

Раздел

[Типы данных](#)

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого


Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

[🏠 → Язык программирования JavaScript](#)
[→ Типы данных](#) 9-го сентября 2019

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**, вне зависимости от кодировки страницы.

Кавычки

В JavaScript есть разные типы кавычек.

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

```
1 let single = 'single-quoted';
2 let double = "double-quoted";
3
4 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 `${...}`:

```
1 function sum(a, b) {
2   return a + b;
3 }
4
5 alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

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

```
1 let guestList = `Guests:
2   * John
3   * Pete
4   * Mary
5 `;
6
7 alert(guestList); // список гостей, состоящий из нескол
```

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

```
1 let guestList = "Guests: // Error: Unexpected token ILL
2   * 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](#). 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

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

```
1 let guestList = "Guests:\n * John\n * Pete\n * Mary";
2
3 alert(guestList); // список гостей, состоящий из нескол
```

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

```
1 // перевод строки добавлен с помощью символа перевода
2 let str1 = "Hello\nWorld";
3
4 // многострочная строка, созданная с использованием обр
5 let str2 = `Hello
6 World`;
7
8 alert(str1 == str2); // true
```

Есть и другие, реже используемые спецсимволы. Вот список:

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

Примеры с Юникодом:

```
1 // ©
2 alert( "\u00A9" );
3
4 // Длинные юникодные коды
5 // 佬, редкий китайский иероглиф
6 alert( "\u{20331}" );
7 // 😊, лицо с улыбкой и глазами в форме сердец
8 alert( "\u{1F60D}" );
```

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

Он также используется, если необходимо вставить в строку кавычку.

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

К примеру:

```
1 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:

```
1 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!

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

Длина строки

Свойство `length` содержит длину строки:

```
1 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

Получить символ, который занимает позицию `pos`, можно с помощью квадратных скобок: `[pos]`. Также можно использовать метод `charAt`: `str.charAt(pos)`. Первый символ занимает нулевую позицию:

```
1 let str = `Hello`;
2
3 // получаем первый символ
4 alert( str[0] ); // H
5 alert( str.charAt(0) ); // H
6
7 // получаем последний символ
8 alert( str[str.length - 1] ); // o
```

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

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



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

```
1 let str = `Hello`;
2
3 alert( str[1000] ); // undefined
4 alert( str.charAt(1000) ); // '' (пустая строка)
```

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

```
1 for (let char of "Hello") {
2   alert(char); // H,e,l,l,o (char — сначала "H", потом
3 }
```

Строки неизменяемы

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:

```
1 let str = 'Hi';
2
3 str[0] = 'h'; // ошибка
4 alert( str[0] ); // не работает
```

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

Например:

```
1 let str = 'Hi';
2
3 str = 'h' + str[1]; // заменяем строку
4
5 alert( str ); // hi
```

В последующих разделах мы увидим больше примеров.

Изменение регистра

Методы `toLowerCase()` и `toUpperCase()` меняют регистр символов:

```
1 alert( 'Interface'.toUpperCase() ); // INTERFACE
2 alert( 'Interface'.toLowerCase() ); // interface
```

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

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

Поиск подстроки

Существует несколько способов поиска подстроки.

`str.indexOf`

Первый метод — `str.indexOf(substr, pos)`.

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

Например:

```
1 let str = 'Widget with id';
2
3 alert( str.indexOf('Widget') ); // 0, потому что подстрока 'Widget' найдена
4 alert( str.indexOf('widget') ); // -1, совпадений нет,
5
6 alert( str.indexOf("id") ); // 1, подстрока "id" найдена
```

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:

```
1 let str = 'Widget with id';
2
3 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:

```
1 let str = 'Ослик Иа-Иа посмотрел на виадук';
2
3 let target = 'Иа'; // цель поиска
4
5 let pos = 0;
6 while (true) {
7     let foundPos = str.indexOf(target, pos);
8     if (foundPos == -1) break;
9
10    alert( `Найдено тут: ${foundPos}` );
11    pos = foundPos + 1; // продолжаем со следующей позиции
12 }
```

Тот же алгоритм можно записать и короче:

```
1 let str = "Ослик Иа-Иа посмотрел на виадук";
2 let target = "Иа";
3
4 let pos = -1;
5 while ((pos = str.indexOf(target, pos + 1)) != -1) {
6     alert( pos );
7 }
```

i `str.lastIndexOf(substr, position)`

Также есть похожий метод `str.lastIndexOf(substr, position)`, 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:

```
1 let str = "Widget with id";
2
3 if (str.indexOf("Widget")) {
4     alert("Совпадение есть"); // не работает
5 }
```

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



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:

```
1 let str = "Widget with id";
2
3 if (str.indexOf("Widget") !== -1) {
4     alert("Совпадение есть"); // теперь работает
5 }
```

Трюк с побитовым НЕ

Существует старый трюк с использованием побитового оператора НЕ — `~`. 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:

```
1 alert( ~2 ); // -3, то же, что -(2+1)
2 alert( ~1 ); // -2, то же, что -(1+1)
3 alert( ~0 ); // -1, то же, что -(0+1)
4 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:

```
1 let str = "Widget";
2
3 if (~str.indexOf("Widget")) {
4     alert( 'Совпадение есть' ); // работает
5 }
```

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

Более современный метод `str.includes(substr, pos)` 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:

```
1 alert( "Widget with id".includes("Widget") ); // true
2
```

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

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

```
1 alert( "Midget".includes("id") ); // true
2 alert( "Midget".includes("id", 3) ); // false, поиск на
```

Методы `str.startsWith` и `str.endsWith` проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

```
1 alert( "Widget".startsWith("Wid") ); // true, "Wid" — н
2 alert( "Widget".endsWith("get") ); // true, "get" — око
```

Получение подстроки

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:

```
1 let str = "stringify";
2 // 'strin', символы от 0 до 5 (не включая 5)
3 alert( str.slice(0, 5) );
4 // 's', от 0 до 1, не включая 1, т. е. только один символ
5 alert( str.slice(0, 1) );
```

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

```
1 let str = "stringify";
2 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:

```
1 let str = "stringify";
2
3 // начинаем с позиции 4 справа, а заканчиваем на позиции
4 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:

```
1 let str = "stringify";
2
3 // для substring эти два примера — одинаковы
4 alert( str.substring(2, 6) ); // "ring"
5 alert( str.substring(6, 2) ); // "ring"
6
7 // ...но не для slice:
8 alert( str.slice(2, 6) ); // "ring" (то же самое)
9 alert( str.slice(6, 2) ); // "" (пустая строка)
```

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



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:

```
1 let str = "stringify";
2 // ring, получаем 4 символа, начиная с позиции 2
3 alert( str.substr(2, 4) );
```

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

```
1 let str = "stringify";
2 // gi, получаем 2 символа, начиная с позиции 4 с конца
3 alert( str.substr(-4, 2) );
```

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

метод	выбирает...	отрицательные значения
<code>slice(start, end)</code>	от <code>start</code> до <code>end</code> (не включая <code>end</code>)	можно передавать отрицательные значения
<code>substring(start, end)</code>	между <code>start</code> и <code>end</code>	отрицательные значения равнозначны <code>0</code>
<code>substr(start, length)</code>	<code>length</code> символов, начиная от <code>start</code>	значение <code>start</code> может быть отрицательным



i 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

Как мы знаем из главы [Операторы сравнения](#), строки сравниваются посимвольно в алфавитном порядке.

Тем не менее, есть некоторые нюансы.

1. Строчные буквы больше заглавных:

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

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

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

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

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



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

Строки кодируются в **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`:

```
1 // одна и та же буква в нижнем и верхнем регистре
2 // будет иметь разные коды
3 alert( "z".codePointAt(0) ); // 122
4 alert( "Z".codePointAt(0) ); // 90
```

`String.fromCharCode(code)`

Создаёт символ по его коду `code`

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

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

```
1 // 90 — 5a в шестнадцатеричной системе счисления
2 alert( '\u005a' ); // Z
```

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

```
1 let str = '';
2
3 for (let i = 65; i <= 220; i++) {
4   str += String.fromCharCode(i);
5 }
6 alert( str );
7 // ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
8 // ;ç£¥!$%&'()*+,-./:;<=>?@A-Z[\]^_`abcdefghijklmnopqrstuvwxyz
```

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. Большой код — большой 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

«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](#)) поддерживают стандарт [ECMA 402](#), обеспечивающий правильное сравнение строк на разных языках с учётом их правил.

Для этого есть соответствующий метод.

