**Строки**

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк — всегда [UTF-16](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.

**[Кавычки](https://learn.javascript.ru/string" \l "kavychki)**

В JavaScript есть разные типы кавычек.

Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в ${…}:

function sum(a, b) {

return a + b;

}

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

Ещё одно преимущество обратных кавычек — они могут занимать более одной строки, вот так:

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList); // список гостей, состоящий из нескольких строк

Выглядит вполне естественно, не правда ли? Что тут такого? Но если попытаться использовать точно так же одинарные или двойные кавычки, то будет ошибка:

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

\* John";

Одинарные и двойные кавычки в языке с незапамятных времён: тогда потребность в многострочных строках не учитывалась. Что касается обратных кавычек, они появились существенно позже, и поэтому они гибче.

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#Tagged_templates). Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

**[Спецсимволы](https://learn.javascript.ru/string" \l "spetssimvoly)**

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

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

alert(guestList); // список гостей, состоящий из нескольких строк

В частности, эти две строки эквивалентны, просто записаны по-разному:

// перевод строки добавлен с помощью символа перевода строки

let str1 = "Hello\nWorld";

// многострочная строка, созданная с использованием обратных кавычек

let str2 = `Hello

World`;

alert(str1 == str2); // true

Есть и другие, реже используемые спецсимволы. Вот список:

| **Символ** | **Описание** |
| --- | --- |
| \n | Перевод строки |
| \r | Возврат каретки: самостоятельно не используется. В текстовых файлах Windows для перевода строки используется комбинация символов \r\n. |
| \', \" | Кавычки |
| \\ | Обратный слеш |
| \t | Знак табуляции |
| \b, \f, \v | Backspace, Form Feed и Vertical Tab — оставлены для обратной совместимости, сейчас не используются. |
| \xXX | Символ с шестнадцатеричным юникодным кодом XX, например, '\x7A' — то же самое, что 'z'. |
| \uXXXX | Символ в кодировке UTF-16 с шестнадцатеричным кодом XXXX, например, \u00A9 — юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр. |
| \u{X…XXXXXX} (от 1 до 6 шестнадцатеричных цифр) | Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом. |

Примеры с Юникодом:

// ©

alert( "\u00A9" );

// Длинные юникодные коды

// 佫, редкий китайский иероглиф

alert( "\u{20331}" );

// 😍, лицо с улыбкой и глазами в форме сердец

alert( "\u{1F60D}" );

Все спецсимволы начинаются с обратного слеша, \ — так называемого «символа экранирования».

Он также используется, если необходимо вставить в строку кавычку.

К примеру:

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

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш — \' — иначе она бы обозначала окончание строки.

Разумеется, требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Так что мы можем применить и более элегантное решение, использовав для этой строки двойные или обратные кавычки:

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

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Вы можете явно видеть это в выводах alert в примерах выше.

Но что, если нам надо добавить в строку собственно сам обратный слеш \?

Это можно сделать, добавив перед ним… ещё один обратный слеш!

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

**[Длина строки](https://learn.javascript.ru/string" \l "dlina-stroki)**

Свойство length содержит длину строки:

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

Обратите внимание, \n — это один спецсимвол, поэтому тут всё правильно: длина строки 3.

length — это свойство

Бывает так, что люди с практикой в других языках случайно пытаются вызвать его, добавляя круглые скобки: они пишут str.length() вместо str.length. Это не работает.

Так как str.length — это числовое свойство, а не функция, добавлять скобки не нужно.

**[Доступ к символам](https://learn.javascript.ru/string" \l "dostup-k-simvolam)**

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: [str.charAt(pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). Первый символ занимает нулевую позицию:

let str = `Hello`;

// получаем первый символ

alert( str[0] ); // H

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

// получаем последний символ

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

Квадратные скобки — современный способ получить символ, в то время как charAt существует в основном по историческим причинам.

Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt — пустую строку:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // '' (пустая строка)

Также можно перебрать строку посимвольно, используя for..of:

for (let char of "Hello") {

alert(char); // H,e,l,l,o (char — сначала "H", потом "e", потом "l" и т. д.)

}

**[Строки неизменяемы](https://learn.javascript.ru/string" \l "stroki-neizmenyaemy)**

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана — она такая навсегда.

Давайте попробуем так сделать, и убедимся, что это не работает:

let str = 'Hi';

str[0] = 'h'; // ошибка

alert( str[0] ); // не работает

Можно создать новую строку и записать её в ту же самую переменную вместо старой.

Например:

let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi

В последующих разделах мы увидим больше примеров.

**[Изменение регистра](https://learn.javascript.ru/string" \l "izmenenie-registra)**

Методы [toLowerCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) и [toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) меняют регистр символов:

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

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

Если мы захотим перевести в нижний регистр какой-то конкретный символ:

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

**[Поиск подстроки](https://learn.javascript.ru/string" \l "poisk-podstroki)**

Существует несколько способов поиска подстроки.

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

Первый метод — [str.indexOf(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf).

Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений.

Например:

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, потому что подстрока 'Widget' найдена в начале

alert( str.indexOf('widget') ); // -1, совпадений нет, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1, подстрока "id" найдена на позиции 1 (..idget with id)

Необязательный второй аргумент позволяет начать поиск с определённой позиции.

Например, первое вхождение "id" — на позиции 1. Для того, чтобы найти следующее, начнём поиск с позиции 2:

let str = 'Widget with id';

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

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;

while (true) {

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

if (foundPos == -1) break;

alert( `Найдено тут: ${foundPos}` );

pos = foundPos + 1; // продолжаем со следующей позиции

}

Тот же алгоритм можно записать и короче:

let str = "Ослик Иа-Иа посмотрел на виадук";

let target = "Иа";

let pos = -1;

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

alert( pos );

}

str.lastIndexOf(substr, position)

Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу.

Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

При проверке indexOf в условии if есть небольшое неудобство. Такое условие не будет работать:

let str = "Widget with id";

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

alert("Совпадение есть"); // не работает

}

Мы ищем подстроку "Widget", и она здесь есть, прямо на позиции 0. Но alert не показывается, т. к. str.indexOf("Widget") возвращает 0, и if решает, что тест не пройден.

Поэтому надо делать проверку на -1:

let str = "Widget with id";

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

alert("Совпадение есть"); // теперь работает

}

**[Трюк с побитовым НЕ](https://learn.javascript.ru/string" \l "tryuk-s-pobitovym-ne)**

Существует старый трюк с использованием [побитового оператора НЕ](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#.7E_.28Bitwise_NOT.29) — ~. Он преобразует число в 32-разрядное целое со знаком (signed 32-bit integer). Дробная часть, в случае, если она присутствует, отбрасывается. Затем все биты числа инвертируются.

На практике это означает простую вещь: для 32-разрядных целых чисел значение ~n равно -(n+1).

В частности:

alert( ~2 ); // -3, то же, что -(2+1)

alert( ~1 ); // -2, то же, что -(1+1)

alert( ~0 ); // -1, то же, что -(0+1)

alert( ~-1 ); // 0, то же, что -(-1+1)

Таким образом, ~n равняется 0 только при n == -1 (для любого n, входящего в 32-разрядные целые числа со знаком).

Соответственно, прохождение проверки if ( ~str.indexOf("…") ) означает, что результат indexOf отличен от -1, совпадение есть.

Это иногда применяют, чтобы сделать проверку indexOf компактнее:

let str = "Widget";

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

alert( 'Совпадение есть' ); // работает

}

Обычно использовать возможности языка каким-либо неочевидным образом не рекомендуется, но этот трюк широко используется в старом коде, поэтому его важно понимать.

Просто запомните: if (~str.indexOf(…)) означает «если найдено».

