**Strings**

In JavaScript, the textual data is stored as strings. There is no separate type for a single character.

**[Quotes](https://javascript.info/string" \l "quotes)**

Let’s remember the kinds of quotes.

Strings can be enclosed either with the single, double quotes or in backticks:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Single and double quotes are essentially the same. Backticks allow to embed any expression into the string, including function calls:

function sum(a, b) {

return a + b;

}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.

Another advantage of using backticks is that they allow a string to span multiple lines:

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList); // a list of guests, multiple lines

If we try to use single or double quotes the same way, there will be an error:

let guestList = "Guests: // Error: Unexpected token ILLEGAL

\* John";

Single and double quotes come from ancient times of language creation, and the need for multiline strings was not taken into account. Backticks appeared much later and thus are more versatile.

**[Special characters](https://javascript.info/string" \l "special-characters)**

It is still possible to create multiline strings with single quotes, using a so-called “newline character” written as \n, that denotes a line break:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";

alert(guestList); // a multiline list of guests

So to speak, these two lines describe the same:

alert( "Hello\nWorld" ); // two lines using a "newline symbol"

// two lines using a normal newline and backticks

alert( `Hello

World` );

There are other, less common “special” characters as well, here’s the list:

| **Character** | **Description** |
| --- | --- |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Tab |
| \uNNNN | A unicode symbol with the hex code NNNN, for instance \u00A9 – is a unicode for the copyright symbol ©. Must be exactly 4 hex digits. |
| \u{NNNNNNNN} | Some rare characters are encoded with two unicode symbols, taking up to 4 bytes. The long unicode requires braces around. |

Examples with unicode:

alert( "\u00A9" ); // ©

alert( "\u{20331}" ); // 𠌱, a rare chinese hieroglyph (long unicode)

alert( "\u{1F60D}"); // a smiling face sumbol (another long unicode)

All special characters start with a backslash character \. It is also called an “escaping character”.

We should also use it if we want to insert the quote into the string.

For instance:

alert( 'I\'m the Walrus!' ); // I'm the Walrus!

See, we have to prepend the inner quote by the backslash \', because otherwise it would mean the string end.

Of course, that refers only for the quotes that are same as the enclosing ones. So, as a more elegant solution, we could switch to double quotes or backticks instead:

alert( `I'm the Walrus!` ); // I'm the Walrus!

Note that the backslash \ serves for the correct reading of the string by JavaScript, then disappears. The in-memory string has no \. You can clearly see that in alert from the examples above.

But what if we need exactly a backslash \ in the string?

That’s possible, but we need to double it like \\:

alert( `The backslash: \\` ); // The backslash: \

**[String length](https://javascript.info/string" \l "string-length)**

The length property has the string length:

alert( `My\n`.length ); // 3

Note that \n is a single “special” character, so the length is indeed 3.

**length is a property**

People with background in some other languages sometimes mistype by calling str.length() instead of just str.length. That doesn’t work.

Please note that str.length is a numeric property, not a function. There is no need to add brackets after it.

**[Accessing characters](https://javascript.info/string" \l "accessing-characters)**

To get a character at position pos, use square brackets [pos] or call the method [str.charAt(pos)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). The first character starts from the zero position:

let str = `Hello`;

// the first character

alert( str[0] ); // H

alert( str.charAt(0) ); // H

// the last character

alert( str[str.length - 1] ); // o

The square brackets is a modern way of getting a character, while charAt exists mostly for historical reasons.

The only difference between them is that if no character found, [] returns undefined, and charAt returns an empty string:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // '' (an empty string)

**[Strings are immutable](https://javascript.info/string" \l "strings-are-immutable)**

Strings can’t be changed in JavaScript. It is impossible to change a character.

Let’s try to see that it doesn’t work:

let str = 'Hi';

str[0] = 'h'; // error

alert( str[0] ); // doesn't work

The usual workaround is to create a whole new string and assign it to str instead of the old one.

For instance:

let str = 'Hi';

str = 'h' + str[1]; // replace the string

alert( str ); // hi

In the following sections we’ll see more examples of that.

