

Современный учебник JavaScript

© Илья Кантор

Сборка от 23 июля 2013 для чтения с устройств

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.

Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу <http://learn.javascript.ru>.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru.

Глава: Структуры данных

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Строки

- Создание строк

- Специальные символы

- Экранирование специальных символов

- Методы и свойства

- Длина length

- Доступ к символам

- Изменения строк

- Смена регистра

- Поиск подстроки

- Поиск всех вхождений

- Взятие подстроки: substr, substring, slice.

- Отрицательные аргументы

- Кодировка Юникод

- Сравнение строк

- Итого

Числа

- Способы записи

- Деление на ноль, Infinity

- NaN

- isFinite(n)

- Преобразование к числу

 - isNaN — проверка на число для строк

- Мягкое преобразование: parseInt и parseFloat

- Проверка на число для всех типов

- toString(система счисления)

- Округление

 - Округление до заданной точности

 - num.toFixed(precision)

- Неточные вычисления

- Другие математические методы

 - Тригонометрия

 - Функции общего назначения

- Итого

Объекты как ассоциативные массивы

- Ассоциативные массивы

- Создание объектов

- Операции с объектом

 - Доступ через квадратные скобки

 - Объявление со свойствами

 - Вложенные объекты в объявлении

- Перебор свойств и значений

 - Количество свойств в объекте

 - Порядок перебора свойств

- Передача по ссылке

Компактное представление объектов

Итого

Массивы с числовыми индексами

Объявление

Методы pop/push, shift/unshift

Конец массива

Начало массива

Внутреннее устройство массива

Влияние на быстродействие

Перебор элементов

Особенности работы length

Используем length для укорачивания массива

Создание вызовом new Array

new Array()

Многомерные массивы

Внутреннее представление массивов

Итого

Массивы: методы

Object.keys(obj)

Метод split

Метод join

Удаление из массива

Метод splice

Метод slice

Сортировка, метод sort(fn)

Свой порядок сортировки

reverse

concat

indexOf/lastIndexOf

Итого

Дата и Время

Создание

Получение компонент даты

Установка компонент даты

Автоисправление даты

Преобразование к числу, разность дат

Бенчмаркинг

Форматирование

Разбор строки, Date.parse

Итого

Преобразование типов

Строковое преобразование

Численное преобразование

Специальные значения

Логическое преобразование

Итого

Решения задач

Строки

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

Внутренним форматом строк, вне зависимости от кодировки страницы, является [Юникод \(Unicode\)](#) .

Создание строк

Строки создаются при помощи двойных или одинарных кавычек:

```
1 | var text = "моя строка";
2 |
3 | var anotherText = 'еще строка';
4 |
5 | var str = "012345";
```

В JavaScript **нет** разницы между двойными и одинарными кавычками.

Специальные символы

Строки могут содержать специальные символы. Самый часто

используемый из таких символов — это *перевод строки*.

Он обозначается как `\n`, например:

```
1 | alert('Привет\nМир'); // выведет "Мир" на новой строке
```

Есть и более редкие символы, вот их список:

Специальные символы

Символ	Описание
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\uNNNN</code>	Символ в кодировке Юникод с шестнадцатиричным кодом NNNN. Например, <code>\u00A9</code> — юникодное представление символа копирайт ©

Экранирование специальных символов

Если строка в одинарных кавычках, то внутренние одинарные кавычки внутри должны быть *экранированы*, то есть снабжены обратным слешем `\`, вот так:

```
var str = 'I\'m a JavaScript programmer';
```

В двойных кавычках — экранируются внутренние двойные:

```
1 | var str = "I'm a JavaScript \"programmer\" ";
2 | alert(str);
```

Экранирование служит исключительно для правильного восприятия строки JavaScript. **В памяти строка будет содержать сам символ без `'\'`**. Вы можете увидеть это, запустив пример выше.

Сам символ обратного слэша `'\'` является служебным, поэтому всегда экранируется, т.е пишется как `\\`:

```
1 | var str = ' символ \\' ;
2 |
3 | alert(str); // символ \
```

Заэкранировать можно любой символ. Если он не специальный, то ничего не произойдёт:

```
1 | alert( "\a" ); // a
2 | // идентично alert( "a" );
```

Методы и свойства

Здесь мы рассмотрим методы и свойства строк, с некоторыми из которых мы знакомились ранее, в главе [Методы и свойства](#).

Длина length

Одно из самых частых действий со строкой — это получение ее длины:

```
1 | var str = "My\n"; // 3 символа. Третий - перевод строки
2 |
3 | alert(str.length); // 3
```

Доступ к символам

Чтобы получить символ, используйте вызов `charAt(позиция)`. Первый символ имеет позицию 0:

```
1 | var str = "jQuery";
2 | alert( str.charAt(0) ); // "j"
```

В JavaScript **нет отдельного типа «символ»**, так что `charAt` возвращает строку, состоящую из выбранного символа.



В современных браузерах (не IE7-) для доступа к символу можно также использовать квадратные скобки:

```
1 | var str = "Я - современный браузер!";
2 | alert(str[0]); // "Я", IE8+
```

Разница между этим способом и `charAt` заключается в том, что если символа нет — `charAt` выдает пустую строку, а скобки — `undefined`:

```
1 | alert( "".charAt(0) ); // пустая строка
2 | alert( ""[0] ); // undefined, IE8+
```



Вызов метода — всегда со скобками

Обратите внимание, `str.length` — это *свойство* строки, а `str.charAt(pos)` — *метод*, т.е. функция.

Обращение к методу всегда идет со скобками, а к свойству — без скобок.

Изменения строк

Строки в JavaScript нельзя изменять. Можно прочитать символ, но нельзя заменить его. Как только строка создана — она такая навсегда.

