode)

Returns a string from the given UTF-16 code units

### [Javascript String substring()](https://www.programiz.com/javascript/library/string/substring)

Returns a specified part of the string

### [Javascript String padStart()](https://www.programiz.com/javascript/library/string/padstart)

Pads a string at the start to a given length

### [Javascript String padEnd()](https://www.programiz.com/javascript/library/string/padend)

Pads a string at the end to a given length

### [JavaScript String codePointAt()](https://www.programiz.com/javascript/library/string/codepointat)

Returns the Unicode point value at given index

### [JavaScript String fromCodePoint()](https://www.programiz.com/javascript/library/string/fromcodepoint)

Returns a string using the given code points

### [Javascript String match()](https://www.programiz.com/javascript/library/string/match)

Returns result of matching string with a regex

### [Javascript String matchAll()](https://www.programiz.com/javascript/library/string/matchall)

Returns iterator of results matching with a regex

### [Javascript String localeCompare()](https://www.programiz.com/javascript/library/string/localeCompare)

Compares two strings in the current locale

### [Javascript String search()](https://www.programiz.com/javascript/library/string/search)

Searches for specified value in the string

### [JavaScript String replaceAll()](https://www.programiz.com/javascript/library/string/replaceAll)

Returns string by replacing all matching patterns

### [JavaScript String concat()](https://www.programiz.com/javascript/library/string/concat)

Concatenates the arguments to the calling string

### [JavaScript String split()](https://www.programiz.com/javascript/library/string/split)

Returns the string divided into list of substring

### [JavaScript String trim()](https://www.programiz.com/javascript/library/string/trim)

Removes whitespace from both ends of a string

### [JavaScript String slice()](https://www.programiz.com/javascript/library/string/slice)

Extracts and returns a section of the string

**Important String methods**

1. [charAt(x)](https://www.edureka.co/blog/javascript-string-functions/#charAt(x))
2. [charCodeAt(x)](https://www.edureka.co/blog/javascript-string-functions/#charCodeAt(x))
3. [concat(v1,v2..)](https://www.edureka.co/blog/javascript-string-functions/#concat(v1,v2..))
4. [fromCharcode(c1,c2)](https://www.edureka.co/blog/javascript-string-functions/#fromCharcode(c1,c2))
5. [indexOf(substr, [start])](https://www.edureka.co/blog/javascript-string-functions/#indexOf(substr,%5Bstart%5D))
6. [lastIndexOf(substr, [start])](https://www.edureka.co/blog/javascript-string-functions/#lastIndexOf(substr,%5Bstart%5D))
7. [match(regexp)](https://www.edureka.co/blog/javascript-string-functions/#match(regexp))
8. [replace(regexp/substr, replacetext)](https://www.edureka.co/blog/javascript-string-functions/#replace(regexp/substr,replacetext))
9. [search(regexp)](https://www.edureka.co/blog/javascript-string-functions/#search(regexp))
10. [slice(start, [end])](https://www.edureka.co/blog/javascript-string-functions/#slice(start,%5Bend%5D))
11. [split(delimiter, [limit])](https://www.edureka.co/blog/javascript-string-functions/#split(delimiter,%5Blimit%5D))
12. [substr(start, [length])](https://www.edureka.co/blog/javascript-string-functions/#substr(start,%20%5Blength%5D))
13. [substring(from, [to])](https://www.edureka.co/blog/javascript-string-functions/#substring(from,%20%5Bto%5D))
14. [toLowerCase()](https://www.edureka.co/blog/javascript-string-functions/#toLowerCase())
15. [toUpperCase()](https://www.edureka.co/blog/javascript-string-functions/#toUpperCase())
16. [includes()](https://www.edureka.co/blog/javascript-string-functions/#includes)
17. [endsWith()](https://www.edureka.co/blog/javascript-string-functions/#endswith)
18. [repeat()](https://www.edureka.co/blog/javascript-string-functions/#repeat)
19. [valueOf()](https://www.edureka.co/blog/javascript-string-functions/#linkValueOf)
20. [trim()](https://www.edureka.co/blog/javascript-string-functions/#trim)