**[Changing the case](https://javascript.info/string" \l "changing-the-case)**

Methods [toLowerCase()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) and [toUpperCase()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) change the case:

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

Or, if we want a single character lowercased:

alert( 'Interface'[0].toLowerCase() ); // 'i'

[**Searching for a substring**](https://javascript.info/string#searching-for-a-substring)

There are multiple ways to look for a substring in a string.

**[str.indexOf](https://javascript.info/string" \l "str-indexof)**

The first method is [str.indexOf(substr, pos)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf).

It looks for the substr in str, starting from the given position pos, and returns the position where the match was found or -1 if nothing can be found.

For instance:

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning

alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)

The optional second parameter allows to search starting from the given position.

For instance, the first occurence of "id" is at the position 1. To look for the next occurence, let’s start the search from the position 2:

let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12

If we’re interested in all occurences, we can run indexOf in a loop. Every new call is made with the position after the previous match:

let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // let's look for it

let pos = 0;

while (true) {

let foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( `Found at ${foundPos}` );

pos = foundPos + 1; // continue the search from the next position

}

The same algorithm can be layed out shorter:

let str = "As sly as a fox, as strong as an ox";

let target = "as";

let pos = -1;

while ((pos = str.indexOf(target, pos + 1)) != -1) {

alert( pos );

}

**str.lastIndexOf(pos)**

There is also a similar method [str.lastIndexOf(pos)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf) that searches from the end of the string to its beginning.

It would list the occurences in the reverse way.

There is a slight inconvenience with indexOf in the if test. We can’t put it in the if like this:

let str = "Widget with id";

if (str.indexOf("Widget")) {

alert("We found it"); // doesn't work!

}

The alert in the example above doesn’t show, because str.indexOf("Widget") returns 0 (meaning that it found the match at the starting position). Right, but if considers that to be false.

So, we should actualy check for -1, like that:

let str = "Widget with id";

if (str.indexOf("Widget") != -1) {

alert("We found it"); // works now!

}

**[includes, startsWith, endsWith](https://javascript.info/string" \l "includes-startswith-endswith)**

The more modern method [str.includes(substr, pos)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes) returns true/false depending on whether str has substr as its part.

It’s the right choice if we need to test for the match, but don’t need its position:

alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false

The optional second argument of str.includes is the position to start searching from:

alert( "Midget".includes("id") ); // true

alert( "Midget".includes("id", 3) ); // false, from position 3 there is no "id"

The methods [str.startsWith](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith) and [str.endsWith](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith) do exactly what they say:

alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with "Wid"

alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"

**[Getting a substring](https://javascript.info/string" \l "getting-a-substring)**

There are 3 methods in JavaScript to get a substring: substring, substr and slice.

**str.slice(start [, end])**

Returns the part of the string from start to (but not including) end.

For instance:

let str = "stringify";

alert( str.slice(0,5) ); // 'strin', the substring from 0 to 5 (not including 5)

alert( str.slice(0,1) ); // 's', from 0 to 1, but not including 1, so only character at 0

If there is no second argument, then slice goes till the end of the string:

let str = "stringify";

alert( str.slice(2) ); // ringify, from the 2nd position till the end

Negative values for start/end are also possible. They mean the position is counted from the string end:

let str = "stringify";

// start at the 4th position from the right, end at the 1st from the right

alert( str.slice(-4, -1) ); // gif

**str.substring(start [, end])**

Returns the part of the string *between* start and end.

Almost the same as slice, but allows start to be greater than end.

For instance:

let str = "stringify";

// these are same for substring

alert( str.substring(2, 6) ); // "ring"

alert( str.substring(6, 2) ); // "ring"

// ...but not for slice:

alert( str.slice(2, 6) ); // "ring" (the same)

alert( str.slice(6, 2) ); // "" (an empty string)

Negative arguments are (unlike slice) not supported, they are treated as 0.

**str.substr(start [, length])**

Returns the part of the string from start, with the given length.