Чтобы это обойти, создаётся новая строка и присваивается в переменную вместо старой:

```
1 | var str = "строка";
2 |
3 | str = str.charAt(3) + str.charAt(4) + str.charAt(5);
4 |
5 | alert(str); // ока
```

Смена регистра

Методы `toLowerCase()` и `toUpperCase()` меняют регистр строки на нижний/верхний:

```
1 | alert( "Интерфейс".toUpperCase() ); // ИНТЕРФЕЙС
```

Пример ниже получает первый символ и приводит его к нижнему регистру:

```
alert( "Интерфейс".charAt(0).toLowerCase() ); // 'и'
```



Задача: Сделать первый символ заглавным

Напишите функцию `ucFirst(str)`, которая возвращает строку `str` с заглавным первым символом, например:

Важность: 5

```
ucFirst("вася") == "Вася";
ucFirst("") == ""; // нет ошибок при пустой строке
```

P.S. В JavaScript нет встроенного метода для этого. Создайте

функцию, используя `toUpperCase()` и `charAt()`.

Решение задачи "Сделать первый символ заглавным" »»

Поиск подстроки

Для поиска подстроки есть метод `indexOf(подстрока[, начальная_позиция])` .

Он возвращает позицию, на которой находится подстрока или `-1`, если ничего не найдено. Например:

```
1 var str = "Widget with id";
2
3 alert( str.indexOf("Widget") ); // 0, т.к. "Widget" найден
  прямо в начале str
4 alert( str.indexOf("id") ); // 1, т.к. "id" найден,
  начиная с позиции 1
5 alert( str.indexOf("Lalala") ); // -1, подстрока не
  найдена
```

Необязательный второй аргумент позволяет искать, начиная с указанной позиции. Например, первый раз `"id"` появляется на позиции 1. Чтобы найти его следующее появление — запустим поиск с позиции 2:

```
1 var str = "Widget with id";
2
3 alert( str.indexOf("id", 2) ); // 12, поиск начат с позиции
  2
```

Также существует аналогичный метод `lastIndexOf` , который ищет не с начала, а с конца строки.



Для красивого вызова `indexOf` применяется побитовый оператор НЕ `~`.

Дело в том, что вызов `~n` эквивалентен выражению `-(n+1)`, например:

```
1 alert( ~2 ); // -(2+1) = -3
2 alert( ~1 ); // -(1+1) = -2
3 alert( ~0 ); // -(0+1) = -1
4 alert( ~-1 ); // -(-1+1) = 0
```

Как видно, `~n` — ноль только в случае, когда `n == -1`.

То есть, проверка `if (~str.indexOf(...))` означает, что результат `indexOf` отличен от `-1`, т.е. совпадение есть.

Вот так:

```
1 | var str = "Widget";
2 |
3 | if( ~str.indexOf("get") ) {
4 |     alert('совпадение есть!');
5 | }
```

Вообще, использовать возможности языка неочевидным образом не рекомендуется, поскольку ухудшает читаемость кода.

Однако, в данном случае, все в порядке. Просто запомните: `'~'` читается как «не минус один», а `"if ~str.indexOf"` читается как "если найдено".



Задача: Проверьте спам

Напишите функцию `checkSpam(str)`, которая возвращает `true`, если строка `str` содержит `'viagra'` or `'XXX'`.

Важность: 5

Функция должна быть нечувствительна к регистру:

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

[Решение задачи "Проверьте спам" >>](#)

Поиск всех вхождений

Чтобы найти все вхождения подстроки, нужно запустить `indexOf` в цикле. Как только получаем очередную позицию — начинаем следующий поиск со следующей.

Пример такого цикла:

```
01 | var str = "Ослик Иа-Иа посмотрел на виадук"; // ищем в
    | этой строке
02 | var target = "Иа"; // цель поиска
03 |
04 | var pos = 0;
```

```

05 while(true) {
06     var foundPos = str.indexOf(target, pos);
07     if (foundPos == -1) break;
08
09     alert(foundPos); // нашли на этой позиции
10     pos = foundPos + 1; // продолжить поиск со следующей
11 }

```

Такой цикл начинает поиск с позиции 0, затем найдя подстроку на позиции foundPos, следующий поиск продолжит с позиции pos = foundPos+1, и так далее, пока что-то находит.

Впрочем, тот же алгоритм можно записать и короче:

```

1 var str = "Ослик Иа-Иа посмотрел на виадук"; // ищем в
  этой строке
2 var target = "Иа"; // цель поиска
3
4 var pos = -1;
5 while ( (pos = str.indexOf(target, pos+1)) != -1) {
6     alert(pos);
7 }

```

Взятие подстроки: substr, substring, slice.

В JavaScript существуют целых 3 (!) метода для взятия подстроки, с небольшими отличиями между ними.

substring(start [, end])

Метод substring(start, end) возвращает подстроку с позиции start до, но не включая end.

```

1 var str = "stringify";
2 alert(str.substring(0,1)); // "s", символы с позиции 0
  по 1 не включая 1.

```

Если аргумент end отсутствует, то идет до конца строки:

```

1 var str = "stringify";
2 alert(str.substring(2)); // ringify, символы с позиции 2
  до конца

```

substr(start [, length])

Первый аргумент имеет такой же смысл, как и в substring, а второй содержит не конечную позицию, а количество символов.

```
1 | var str = "stringify";
2 | str = str.substr(2,4); // ring, со 2й позиции 4 символа
3 | alert(str)
```

Если второго аргумента нет — подразумевается «до конца строки».

slice(start [, end])

Возвращает часть строки от позиции start до, но не включая, позиции end. Смысл параметров — такой же как в substring.