Вызов `str.localeCompare(str2)` возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

- Отрицательное число, если `str` меньше `str2`.

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



- Положительное число, если `str` больше `str2`.
- 0, если строки равны.

Например:

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

У этого метода есть два дополнительных аргумента, которые указаны в [документации](#). 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 "а" и "á".

Как всё устроено, Юникод

⚠️ 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

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:

```
1 alert( '𐀀'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL A
2 alert( '😭'.length ); // 2, FACE WITH TEARS OF JOY
3 alert( '𐀀'.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` и `str.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:

```
1 alert( '𐀀'[0] ); // странные символы...
2 alert( '𐀀'[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:

```
1 // charCodeAt не поддерживает суррогатные пары, поэтому
2
3 alert( '𐀀'.charCodeAt(0).toString(16) ); // d835, между
4 alert( '𐀀'.charCodeAt(1).toString(16) ); // dcb3, между
```

Дальше в главе [Перебираемые объекты](#) 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

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

Например, буква *а* — это основа для *ăăăăăăăă*. 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 *Š*.

```
1 alert( 'S\u0307' ); // Š
```

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: *Š*.

Добавляем два символа:

```
1 alert( 'S\u0307\u0323' ); // Š
```

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:

```
1 let s1 = 'S\u0307\u0323'; // Š, S + точка сверху + точк
2 let s2 = 'S\u0323\u0307'; // Š, S + точка снизу + точка
3
4 alert( `s1: ${s1}, s2: ${s2}` );
5
6 alert( s1 == s2 ); // false, хотя на вид символы одинак
```

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

Его реализует метод `str.normalize()`.

```
1 alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".n
```

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

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

Раздел

[Типы данных](#)

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



[Редактировать на GitHub](#)

```
2
3 alert( "S\u0307\u0323".normalize() == "\u1e68" ); // tr
```

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



Разумеется, так происходит не всегда. Просто \$ — 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](#). Для большинства практических целей информации из этого раздела достаточно.

Итого

- 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](#).

Также есть методы для поиска и замены с использованием регулярных выражений. Но это отдельная большая тема, поэтому ей посвящена отдельная глава учебника [Регулярные выражения](#).

✓ Задачи

Сделать первый символ заглавным

важность: 5

Напишите функцию `ucFirst(str)`, возвращающую строку `str` с заглавным первым символом. Например:

```
1 ucFirst("вася") == "Вася";
```

[Открыть песочницу с тестами для задачи.](#)

решение

Проверка на спам

важность: 5

Напишите функцию `checkSpam(str)`, возвращающую `true`, если `str` содержит `'viagra'` или `'XXX'`, а иначе `false`.

Функция должна быть нечувствительна к регистру:

```
1 checkSpam('buy ViAgRA now') == true
2 checkSpam('free xxxxx') == true
3 checkSpam("innocent rabbit") == false
```

Раздел

Типы данных

Навигация по уроку

Кавычки

Спецсимволы

Длина строки

Доступ к символам

Строки неизменяемы

Изменение регистра

Поиск подстроки

Получение подстроки

Сравнение строк

Как всё устроено, Юникод

Итого

Задачи (4)

Комментарии

Поделиться



Редактировать на GitHub



[Открыть песочницу с тестами для задачи.](#)

решение

Усечение строки

важность: 5

Создайте функцию `truncate(str, maxLength)`, которая проверяет длину строки `str` и, если она превосходит `maxLength`, заменяет конец `str` на "...", так, чтобы её длина стала равна `maxLength`.

Результатом функции должна быть та же строка, если усечение не требуется, либо, если необходимо, усечённая строка.

Например:

```
1 truncate("Вот, что мне хотелось бы сказать на эту тему:", 20) === "Вот, что мне хотело"
2
3 truncate("Всем привет!", 20) === "Всем привет!"
```

[Открыть песочницу с тестами для задачи.](#)

решение

Выделить число

важность: 4

Есть стоимость в виде строки "\$120". То есть сначала идёт знак валюты, а затем – число.

Создайте функцию `extractCurrencyValue(str)`, которая будет из такой строки выделять числовое значение и возвращать его.

Например:

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

[Открыть песочницу с тестами для задачи.](#)

решение

Проводим [курсы по JavaScript и фреймворкам.](#)

Комментарии

перед тем как писать...