In contrast with the previous methods, this one allows to specify the length instead of the ending position:

let str = "stringify";

alert( str.substr(2, 4) ); // ring, from the 2nd position get 4 characters

The first argument may be negative, to count from the end:

let str = "stringify";

alert( str.substr(-4, 2) ); // gi, from the 4th position get 2 characters

Let’s recap the methods to avoid any confusion:

| **Method** | **selects…** | **negatives** |
| --- | --- | --- |
| slice(start, end) | from start to end | allows negatives |
| substring(start, end) | between start and end | negative values mean 0 |
| substr(start, length) | from start get length characters | allows negative start |

**Which one to choose?**

All of them can do the job. Formally, substr has a minor drawback: it is described not in the core JavaScript specification, but in Annex B, which covers browser-only features that exist mainly for historical reasons. So, non-browser environments may fail to support it. But in practice it works everywhere.

The author finds himself using slice almost all the time.

[**Comparing strings**](https://javascript.info/string#comparing-strings)

Strings are compared character-by-character, in the alphabet order.

Although, there are some oddities.

1. A lowercase letter is always greater than the uppercase:

alert( 'a' > 'Z' ); // true

1. Letters with diacritical marks are “out of order”:

alert( 'Österreich' > 'Zealand' ); // true

That may lead to strange results if we sort country names. Usually people would await for Zealand to be after Österreich in the list.

To understand what happens, let’s review the internal representation of strings in JavaScript.

All strings are encoded using [UTF-16](https://en.wikipedia.org/wiki/UTF-16). That is: each character has a corresponding numeric code. There are special methods that allow to get the character for the code and back.

str.charCodeAt(pos); Returns the code for the character at position pos:

// different case letters have different codes

alert( "z". charCodeAt (0) ); // 122

alert( "Z". charCodeAt (0) ); // 90

String.fromCharCode(**code)**

Creates a character by its numeric code

alert( String. String.fromCharCode(90); // Z

We can also add unicode characters by their codes using \u followed by the hex code:

// 90 is 5a in hexadecimal system

alert( '\u005a' ); // Z

Now let’s see the characters with codes 65..220 (the latin alphabet and a little bit extra) by making a string of them:

let str = '';

for (let i = 65; i <= 220; i++) {

str += String. charCodeAt (i);

}

alert( str );

// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~‚ƒ„

// ¡¢£¤¥¦§¨©ª«¬­®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ

See? Capital character go first, then few special ones, then lowercase characters.

Now it becomes obvious why a > Z.

The characters are compared by their numeric code. The greater code means that the character is greater. The code for a (97) is greater than the code for Z (90).

* All lowercase letters go after uppercase letters, their codes are greater.
* Some letters like Ö stand apart from the main alphabet. Here, it’s code is greater than anything from a to z.

**[Summary](https://javascript.info/string" \l "summary)**

* There are 3 types of quotes. Backticks allow a string to span multiple lines and embed expressions.
* Strings in JavaScript are encoded using UTF-16.
* We can use special characters like \n and insert letters by their unicode using \u....
* To get a character: use [].
* To get a substring: use slice or substring.
* To lowercase/uppercase a string: use toLowerCase/toUpperCase.
* To look for a substring: use indexOf, or includes/startsWith/endsWith for simple checks.
* To compare strings according to the language, use localeCompare, otherwise they are compared by character codes.

There are several other helpful methods in strings:

* str.trim() – removes (“trims”) spaces from the beginning and end of the string.
* str.repeat(n) – repeats the string n times.
* …and others, see the [internet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) for details.

Also strings have methods for doing search/replace with regular expressions. But that topic deserves a separate chapter, so we’ll return to that later.

**Questions and Exercises**

* 1. Write a function ucFirst(str) that returns the string str with the uppercased first character, for instance:

ucFirst("john") == "John";

* 1. Write a function checkSpam(str) that returns true if str contains ‘buy or ‘sell’, otherwise `false.

The function must be case-insensitive:

checkSpam('buy a car now') == true

checkSpam('free sell) == true

checkSpam("innocent rabbit") == false

* 1. Create a function truncate(str, maxlength) that checks the length of the str and, if it exceeds maxlength – replaces the end of str with the ellipsis character "…", to make its length equal to maxlength.