Отрицательные аргументы

Различие между substring и slice — в том, как они работают с отрицательными и выходящими за границу строки аргументами:

substring(start, end)

Отрицательные аргументы интерпретируются как равные нулю. Слишком большие значения усекаются до длины строки:

```
1 | alert( "testme".substring(-2) ); // "testme", -2
   | становится 0
```

Кроме того, если start > end, то аргументы меняются местами, т.е. возвращается участок строки между start и end:

```
1 | alert( "testme".substring(4, -1) ); // "test"
2 | // -1 становится 0 -> получили substring(4, 0)
3 | // 4 > 0, так что аргументы меняются местами ->
   | substring(0, 4) = "test"
```

slice

Отрицательные значения отсчитываются от конца строки:

```
1 | alert( "testme".slice(-2) ); // "me", от 2 позиции с
   | конца

1 | alert( "testme".slice(1, -1) ); // "estm", от 1 позиции
   | до первой с конца.
```

Это гораздо более удобно, чем странная логика substring.

Отрицательное значение первого параметра поддерживается в substr во всех браузерах, кроме IE8-.

Выводы.

Самый удобный метод — это `slice(start, end)`.

В качестве альтернативы можно использовать `substr(start, length)`, помня о том, что IE8- не поддерживает отрицательный `start`.



Задача: 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 ее длина стала равна `maxLength`.

Результатом функции должна быть (при необходимости) усечённая строка.

Например:

```
truncate("Вот, что мне хотелось бы сказать на эту тему:", 20) = "Вот, что мне хотело..."
```

```
truncate("Всем привет!", 20) = "Всем привет!"
```

Эта функция имеет применение в жизни. Она используется, чтобы усекать слишком длинные темы сообщений.

[Решение задачи "Усечение строки" »»](#)

Кодировка Юникод

Если вы знакомы со сравнением строк в других языках, то позвольте предложить одну маленькую загадку. Даже не одну, а целых две.

Как мы знаем, символы сравниваются в алфавитном порядке

`'А' < 'Б' < 'В' < ... < 'Я'`.

Но есть несколько странностей..

1. Почему буква `'а'` маленькая больше буквы `'я'` большой?

```
1 | alert( 'а' > 'я' ); // true
```

2. Буква 'ё' находится в алфавите между е и ж: абвгдеёжз... Но почему тогда 'ё' больше 'я'?

```
1 | alert( 'ё' > 'я' ); // true
```

Чтобы разобраться с этим, обратимся к внутреннему представлению строк в JavaScript.

Все строки имеют внутреннюю кодировку [Юникод](#) .

Неважно, на каком языке написана страница, находится ли она в windows-1251 или utf-8. Внутри JavaScript-интерпретатора все строки приводятся к единому «юникодному» виду. Каждому символу соответствует свой код.

Есть метод для получения символа по его коду:

String.fromCharCode(code)

Возвращает символ по коду code:

```
1 | alert( String.fromCharCode(1072) ); // 'а'
```

... И метод для получения цифрового кода из символа:

str.charCodeAt(pos)

Возвращает код символа на позиции pos. Отсчет позиции начинается с нуля.

```
1 | alert( "абрикос".charCodeAt(0) ); // 1072, код 'а'
```

Теперь вернемся к примерам выше. Почему сравнения 'ё' > 'я' и 'а' > 'я' дают такой странный результат?

Дело в том, что **символы сравниваются не по алфавиту, а по коду**. У кого код больше — тот и больше. В юникоде есть много разных символов. Кириллическим буквам соответствует только небольшая часть из них, подробнее — [Кириллица в Юникоде](#) .

Выведем отрезок символов юникода с кодами от 1034 до 1113:

```
1 | var str = '';  
2 | for (var i=1034; i<=1113; i++) {  
3 |     str += String.fromCharCode(i);  
4 | }  
5 | alert(str);
```

Результат:

ЪЪКЙЎЦАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчщ

Мы можем увидеть из этого отрезка две важных вещи:

1. **Строчные буквы идут после заглавных, поэтому они всегда больше.**

В частности, 'а' (код 1072) > 'я' (код 1071).

То же самое происходит и в английском алфавите, там 'а' > 'z'.

2. **Ряд букв, например ё, находятся вне основного алфавита.**

В частности, маленькая буква ё имеет код, больший чем я, поэтому

'ё' (код 1105) > 'я' (код 1103).

Кстати, большая буква Ё располагается в Unicode до А, поэтому

'Ё' (код 1025) < 'А' (код 1040). Удивительно: есть буква меньше чем А



Юникод в HTML

Кстати, если мы знаем код символа в кодировке юникод, то можем добавить его в HTML, используя «числовую ссылку» (numeric character reference).

Для этого нужно написать сначала &#, затем код, и завершить точкой с запятой ';'. Например, символ 'а' в виде числовой ссылки: а.

Если код хотят дать в 16-ричной системе счисления, то начинают с &#x.

В юникоде есть много забавных и полезных символов, например, символ ножниц: (✂), дроби: ½ (½) ¾ (¾) и другие. Их можно удобно использовать вместо картинок в дизайне.

Сравнение строк

Строки сравниваются *лексикографически*, в порядке «телефонного справочника».

Сравнение строк s1 и s2 обрабатывается по следующему алгоритму:

1. Сравняются первые символы: `a = s1.charAt(0)` и `b = s2.charAt(0)`. Если они одинаковы, то следующий шаг, иначе, в зависимости от результата их сравнения, вернуть `true` или `false`
2. Сравняются вторые символы, затем третьи и так далее... Если в одной строке закончились символы, то она меньше. Если в обеих закончились — они равны.

