> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

# Форматирование текста

В этой главе приводится порядок работы со строками и текстом в JavaScript.

## Строки

Строки используются для представления текстовых данных. Каждая строка - это набор "элементов", а каждый элемент - 16 битное беззнаковое целое значение. Элементы имеют определённые позиции. Так первый элемент имеет индекс 0, следующий - 1, и так далее. Длина строки - это количество элементов в ней. Вы можете создать строки, используя строковые литералы или объекты класса String.

### Строковые литералы

Вы можете создавать простые строки, используя либо одинарные, либо двойные кавычки:

```
JS
'foo';
"bar";
```

Начиная со стандарта ES6 (ES-2015) для простых и сложных строк можно использовать обратные косые кавычки, а также, вставлять значения:

```
const name = "Alex";
const str = `Привет, ${name},
   как дела?`;

console.log(str);
// Привет, Alex,
// как дела?
```

Подробнее про использование обратных косых кавычек (` `), читайте ниже.

Строки с более богатым содержимым можно создать с помощью ESC-последовательностей(комбинация символов, обычно используемая для задания неотображаемых символов и символов, имеющих специальное значение):

## Шестнадцатеричные экранированные последовательности

Число после \x трактуется как шестнадцатеричное.

JS

```
"\xA9"; // "©"
```

## Unicode экранированные последовательности

Экранированные последовательности Unicode требуют по меньшей мере 4 символа после `\u`.

JS

```
"\u00A9"; // "©"
```

## Экранирование элементов кода Unicode

Нововведение ECMAScript 6, которое позволяет экранировать каждый Unicode символ, используя шестнадцатеричные значения (вплоть до `0x10FFFF`). С простым экранированием Unicode обычно требуется писать связанные друг с другом части по - отдельности для получения того же результата.

Смотрите также `String.fromCodePoint()` или `String.prototype.codePointAt()`.

```
JS
```

```
"\u{2F804}";

// То же самое с простым Unicode
"\uD87E\uDC04";
```

## Объекты String

Объект `String` - это обёртка вокруг примитивного строкового типа данных.

```
JS
```

```
var s = new String("foo"); // Создание объекта
console.log(s); // Отобразится: { '0': 'f', '1': 'o', '2': 'o'}
typeof s; // Вернёт 'object'
```

Вы можете вызвать любой метод объекта класса `String` на строковом литерале - JavaScript сам преобразует строковый литерал во временный объект `String`, вызовет требуемый метод и затем уничтожит этот временный объект. Со строковыми литералами вы также можете использовать и `String.length` свойство.

Следует использовать строковые литералы до тех пор, пока вам действительно не обойтись без `String` объекта, потому что, порой, объект String может вести себя неожиданно (не так, как строковый литерал). Например:

```
JS
```

```
var s1 = "2 + 2"; // Создание строкового литерала
var s2 = new String("2 + 2"); // Создание String объекта
eval(s1); // Вернёт 4
eval(s2); // Вернёт строку "2 + 2"
```

Объект `String` имеет свойство `length`, которое обозначает количество символов в строке. Например, в следующем коде x получит значение 13 потому, что "Hello, World!" содержит 13 символов, каждый из которых представлен одним кодом UTF-16. Вы можете обратиться к каждому коду с помощью квадратных скобок. Вы не можете изменять отдельные символы строки, т.к. строки это массива-подобные неизменяемые объекты:

JS

```
var mystring = "Hello, World!";
var x = mystring.length;
mystring[0] = "L"; // Ничего не произойдёт, т.к. строки неизменяемые
mystring[0]; // Вернёт: "H"
```

Объект `String` имеет множество методов, в том числе и те, которые возвращают преобразованную исходную строку (методы `substring`, `toUpperCase` и другие).

В таблице ниже представлены методы String объекта.

| Метод | Описание |
| --- | --- |
| charAt, charCodeAt, codePointAt | Возвращает символ или символьный код в указанной позиции в строке. |
| indexOf, lastIndexOf | Возвращает первую (indexOf) или последнюю (lastIndexOf) позицию указанной подстроки в строке. Если данная подстрока не найдена, то возвращает -1. |
| startsWith, endsWith, includes | Проверяет, начинается/кончается/содержит ли строка указанную подстроку. |
| concat | Объединяет две строки и возвращает результат в качестве новой строки. |
| fromCharCode, fromCodePoint | Создаёт строку из указанной последовательности Unicode значений. Это метод класса String, а не отдельного экземпляра этого класса. |
| split | Разбивает строку на подстроки, результат возвращает в виде массива строк. |
| slice | Извлекает часть строки и возвращает её в качестве новой строки. |
| substring, substr | Возвращает указанную часть строки по начальному и конечному индексам, либо по начальному индексу и длине. |

| Метод | Описание |
| --- | --- |
| match , replace , search | Работа с регулярными выражениями. |
| toLowerCase , toUpperCase | Возвращает строку полностью в нижнем (toLowerCase) или верхнем (toUpperCase) регистре. |
| normalize | Возвращает нормализованную Unicode форму строки - значения объекта String, на котором вызывается. |
| repeat | Возвращает строку, которая представляет собой повторение исходной строки указанное количество раз. |
| trim | Убирает пробелы в начале и в конце строки, результат возвращается в качестве новой строки. |

## Многострочные шаблонные строки

Шаблонные строки представляют собой строковые литералы, которые могут содержать внутри себя встроенные выражения. С ними вы можете использовать многострочные строковые литералы и интерполяцию строк.