The result of the function should be the truncated (if needed) string.

For instance:

truncate("What I'd like to tell on this topic is:", 20) = "What I'd like to te…"

truncate("Hi everyone!", 20) = "Hi everyone!"

* 1. We have a cost in the form "$120". That is: the dollar sign goes first, and then the number.

Create a function extractCurrencyValue(str) that would extract the numeric value from such string and return it.

The example:

alert( extractCurrencyValue('$120') === 120 ); // true

**More Problems...**

1. Write a program to solve each of the following problems:

a) input a word. Isolate the first letter and output it as a capital.

$("#btnQ1").click(function(event){

let data=$("#txtQ1").val();

let capital=data.charAt(0).toUpperCase();

$("#lblQ1").text(capital);

});//end of btnQ1 click

b) input a sentence. Count the number of spaces and determine the number of words in the sentence.

$("#btnQ1").click(function(event){

let sentence=$("#txtQ1").val();

let wordCount=1;

for(x=0;x<sentence.length;x++)

{

alert(x+sentence.charAt(x));

{

wordCount++;

}

}

$("#lblQ1").text(wordCount);

});//end of btnQ1 click

c) input a sentence. Convert the sentence to Uppercase letters and output.

$("#btnQ1").click(function(event){

let sentence=$("#txtQ1").val();

sentence=sentence.toUpperCase();

$("#lblQ1").text(sentence);

});//end of btnQ1 click

d) input a word. Create a new word containing the letters of the original word in the reverse order. For example, “bread” will become “daerb”.

$("#btnQ1").click(function(event){

let word=$("#txtQ1").val();

let newWord="";

for(x=word.length-1;x>=0;x--)

{

newWord+=word.charAt(x);

}

$("#lblQ1").text(newWord);

});//end of btnQ1 click

2. Write a program that prompts the user to enter a word using only uppercase letters. The program is to display the location of the first\*vowel in the word and the vowel itself. If the word does not contain a vowel, then an appropriate message should be output to the screen. For example,

Sample Input: VISUAL

Sample Output: The first vowel is “I” and it is in position 2.

3. Write a program which prompts the user to enter a word and a search pattern. The program is to display the word along with the beginning position of the search pattern in the word. Give an appropriate response if the pattern does not exist in the word.

Sample Input: Enter a word: KALAMAZOO

Enter a search pattern: AZ

Sample Output: The pattern “AZ” starts at position 6.

4. Write a program that accepts a sentence from the user. The output is to be the frequency of each of the vowels used in the sentence. This should not be case sensitive.

Sample Input: Enter a sentence: I love Visual Basic

Sample Output: VOWEL FREQUENCY

A 2

E 1

I 3

O 1

U 1

5. Write a program to check the spelling rule “i before e except after c” for a word entered by the user. If “cie” occurs, rebuild the word with the proper spelling of “cei”. You may want to look at other special occurrences: for example “height” is not i before e even though there is no c.

6. Enhance #3 so that your program counts the number of occurrences of a the pattern.

Sample Input: Enter a word: abracadabra

Enter a pattern: ab

Sample Output The pattern “ab” occurs 2 times.

7. Write a program that accepts a sentence from the user in lowercase letters only. The program will search for any of these letters m n s t w and replace these with the number corresponding to the position of the letter in the string “mnstw”. For example, the sentence “visual basic is the best” would be coded “vi3ual ba3ic i3 4he be34” because the s is the 3rd letter in the string “mnstw” and t is the 4th letter. The program will display the original sentence as well as the coded sentence.

# Problem Set: Code / Decode

**Task:** Write a program that will accept as input a phrase, a code offset and a selection of encode or decode. The program will then encode or decode the text using the offset as a means of determining the letter substitution.

**Example:** If the offset is 3 then letter “A” becomes “D”, 3 letters down the alphabet. Letter “Y” would become “B” as the alphabet would wrap around. The word “THE” would become the word “WKH”. Your program must be able to handle a phrase, not just a word and should ignore any non letter characters.