Спецификация языка определяет этот алгоритм более детально, но смысл в точности соответствует порядку, по которому имена заносятся в телефонный справочник.

```
"Z" > "A" // true
```

```
"Вася" > "Ваня" // true, т.к. с > н
```

```
"aa" > "a" // true, т.к. начало совпадает, но в 1й строке больше символов
```



Числа в виде строк сравниваются как строки

Бывает, что числа приходят в скрипт в виде строк, например как результат `prompt`. В этом случае результат их сравнения будет неверным:

```
1 | alert("2" > "14"); // true, так как это строки, и для первых символов верно "2" > "1"
```

Чтобы получать верный результат, хотя бы один из аргументов должен не быть строкой. Тогда и другой будет преобразован к числу:

```
1 | alert(2 > "14"); // false
```

Итого

- Строки в JavaScript имеют внутреннюю кодировку Юникод. При написании строки можно использовать специальные символы, например `\n` и вставлять юникодные символы по коду.
- Мы познакомились со свойством `length` и методами `charAt`, `toLowerCase`/`toUpperCase`, `substring`/`substr`/`slice` (предпочтителен `slice`)

- Строки сравниваются побуквенно. Поэтому если число получено в виде строки, то такие числа могут сравниваться некорректно, нужно преобразовать его к типу *number*.
- При сравнении строк следует иметь в виду, что буквы сравниваются по их кодам. Поэтому большая буква меньше маленькой, а буква ё вообще вне основного алфавита.

Больше информации о методах для строк можно получить в справочнике: <http://javascript.ru/String>.

Числа

Все числа в JavaScript, как целые так и дробные, имеют тип `Number` и хранятся в 64-битном формате [IEEE-754](#), также известном как «double precision».

Здесь мы рассмотрим различные тонкости, связанные с работой с числами в JavaScript.

Способы записи

В JavaScript можно записывать числа не только в десятичной, но и в шестнадцатеричной (начинается с `0x`), а также восьмеричной (начинается с `0`) системах счисления:

```
1 alert( 0xFF ); // 255 в шестнадцатеричной системе
2 alert( 010 ); // 8 в восьмеричной системе
```

Также доступна запись в «*научном формате*» (ещё говорят «запись с плавающей точкой»), который выглядит как `<число>e<кол-во нулей>`.

Например, `1e3` — это 1 с 3 нулями, то есть 1000.

```
1 // еще пример научной формы: 3 с 5 нулями
2 alert( 3e5 ); // 300000
```

Если количество нулей отрицательно, то число сдвигается вправо за десятичную точку, так что получается десятичная дробь:

```
1 // здесь 3 сдвинуто 5 раз вправо, за десятичную точку.
2 alert( 3e-5 ); // 0.00003 <-- 5 нулей, включая начальный
```


Деление на ноль, Infinity

Представьте, что вы собираетесь создать новый язык... Люди будут называть его «JavaScript» (или LiveScript... неважно).

Что должно происходить при попытке деления на ноль?

Как правило, ошибка в программе... Во всяком случае, в большинстве языков программирования это именно так.

Но создатель JavaScript решил пойти математически правильным путем. Ведь чем меньше делитель, тем больше результат. При делении на очень-очень маленькое число должно получиться очень большое. В математическом анализе это описывается через **пределы**, и если подразумевать предел, то в качестве результата деления на 0 мы получаем «бесконечность», которая обозначается символом ∞ или, в JavaScript: "Infinity".

```
1 | alert(1/0); // Infinity
2 | alert(12345/0); // Infinity
```

Infinity — особенное численное значение, которое ведет себя в точности как математическая бесконечность ∞ .

- Infinity больше любого числа.
- Добавление к бесконечности не меняет её.

```
1 | alert(Infinity > 1234567890); // true
2 | alert(Infinity + 5 == Infinity); // true
```

Бесконечность можно присвоить и в явном виде: `var x = Infinity`.

Бывает и минус бесконечность `-Infinity`:

```
1 | alert( -1 / 0 ); // -Infinity
```

Бесконечность можно получить также, если сделать ну очень большое число, для которого количество разрядов в двоичном представлении не помещается в соответствующую часть стандартного 64-битного формата, например:

```
1 | alert( 1e500 ); // Infinity
```

NaN

Если математическая операция не может быть совершена, то возвращается специальное значение NaN (Not-A-Number).

Например, деление $0/0$ в математическом смысле неопределено, поэтому возвращает NaN:

```
1 | alert( 0 / 0 ); // NaN
```

Значение NaN используется для обозначения математической ошибки и обладает следующими свойствами:

- Значение NaN — единственное, в своем роде, которое *не равно ничему, включая себя*.

Следующий код ничего не выведет:

```
1 | if (NaN == NaN) alert("=="); // Ни один вызов
2 | if (NaN === NaN) alert("==="); // не сработает
```

- Значение NaN можно проверить специальной функцией `isNaN(n)`, которая возвращает `true` если аргумент — NaN и `false` для любого другого значения.

```
1 | var n = 0/0;
2 |
3 | alert( isNaN(n) ); // true
```



Ещё один забавный способ проверки значения на NaN — это проверить его на равенство самому себе, вот так:

```
1 | var n = 0/0;
2 |
3 | if (n !== n) alert('n = NaN!');
```

Это работает, но для наглядности лучше использовать `isNaN`.

- Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN.

```
1 | alert( NaN + 1 ); // NaN
```

Если аргумент `isNaN` — не число, то он автоматически преобразуется к числу.

Никакие математические операции в JavaScript не могут привести к ошибке или «обрушить» программу.

В худшем случае, результат будет `NaN`.

`isFinite(n)`

Итак, в JavaScript есть обычные числа и три специальных числовых значения: `NaN`, `Infinity` и `-Infinity`.