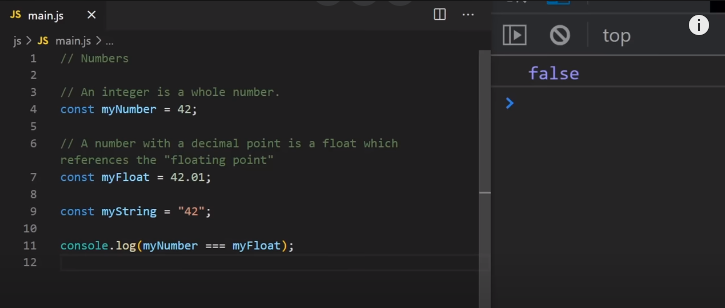
**Numbers**

Number values represent floating-point numbers like 37 or -9.25.

The Number constructor contains constants and methods for working with numbers. Values of other types can be converted to numbers using the Number() function.

Description

Numbers are most commonly expressed in literal forms like 255 or 3.14159. The lexica grammar contains a more detailed reference.



JS

255; // two-hundred and fifty-five

255.0; // same number

255 === 255.0; // true

255 === 0xff; // true (hexadecimal notation)

255 === 0b11111111; // true (binary notation)

255 === 0.255e3; // true (decimal exponential notation)

A number literal like 37 in JavaScript code is a floating-point value, not an integer. There is no separate integer type in common everyday use. (JavaScript also has a BigInt type, but it's not designed to replace Number for everyday uses. 37 is still a number, not a BigInt.)

When used as a function, Number(value) converts a string or other value to the Number type. If the value can't be converted, it returns NaN.

JS

Number("123"); // returns the number 123

Number("123") === 123; // true

Number("unicorn"); // NaN

Number(undefined); // NaN

### [Number coercion](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_coercion)

Many built-in operations that expect numbers first coerce their arguments to numbers (which is largely why Number objects behave similarly to number primitives). [The operation](https://tc39.es/ecma262/multipage/abstract-operations.html#sec-tonumber) can be summarized as follows:

* Numbers are returned as-is.
* [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) turns into [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN).
* [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) turns into 0.
* true turns into 1; false turns into 0.
* Strings are converted by parsing them as if they contain a [number literal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#numeric_literals). Parsing failure results in NaN. There are some minor differences compared to an actual number literal:
  + Leading and trailing whitespace/line terminators are ignored.
  + A leading 0 digit does not cause the number to become an octal literal (or get rejected in strict mode).
  + + and - are allowed at the start of the string to indicate its sign. (In actual code, they "look like" part of the literal, but are actually separate unary operators.) However, the sign can only appear once, and must not be followed by whitespace.
  + Infinity and -Infinity are recognized as literals. In actual code, they are global variables.
  + Empty or whitespace-only strings are converted to 0.
  + [Numeric separators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#numeric_separators) are not allowed.
* [BigInts](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) throw a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError) to prevent unintended implicit coercion causing loss of precision.
* [Symbols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) throw a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError).
* Objects are first [converted to a primitive](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_coercion) by calling their [[@@toPrimitive]()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/toPrimitive) (with "number" as hint), valueOf(), and toString() methods, in that order. The resulting primitive is then converted to a number.

The Number() constructor creates Number objects. When called as a function, it returns primitive values of type Number.

Syntax

JS

new Number(value)

Number(value)

JS

const a = new Number("123"); // a === 123 is false

const b = Number("123"); // b === 123 is true

a instanceof Number; // is true

b instanceof Number; // is false

typeof a; // "object"

typeof b; // "number"

Number Function VS Number Constructor

When Number is called as a constructor (with new ), it creates a Number object, which is not a primitive. When Number is called as a function, it coerces the parameter to a number primitive. BigInts are converted to numbers. If the value can't be converted, it returns NaN