Впрочем, если быть точнее, из-за того, что большие числа обрезаются до 32 битов оператором ~, существуют другие числа, для которых результат тоже будет 0, самое маленькое из которых — ~4294967295=0. Поэтому такая проверка будет правильно работать только для строк меньшей длины.

На данный момент такой трюк можно встретить только в старом коде, потому что в новом он просто не нужен: есть метод .includes (см. ниже).

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

Более современный метод [str.includes(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/includes) возвращает true, если в строке str есть подстрока substr, либо false, если нет.

Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна:

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

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

Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:

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

alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3

Методы [str.startsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith) и [str.endsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith) проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

alert( "Widget".startsWith("Wid") ); // true, "Wid" — начало "Widget"

alert( "Widget".endsWith("get") ); // true, "get" — окончание "Widget"

**[Получение подстроки](https://learn.javascript.ru/string" \l "poluchenie-podstroki)**

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

str.slice(start [, end])

Возвращает часть строки от start до (не включая) end.

Например:

let str = "stringify";

// 'strin', символы от 0 до 5 (не включая 5)

alert( str.slice(0, 5) );

// 's', от 0 до 1, не включая 1, т. е. только один символ на позиции 0

alert( str.slice(0, 1) );

Если аргумент end отсутствует, slice возвращает символы до конца строки:

let str = "stringify";

alert( str.slice(2) ); // ringify, с позиции 2 и до конца

Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов *с конца строки*:

let str = "stringify";

// начинаем с позиции 4 справа, а заканчиваем на позиции 1 справа

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

str.substring(start [, end])

Возвращает часть строки *между* start и end.

Это — почти то же, что и slice, но можно задавать start больше end.

Например:

let str = "stringify";

// для substring эти два примера — одинаковы

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

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

// …но не для slice:

alert( str.slice(2, 6) ); // "ring" (то же самое)

alert( str.slice(6, 2) ); // "" (пустая строка)

Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

str.substr(start [, length])

Возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

let str = "stringify";

// ring, получаем 4 символа, начиная с позиции 2

alert( str.substr(2, 4) );

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

let str = "stringify";

// gi, получаем 2 символа, начиная с позиции 4 с конца строки

alert( str.substr(-4, 2) );

Давайте подытожим, как работают эти методы, чтобы не запутаться:

| **метод** | **выбирает…** | **отрицательные значения** |
| --- | --- | --- |
| slice(start, end) | от start до end (не включая end) | можно передавать отрицательные значения |
| substring(start, end) | между start и end | отрицательные значения равнозначны 0 |
| substr(start, length) | length символов, начиная от start | значение start может быть отрицательным |

Какой метод выбрать?

Все эти методы эффективно выполняют задачу. Формально у метода substr есть небольшой недостаток: он описан не в собственно спецификации JavaScript, а в приложении к ней — Annex B. Это приложение описывает возможности языка для использования в браузерах, существующие в основном по историческим причинам. Таким образом, в другом окружении, отличном от браузера, он может не поддерживаться. Однако на практике он работает везде.

Из двух других вариантов, slice более гибок, он поддерживает отрицательные аргументы, и его короче писать. Так что, в принципе, можно запомнить только его.

**[Сравнение строк](https://learn.javascript.ru/string" \l "sravnenie-strok)**

Как мы знаем из главы [Операторы сравнения](https://learn.javascript.ru/comparison), строки сравниваются посимвольно в алфавитном порядке.

Тем не менее, есть некоторые нюансы.

1. Строчные буквы больше заглавных:

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

Буквы, имеющие диакритические знаки, идут «не по порядку»:

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

1. Это может привести к своеобразным результатам при сортировке названий стран: нормально было бы ожидать, что Zealand будет после Österreich в списке.