Функция `isFinite(n)` возвращает `true` только тогда, когда `n` — обычное число, а не одно из этих значений:

```
1 | alert( isFinite(1) ); // true
2 | alert( isFinite(Infinity) ); // false
3 | alert( isFinite(NaN) ); // false
```

Если аргумент `isFinite` — не число, то он автоматически преобразуется к числу.

Преобразование к числу

Строгое преобразование можно осуществить унарным плюсом `'+'`

```
1 | var s = "12.34";
2 | alert( +s ); // 12.34
```

Строгое — означает, что если строка не является в точности числом, то результат будет `NaN`:

```
1 | alert( +"12test" ); // NaN
```

Единственное исключение — пробельные символы в начале и в конце строки, которые игнорируются:

```
1 | alert( +" -12" ); // -12
2 | alert( +" \n34 \n" ); // 34, перевод строки \n является
   пробельным символом
```

```
3 | alert( "+" ); // 0, пустая строка становится нулем
4 | alert( +"1 2" ); // NaN, пробел посередине числа - ошибка
```

Аналогичным образом происходит преобразование и в других математических операторах и функциях:

```
1 | alert( '12.34' / "-2" ); // -6.17
```



Задача: Интерфейс sum

Создайте страницу, которая предлагает ввести два числа и выводит их сумму.

Важность: 5

Работать должно так:

<http://learn.javascript.ru/files/tutorial/intro/sum.html>.

P.S. Есть «подводный камень» при работе с типами.

[Решение задачи "Интерфейс sum" »»](#)

isNaN Ñ проверка на число для строк

Функция `isNaN` является математической, она преобразует аргумент в число, а затем проверяет, NaN это или нет.

Поэтому можно использовать ее для проверки:

```
1 | var x = "-11.5";
2 | if (isNaN(x)) {
3 |     alert("Строка преобразовалась в NaN. Не число");
4 | } else {
5 |     alert("Число");
6 | }
```

Единственный тонкий момент — в том, что пустая строка и строка из пробельных символов преобразуются к 0:

```
1 | alert(isNaN(" \n\n ")) // false, т.к. строка из пробелов
   | преобразуется к 0
```

И, конечно же, проверка `isNaN` посчитает числами значения `false`, `true`, `null`, т.к. они хотя и не числа, но преобразуются к ним:

```
1 | +false = 0
2 | +true = 1
3 | +null = 0
```

```
4 | +undefined = NaN;
```

Мягкое преобразование: parseInt и parseFloat

В мире HTML/CSS многие значения не являются в точности числами. Например, метрики CSS: 10pt или -12px.

Оператор '+' для таких значений возвратит NaN:

```
1 | alert( +"12px" ) // NaN
```

Для удобного чтения таких значений существует функция parseInt:

```
1 | alert( parseInt('12px') ); // 12
```

parseInt и ее аналог parseFloat преобразуют строку символ за символом, пока это возможно.

При возникновении ошибки возвращается число, которое получилось. parseInt читает из строки целое число, а parseFloat — дробное.

```
1 | alert( parseInt('12px') ) // 12, ошибка на символе 'p'
2 | alert( parseFloat('12.3.4') ) // 12.3, ошибка на второй
   | точке
```

Конечно, существуют ситуации, когда parseInt/parseFloat возвращают NaN. Это происходит при ошибке на первом же символе:

```
1 | alert( parseInt('a123') ); // NaN
```



Ошибка parseInt('0..')

parseInt (но не parseFloat) понимает 16-ричную систему счисления:

```
1 | alert( parseInt('0xFF') ) // 255
```

В старом стандарте JavaScript он умел понимать и восьмеричную:

```
1 | alert( parseInt('010') ) // в некоторых браузерах
   | 8
```

Если вы хотите быть уверенным, что число, начинающееся с

нуля, будет интерпретировано верно — используйте второй необязательный аргумент `parseInt` — основание системы счисления:

```
1 | alert( parseInt('010', 10) ); // во всех браузерах  
  | 10
```

Проверка на число для всех типов

Если вам нужна действительно точная проверка на число, которая не считает числом строку из пробелов, логические и специальные значения — используйте следующую функцию `isNumeric`:

```
function isNumeric(n) {  
  return !isNaN(parseFloat(n)) && isFinite(n);  
}
```

Разберёмся, как она работает. Начнём справа.

→ Функция `isFinite(n)` преобразует аргумент к числу и возвращает `true`, если это не `Infinity`/`-Infinity`/`NaN`.

Таким образом, правая часть отсеет заведомо не-числа, но оставит такие значения как `true/false/null` и пустую строку `' '`, т.к. они корректно преобразуются в числа.

→ Для их проверки нужна левая часть. Вызов `parseFloat(true/false/null/' ')` вернёт `NaN` для этих значений.

Так устроена функция `parseFloat`: она преобразует аргумент к строке, т.е. `true/false/null` становятся `"true"/"false"/"null"`, а затем считывает из неё число, при этом пустая строка даёт `NaN`.

В результате отсеивается всё, кроме строк-чисел и обычных чисел.

`toString(система счисления)`

Как показано выше, числа можно записывать не только в 10-чной, но и в 16-ричной системе. Но бывает и противоположная задача: получить 16-ричное представление числа. Для этого используется метод `toString(основание системы)`, например:

```
1 | var n = 255;
2 |
3 | alert( n.toString(16) ); // ff
```

Основание может быть любым от 2 до 36.

→ Основание 2 бывает полезно для отладки битовых операций, которые мы пройдем чуть позже:

```
1 | var n = 4;
2 | alert( n.toString(2) ); // 100
```

→ Основание 36 (по количеству букв в английском алфавите — 26, вместе с цифрами, которых 10) используется для того, чтобы «кодировать» число в виде буквенно-цифровой строки. В этой системе счисления сначала используются цифры, а затем буквы от a до z:

```
1 | var n = 1234567890;
2 | alert( n.toString(36) ); // kf12oi
```

При помощи такого кодирования можно сделать длинный цифровой идентификатор короче, чтобы затем использовать его в URL.