Такого типа строки заключаются в пару обратных штрихов (` `) (grave accent ) вместо двойных или одинарных кавычек. Шаблонные строки могут содержать заполнители, которые выделяются знаком доллара и фигурными скобками ( `${выражение}` ).

## Многострочная запись

Каждая новая горизонтальная линия символов, вставленная в исходный код, является частью шаблонной строки. Используя обычные строки, вам бы потребовалось использовать следующий синтаксис для многострочной записи:

JS

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

```
// "string text line 1
// string text line 2"
```

Того же результата можно добиться и другим способом (используя синтаксис шаблонных строк):

JS

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

### Встроенные выражения

Для того, чтобы добавить выражения внутрь обычных строк, вы бы использовали следующий синтаксис:

JS

```
var a = 5;
var b = 10;
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
// "Fifteen is 15 and
// not 20."
```

Теперь же, используя шаблонные строки, вы можете сделать это так:

JS

```
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and\nnot ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Для более подробной информации смотри Шаблонные строки в справочнике по JavaScript.

## Интернационализация

Объект `Intl` представляет собой пространство имён для ECMAScript API по интернационализации, которое обеспечивает чувствительное к языку сравнение строк, форматирование чисел, времени и даты. Конструкторы для объектов `Collator`, `NumberFormat` и `DateTimeFormat` являются свойствами объекта `Intl`.

## Форматирование времени и даты

Объект `DateTimeFormat` полезен для форматирования времени и даты. В примере ниже дата форматируется так, как это принято в США (результат отличен для разных временных зон).

```
JS
```

```
var msPerDay = 24 * 60 * 60 * 1000;

// July 17, 2014 00:00:00 UTC.
var july172014 = new Date(msPerDay * (44 * 365 + 11 + 197));

var options = {
  year: "2-digit",
  month: "2-digit",
  day: "2-digit",
  hour: "2-digit",
  minute: "2-digit",
  timeZoneName: "short",
};
var americanDateTime = new Intl.DateTimeFormat("en-US", options).format;

console.log(americanDateTime(july172014)); // 07/16/14, 5:00 PM PDT
```

## Форматирование чисел

Объект `NumberFormat` полезен при форматировании чисел, например, валют.

```
JS
```

```
var gasPrice = new Intl.NumberFormat("en-US", {
  style: "currency",
  currency: "USD",
  minimumFractionDigits: 3,
});
```

```
console.log(gasPrice.format(5.259)); // $5.259

var hanDecimalRMBInChina = new Intl.NumberFormat("zh-CN-u-nu-hanidec", {
  style: "currency",
  currency: "CNY",
});

console.log(hanDecimalRMBInChina.format(1314.25)); // ￥ 一,三一四.二五
```

## Сравнение

Объект `Collator` полезен для сравнения и сортировки строк.

Например, в Германии есть два различных порядка сортировки строк в зависимости от документа: телефонная книга или словарь. Сортировка по типу телефонной книги подчёркивает звуки.

JS

```
var names = ["Hochberg", "Hönigswald", "Holzman"];

var germanPhonebook = new Intl.Collator("de-DE-u-co-phonebk");

// as if sorting ["Hochberg", "Hoenigswald", "Holzman"]:
console.log(names.sort(germanPhonebook.compare).join(", "));
// logs "Hochberg, Hönigswald, Holzman"
```

Примером по сортировке для словаря слов на немецком языке служит следующий код:

JS

```
var germanDictionary = new Intl.Collator("de-DE-u-co-dict");

// as if sorting ["Hochberg", "Honigswald", "Holzman"]:
console.log(names.sort(germanDictionary.compare).join(", "));
// logs "Hochberg, Holzman, Hönigswald"
```

Для более подробной информации об `Intl` API смотри Introducing the JavaScript Internationalization API .

## Регулярные выражения

Регулярные выражения - это шаблоны, которые используются для описания некоторого множества строк . Это очень мощный и в некоторый степени непростой механизм, и поэтому ему посвящена отдельная глава. Узнать больше о регулярных выражениях можно здесь:

- Регулярные выражения JavaScript в руководстве по JavaScript.
- RegExp ссылка в документации.