Чтобы разобраться, что происходит, давайте ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в [UTF-16](https://ru.wikipedia.org/wiki/UTF-16). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

str.codePointAt(pos)

Возвращает код для символа, находящегося на позиции pos:

// одна и та же буква в нижнем и верхнем регистре

// будет иметь разные коды

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

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

String.fromCodePoint(code)

Создаёт символ по его коду code

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

Также можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа:

// 90 — 5a в шестнадцатеричной системе счисления

alert( '\u005a' ); // Z

Давайте сделаем строку, содержащую символы с кодами от 65 до 220 — это латиница и ещё некоторые распространённые символы:

let str = '';

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

str += String.fromCodePoint(i);

}

alert( str );

// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

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

Как видите, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и Ö ближе к концу вывода.

Теперь очевидно, почему a > Z.

Символы сравниваются по их кодам. Больший код — больший символ. Код a (97) больше кода Z (90).

* Все строчные буквы идут после заглавных, так как их коды больше.
* Некоторые буквы, такие как Ö, вообще находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.

**[Правильное сравнение](https://learn.javascript.ru/string" \l "pravilnoe-sravnenie)**

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты.

Поэтому браузеру нужно знать, какой язык использовать для сравнения.

К счастью, все современные браузеры (для IE10− нужна дополнительная библиотека [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), обеспечивающий правильное сравнение строк на разных языках с учётом их правил.

Для этого есть соответствующий метод.

Вызов [str.localeCompare(str2)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

* Отрицательное число, если str меньше str2.
* Положительное число, если str больше str2.
* 0, если строки равны.

Например:

alert( 'Österreich'.localeCompare('Zealand') ); // -1

У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

**[Как всё устроено, Юникод](https://learn.javascript.ru/string" \l "kak-vsyo-ustroeno-yunikod)**

Глубокое погружение в тему

Этот раздел более подробно описывает, как устроены строки. Такие знания пригодятся, если вы намерены работать с эмодзи, редкими математическими символами, иероглифами, либо с ещё какими-то редкими символами.

Если вы не планируете их поддерживать, эту секцию можно пропустить.

**[Суррогатные пары](https://learn.javascript.ru/string" \l "surrogatnye-pary)**

Многие символы возможно записать одним 16-битным словом: это и буквы большинства европейских языков, и числа, и даже многие иероглифы.

Но 16 битов — это 65536 комбинаций, так что на все символы этого, разумеется, не хватит. Поэтому редкие символы записываются двумя 16-битными словами — это также называется «суррогатная пара».

Длина таких строк — 2:

alert( '𝒳'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL X

alert( '😂'.length ); // 2, FACE WITH TEARS OF JOY

alert( '𩷶'.length ); // 2, редкий китайский иероглиф

Обратите внимание, суррогатные пары не существовали, когда был создан JavaScript, поэтому язык не обрабатывает их адекватно!

Ведь в каждой из этих строк только один символ, а length показывает длину 2.

String.fromCodePoint и str.codePointAt — два редких метода, правильно работающие с суррогатными парами, но они и появились в языке недавно. До них были только [String.fromCharCode](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/fromCharCode) и [str.charCodeAt](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charCodeAt). Эти методы, вообще, делают то же самое, что fromCodePoint/codePointAt, но не работают с суррогатными парами.

Получить символ, представленный суррогатной парой, может быть не так просто, потому что суррогатная пара интерпретируется как два символа:

alert( '𝒳'[0] ); // странные символы…

alert( '𝒳'[1] ); // …части суррогатной пары

Части суррогатной пары не имеют смысла сами по себе, так что вызовы alert в этом примере покажут лишь мусор.

Технически, суррогатные пары возможно обнаружить по их кодам: если код символа находится в диапазоне 0xd800..0xdbff, то это — первая часть суррогатной пары. Следующий символ — вторая часть — имеет код в диапазоне 0xdc00..0xdfff. Эти два диапазона выделены исключительно для суррогатных пар по стандарту.