Округление

Одна из самых частых операций с числом — округление. В JavaScript существуют целых 3 функции для этого.

Math.floor

Округляет вниз

Math.ceil

Округляет вверх

Math.round

Округляет до ближайшего целого

```
1 | alert( Math.floor(3.1) ); // 3
2 | alert( Math.ceil(3.1) ); // 4
3 | alert( Math.round(3.1) ); // 3
```



Округление битовыми операторами

Битовые операторы делают любое число 32-битным целым,

обрезаая десятичную часть.

В результате побитовая операция, которая не изменяет число, например, двойное битовое НЕ — округляет его:

```
1 | alert( ~~12.3 ); // 12
```

Любая побитовая операция такого рода подойдет, например XOR (исключающее ИЛИ, "^") с нулем:

```
1 | alert( 12.3 ^ 0 ); // 12
2 | alert( 1.2 + 1.3 ^ 0 ); // 2, приоритет ^ меньше,
   | чем +
```

Это удобно в первую очередь тем, что легко читается и не заставляет ставить дополнительные скобки как `Math.floor(...)`:

```
var x = a * b / c ^ 0; // читается так: "a*b/c и
                        округлить"
```

Округление до заданной точности

Обычный трюк — это умножить и поделить на 10 с нужным количеством нулей. Например, округлим 3.456 до 2го знака после запятой:

```
1 | var n = 3.456;
2 | alert( Math.round( n * 100 ) / 100 ); // 3.456 -> 345.6 -
   | > 346 -> 3.46
```

Таким образом можно округлять число и вверх и вниз.

num.toFixed(precision)

Существует специальный метод `num.toFixed(precision)`, который округляет число `num` до точности `precision` и возвращает результат в виде строки:

```
1 | var n = 12.34;
2 | alert( n.toFixed(1) ); // "12.3"
```

Округление идёт до ближайшего значения, аналогично `Math.round`:

```
1 | var n = 12.36;
2 | alert( n.toFixed(1) ); // "12.4"
```


Итоговая строка, при необходимости, дополняется нулями до нужной точности:

```
1 | var n = 12.34;
2 | alert( n.toFixed(5) ); // "12.34000", добавлены нули до 5
   | знаков после запятой
```

Если нам нужно именно число, то мы можем получить его, применив '+' к результату `n.toFixed(..)`:

```
1 | var n = 12.34;
2 | alert( +n.toFixed(5) ); // 12.34
```

Метод `toFixed` не эквивалентен `Math.round`!

Например, произведём округление до одного знака после запятой с использованием двух способов:

```
1 | var price = 6.35;
2 |
3 | alert( price.toFixed(1) ); // 6.3
4 | alert( Math.round(price*10)/10 ); // 6.4
```

Как видно, результат разный! Вариант округления через `Math.round` получился более корректным, так как по общепринятым правилам 5 округляется вверх. А `toFixed` может округлить его как вверх, так и вниз. Почему? Скоро узнаем!

Неточные вычисления

Запустите этот пример:

```
1 | alert(0.1 + 0.2 == 0.3);
```

Запустили? Если нет — все же сделайте это.

Ок, вы запустили его. Результат несколько странный, не так ли?

Возможно, ошибка в браузере? Поменяйте браузер, запустите еще раз.

Хорошо, теперь мы можем быть уверены: `0.1 + 0.2` это не `0.3`. Но тогда что же это?

```
1 | alert(0.1 + 0.2); // 0.30000000000000004
```

Как видите, произошла небольшая вычислительная ошибка.

Дело в том, что в стандарте IEEE 754 на число выделяется ровно 8 байт(=64 бита), не больше и не меньше.

Число 0.1 (=1/10) короткое в десятичном формате, а в двоичной системе счисления это бесконечная дробь ([перевод десятичной дроби в двоичную систему](#)). Также бесконечной дробью является 0.2 (=2/10).

Двоичное значение бесконечных дробей хранится только до определенного знака, поэтому возникает неточность. Это даже можно увидеть:

```
1 | alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

Когда мы складываем 0.1 и 0.2, то две неточности складываются, получаем третью.

Конечно, это не означает, что точные вычисления для таких чисел невозможны. Они возможны. И даже необходимы.

Например, есть два способа сложить 0.1 и 0.2:

1. Сделать их целыми, сложить, а потом поделить:

```
1 | alert( (0.1*10 + 0.2*10) / 10 ); // 0.3
```

Это работает, т.к. числа $0.1 \cdot 10 = 1$ и $0.2 \cdot 10 = 2$ могут быть точно представлены в двоичной системе.

2. Сложить, а затем округлить до разумного знака после запятой.

Округления до 10-го знака обычно бывает достаточно, чтобы отсечь ошибку вычислений:

```
1 | var result = 0.1 + 0.2;  
2 | alert( +result.toFixed(10) ); // 0.3
```



Задача: Почему `6.35.toFixed(1) == 6.3`?

В математике принято, что 5 округляется вверх, например:

Важность: 4

```
1 | alert( 1.5.toFixed(0) ); // 2  
2 | alert( 1.35.toFixed(1) ); // 1.4
```

Но почему в примере ниже 6.35 округляется до 6.3?