**Conditional Statement**

**If..else …else if statement:**

### **Syntax**

The syntax of an if-else-if statement is as follows −

if (expression 1) {

Statement(s) to be executed if expression 1 is true

} else if (expression 2) {

Statement(s) to be executed if expression 2 is true

} else if (expression 3) {

Statement(s) to be executed if expression 3 is true

} else {

Statement(s) to be executed if no expression is true

}

**Switch statement:**

switch (expression) {

case condition 1: statement(s)

break;

case condition 2: statement(s)

break;

...

case condition n: statement(s)

break;

default: statement(s)

}

**Tenary Operator**

Syntax

JS

condition ? exprIfTrue : exprIfFalse

example:

function getFee(isMember) {

return (isMember ? '$2.00' : '$10.00');

}

console.log(getFee(true));

// Expected output: "$2.00"

console.log(getFee(false));

// Expected output: "$10.00"

console.log(getFee(null));

// Expected output: "$10.00"

### [A simple example](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator#a_simple_example)

JS

const age = 26;

const beverage = age >= 21 ? "Beer" : "Juice";

console.log(beverage); // "Beer"

### [Handling null values](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator#handling_null_values)

One common usage is to handle a value that may be null:

JS

const greeting = (person) => {

const name = person ? person.name : "stranger";

return `Howdy, ${name}`;

};

console.log(greeting({ name: "Alice" })); // "Howdy, Alice"

console.log(greeting(null)); // "Howdy, stranger"

### [Conditional chains](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator#conditional_chains)

The ternary operator is right-associative, which means it can be "chained" in the following way, similar to an if … else if … else if … else chain:

JS

function example() {

return condition1 ? value1

: condition2 ? value2

: condition3 ? value3

: value4;

}

This is equivalent to the following [if...else](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) chain.

JS

function example() {

if (condition1) {

return value1;

} else if (condition2) {

return value2;

} else if (condition3) {

return value3;

} else {

return value4;

}

}

# Loops and iteration

The statements for loops provided in JavaScript are: with links

* [for statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for_statement)
* [do...while statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#do...while_statement)
* [while statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#while_statement)
* [labeled statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#labeled_statement)
* [break statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#break_statement)
* [continue statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#continue_statement)
* [for...in statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for...in_statement)
* [for...of statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for...of_statement)

Created Separate File for loops and iteration for in-depth Knowledge

# Template literals (Template strings or Backticks)

# Template literals are literals delimited with backtick (`) characters, allowing for multi-line strings, string interpolation with embedded expressions, and special constructs called tagged templates.

# Template literals are sometimes informally called template strings, because they are used most commonly for string interpolation (to create strings by doing substitution of placeholders). However, a tagged template literal may not result in a string; it can be used with a custom tag function to perform whatever operations you want on the different parts of the template literal.

JS

`string text`

`string text line 1

string text line 2`

`string text ${expression} string text`

tagFunction`string text ${expression} string text`

To escape a backtick in a template literal, put a backslash (\) before the backtick.

`\`` === "`"; // true

Dollar signs can be escaped as well to prevent interpolation.

`\${1}` === "${1}"; // true

**Multi-line strings**

Any newline characters inserted in the source are part of the template literal.

Using normal strings, you would have to use the following syntax in order to get multi-line strings:

JS

console.log("string text line 1\n" + "string text line 2");

// "string text line 1

// string text line 2"

Using template literals, you can do the same with this:

JS

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

String interpolation

Without template literals, when you want to combine output from expressions with strings, you'd concatenate them using the addition operator +:

JS

const a = 5;

const b = 10;

console.log("Fifteen is " + (a + b) + " and\nnot " + (2 \* a + b) + ".");

// "Fifteen is 15 and

// not 20."

That can be hard to read – especially when you have multiple expressions.

With template literals, you can avoid the concatenation operator — and improve the readability of your code — by using placeholders of the form ${expression} to perform substitutions for embedded expressions:

JS

const a = 5;

const b = 10;

console.log(`Fifteen is ${a + b} and

not ${2 \* a + b}.`);

// "Fifteen is 15 and

// not 20."