В данном случае:

// charCodeAt не поддерживает суррогатные пары, поэтому возвращает код для их частей

alert( '𝒳'.charCodeAt(0).toString(16) ); // d835, между 0xd800 и 0xdbff

alert( '𝒳'.charCodeAt(1).toString(16) ); // dcb3, между 0xdc00 и 0xdfff

Дальше в главе [Перебираемые объекты](https://learn.javascript.ru/iterable) будут ещё способы работы с суррогатными парами. Для этого есть и специальные библиотеки, но нет достаточно широко известной, чтобы предложить её здесь.

**[Диакритические знаки и нормализация](https://learn.javascript.ru/string" \l "diakriticheskie-znaki-i-normalizatsiya)**

Во многих языках есть символы, состоящие из некоторого основного символа со знаком сверху или снизу.

Например, буква a — это основа для àáâäãåā. Наиболее используемые составные символы имеют свой собственный код в таблице UTF-16. Но не все, в силу большого количества комбинаций.

Чтобы поддерживать любые комбинации, UTF-16 позволяет использовать несколько юникодных символов: основной и дальше один или несколько особых символов-знаков.

Например, если после S добавить специальный символ «точка сверху» (код \u0307), отобразится Ṡ.

alert( 'S\u0307' ); // Ṡ

Если надо добавить сверху (или снизу) ещё один знак — без проблем, просто добавляем соответствующий символ.

Например, если добавить символ «точка снизу» (код \u0323), отобразится S с точками сверху и снизу: Ṩ.

Добавляем два символа:

alert( 'S\u0307\u0323' ); // Ṩ

Это даёт большую гибкость, но из-за того, что порядок дополнительных символов может быть различным, мы получаем проблему сравнения символов: можно представить по-разному символы, которые ничем визуально не отличаются.

Например:

let s1 = 'S\u0307\u0323'; // Ṩ, S + точка сверху + точка снизу

let s2 = 'S\u0323\u0307'; // Ṩ, S + точка снизу + точка сверху

alert( `s1: ${s1}, s2: ${s2}` );

alert( s1 == s2 ); // false, хотя на вид символы одинаковы (?!)

Для решения этой проблемы есть алгоритм «юникодной нормализации», приводящий каждую строку к единому «нормальному» виду.

Его реализует метод [str.normalize()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/normalize).

alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() ); // true

Забавно, но в нашем случае normalize() «схлопывает» последовательность из трёх символов в один: \u1e68 — S с двумя точками.

alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true

Разумеется, так происходит не всегда. Просто Ṩ — это достаточно часто используемый символ, поэтому создатели UTF-16 включили его в основную таблицу и присвоили ему код.

Подробнее о правилах нормализации и составлении символов можно прочитать в дополнении к стандарту Юникод: [Unicode Normalization Forms](http://www.unicode.org/reports/tr15/). Для большинства практических целей информации из этого раздела достаточно.

**[Итого](https://learn.javascript.ru/string" \l "itogo)**

* Есть три типа кавычек. Строки, использующие обратные кавычки, могут занимать более одной строки в коде и включать выражения ${…}.
* Строки в JavaScript кодируются в UTF-16.
* Есть специальные символы, такие как \n, и можно добавить символ по его юникодному коду, используя \u….
* Для получения символа используйте [].
* Для получения подстроки используйте slice или substring.
* Для того, чтобы перевести строку в нижний или верхний регистр, используйте toLowerCase/toUpperCase.
* Для поиска подстроки используйте indexOf или includes/startsWith/endsWith, когда надо только проверить, есть ли вхождение.
* Чтобы сравнить строки с учётом правил языка, используйте localeCompare.

Строки также имеют ещё кое-какие полезные методы:

* str.trim() — убирает пробелы в начале и конце строки.
* str.repeat(n) — повторяет строку n раз.
* …и другие, которые вы можете найти в [справочнике](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String).