```
1 | alert( 6.35.toFixed(1) ); // 6.3
```

[Решение задачи "Почему 6.35.toFixed\(1\) == 6.3?" »»](#)



Задача: Сложение цен

Представьте себе электронный магазин. Цены даны с точностью до копейки(цента, евроцента и т.п.). Важность: 5

Вы пишете интерфейс для него. Основная работа происходит на сервере, но и на клиенте все должно быть хорошо. Сложение цен на купленные товары и умножение их на количество является обычной операцией.

Получится глупо, если при заказе двух товаров с ценами 0.10\$ и 0.20\$ человек получит общую стоимость 0.300000000000000004\$:

```
1 | alert( 0.1 + 0.2 + '$' );
```

Что можно сделать, чтобы избежать проблем с ошибками округления?

[Решение задачи "Сложение цен" »»](#)



Забавный пример

Привет! Я — число, растущее само по себе!

```
1 | alert(9999999999999999);
```

Причина та же — потеря точности.

Из 64 бит, отведённых на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит — знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

Ради справедливости заметим, что в точности то же самое происходит в любом другом языке, где используется формат IEEE 754, включая Java, C, PHP, Ruby, Perl.

Другие математические методы

JavaScript предоставляет базовые тригонометрические и некоторые другие функции для работы с числами.

Тригонометрия

Встроенные функции для тригонометрических вычислений:

Math.acos(x)

Возвращает арккосинус x (в радианах)

Math.asin(x)

Возвращает арксинус x (в радианах)

Math.atan

Возвращает арктангенс x (в радианах)

Math.atan2(y, x)

Возвращает угол до точки (y, x) . Описание функции: [Atan2](#) .

Math.sin(x)

Вычисляет синус x (в радианах)

Math.cos(x)

Вычисляет косинус x (в радианах)

Math.tan(x)

Возвращает тангенс x (в радианах)

Функции общего назначения

Разные полезные функции:

Math.sqrt(x)

Возвращает квадратный корень из x .

Math.log(x)

Возвращает натуральный (по основанию e) логарифм x .

Math.pow(x, exp)

Возводит число в степень, возвращает x^{exp} , например

`Math.pow(2, 3) = 8`. Работает в том числе с дробными и

отрицательными степенями, например: `Math.pow(4, -1/2) = 0.5`.

`Math.abs(x)`

Возвращает абсолютное значение числа

`Math.exp(x)`

Возвращает e^x , где e — основание натуральных логарифмов.

`Math.max(a, b, c...)`

Возвращает наибольший из списка аргументов

`Math.min(a, b, c...)`

Возвращает наименьший из списка аргументов

`Math.random()`

Возвращает псевдо-случайное число в интервале $[0,1)$ - то есть между 0(включительно) и 1(не включая). Генератор случайных чисел инициализируется текущим временем.

Итого

- Числа могут быть записаны в шестнадцатичной, восьмеричной системе, а также «научным» способом.
- В JavaScript существует числовое значение бесконечность `Infinity`.
- Ошибка вычислений дает `NaN`.
- Арифметические и математические функции преобразуют строку в точности в число, игнорируя начальные и конечные пробелы.
- Функции `parseInt/parseFloat` делают числа из строк, которые начинаются с числа.
- Есть четыре способа округления: `Math.floor`, `Math.round`, `Math.ceil` и битовый оператор. Для округления до нужного знака используйте `+n.toFixed(p)` или трюк с умножением и делением на 10^p .
- Дробные числа дают ошибку вычислений. При необходимости ее можно отсечь округлением до нужного знака.
- Случайные числа от 0 до 1 генерируются с помощью `Math.random()`, остальные — преобразованием из них.

Существуют и другие математические функции. Вы можете ознакомиться с ними в справочнике в разделах [Number](#) и [Math](#).



Задача: Бесконечный цикл по ошибке

Этот цикл - бесконечный. Почему?

Важность: 4

```
1 | var i = 0;  
2 | while(i != 10) {  
3 |     i += 0.2;  
4 | }
```

[Решение задачи "Бесконечный цикл по ошибке" »»](#)



Задача: Как получить дробную часть числа?

Напишите функцию `getDecimal(num)`, которая возвращает десятичную часть положительного числа:

Важность: 3

```
alert( getDecimal(12.5) ); // 0.5  
alert( getDecimal(6.25) ); // 0.25
```

Пожалуйста, не преобразуйте число к строке в процессе.

Ответьте на два вопроса:

- Точно ли работает ваша функция для большинства чисел?
- Если нет, но при этом есть информация, что дробная часть может быть не больше 6 знаков после запятой — можно ли поправить функцию, чтобы работала корректно?

[Решение задачи "Как получить дробную часть числа?" »»](#)



Задача: Формула Бине

Последовательность **чисел Фибоначчи** имеет формулу $F_n = F_{n-1} + F_{n-2}$. То есть, следующее число получается как сумма двух предыдущих.

Важность: 4

Первые два числа равны 1, затем 2(1+1), затем 3(1+2), 5(2+3) и так далее: 1, 1, 2, 3, 5, 8, 13, 21....

Код для их вычисления (из задачи **Числа Фибоначчи**):

```
1 | function fib(n){  
2 |     var a=1, b=0, x;  
3 |     for(i=0; i<n; i++) {  
4 |         x = a+b;
```

```

5     a = b
6     b = x;
7 }
8 return b;
9 }

```

Существует **формула Бине** , согласно которой F_n равно ближайшему целому для $\varphi^n/\sqrt{5}$, где $\varphi=(1+\sqrt{5})/2$ — золотое сечение.

Напишите функцию `fib(n)`, которая будет вычислять F_n , используя эту формулу. Проверьте её для значения F_{77} (должно получиться `fib(77) = 5527939700884757`).

Правильен ли полученный результат? Если нет, то почему?

[Решение задачи "Формула Бине" »»](#)



Задача: Случайное из интервала (0, b)

Напишите код для генерации случайного значения в диапазоне от 0 до `max`, не включая `max`.

Важность: 2

[Решение задачи "Случайное из интервала \(0, b\)" »»](#)



Задача: Случайное из интервала (a, b)

Напишите код для генерации случайного числа от `min` до `max`, не включая `max`.

Важность: 2

[Решение задачи "Случайное из интервала \(a, b\)" »»](#)



Задача: Случайное целое от `min` до `max`

Напишите код для генерации случайного **целого** числа между `min` и `max`, включая `min`, `max` как возможные значения.

Важность: 2

Любое число из интервала `min..max` должно иметь одинаковую вероятность.

[Решение задачи "Случайное целое от min до max" »»](#)

Объекты как ассоциативные массивы

Объекты в JavaScript являются «двуличными». Они сочетают в себе два важных функционала.

Первый — это ассоциативный массив: структура, пригодная для хранения любых данных. В этой главе мы рассмотрим использование объектов именно как массивов.

Второй — языковые возможности для объектно-ориентированного программирования. Эти возможности мы изучим в последующих разделах учебника.

Ассоциативные массивы

Ассоциативный массив — структура данных, в которой можно хранить любые данные в формате ключ-значение.

Её можно легко представить как шкаф с подписанными ящиками. Все данные хранятся в ящичках. По имени можно легко найти ящик и взять то значение, которое в нём лежит.

В отличие от реальных шкафов, в ассоциативный массив можно в любой момент добавить новые именованные «ящики» или удалить существующие. Далее мы увидим примеры, как это делается.

Кстати, в других языках программирования такую структуру данных также называют «словарь» и «хэш».

Создание объектов

Пустой объект («пустой шкаф») может быть создан одним из двух синтаксисов:

```
1. o = new Object();  
2. o = {}; // пустые фигурные скобки
```

Обычно все пользуются синтаксисом (2), т.к. он короче.

Операции с объектом

Объект может содержать в себе любые значения, которые называются *свойствами объекта*. Доступ к свойствам осуществляется по *имени свойства* (иногда говорят «по ключу»).

Например, создадим объект `person` для хранения информации о человеке:

```
var person = {}; // пока пустой
```

Основные операции с объектами — это:

1. Присвоение свойства по ключу.
2. Чтение свойства по ключу.
3. Удаление свойства по ключу.

Для обращения к свойствам используется запись «через точку», вида `объект.свойство`:

```
01 var person = {};  
02  
03 // 1. присвоение  
04 // при присвоении свойства в объекте автоматически  
    создаётся "ящик"  
05 // с именем "name" и в него записывается содержимое 'Вася'  
06 person.name = 'Вася';  
07  
08 person.age = 25; // запишем ещё одно свойство: с именем  
    'age' и значением 25  
09  
10 // 2. чтение  
11 alert(person.name + ': ' + person.age); // вывести  
    значения  
12  
13 // 3. удаление  
14 delete person.name; // удалить "ящик" с именем "name"  
    вместе со значением в нём
```

Следующая операция:

4. Проверка существования свойства с определенным ключом.

Например, есть объект `person`, и нужно проверить, существует ли в нем свойство `age`.

Для проверки существования есть оператор `in`. Его синтаксис:

"prop" in obj, причем имя свойства — в виде строки, например:

```
1 | var person = { };
2 |
3 | if ("age" in person) {
4 |     alert("Свойство age существует!");
5 | }
```

Впрочем, чаще используется другой способ — сравнение значения с undefined.

Дело в том, что в JavaScript можно обратиться к любому свойству объекта, даже если его нет. Ошибки не будет.

Но если свойство не существует, то вернется специальное значение undefined:

```
1 | var person = {};
2 |
3 | alert(person.lalala); // undefined, нет свойства с ключом lalala
```

Таким образом мы можем легко проверить существование свойства — получив его и сравнив с undefined:

```
1 | var person = { name: "Василий" };
2 |
3 | alert(person.lalala === undefined); // true, свойства нет
4 | alert(person.name === undefined); // false, свойство есть.
```

Разница между проверками in и === undefined

Сравнение с undefined не работает, если значение свойства равно undefined:

```
1 | var obj = {};
2 | obj.test = undefined; // добавили свойство в объект
3 |
4 | // проверим наличие свойств test и заведомо
   | отсутствующего blabla
5 | alert(obj.test === undefined); // true
6 | alert(obj.blabla === undefined); // true
```

.. А оператор in гарантирует точный результат:

```
1 | var obj = {};
```

```
2 | obj.test = undefined;  
3 |  
4 | alert( "test" in obj ); // true  
5 | alert( "blabla" in obj ); // false
```

Чтобы у вас работали обе проверки — не присваивайте `undefined`. В качестве значения, обозначающего неизвестность и неопределенность, используйте `null`.

Доступ через квадратные скобки

Существует альтернативный синтаксис работы со свойствами, использующий квадратные скобки объект `['свойство']`:

```
1 | var person = {};  
2 |  
3 | person['name'] = 'Вася'; // то же что и person.name  
4 |  
5 | alert(person['name']);  
6 |  
7 | delete person['name'];
```

В квадратные скобки можно передать переменную:

`obj[key]` использует значение переменной `key` в качестве имени свойства:

```
1 | var person = { age: 25 };  
2 | var key = 'age';  
3 |  
4 | alert(person[key]); // выведет person['age']
```

Записи `person['age']` и `person.age` идентичны. С другой стороны, если имя свойства хранится в переменной (`var key = "age"`), то единственный способ к нему обратиться — квадратные скобки `person[key]`.

Обращение через точку используется, если мы на этапе написания программы уже знаем название свойства. А если оно будет определено по ходу выполнения, например, введено посетителем и записано в переменную, то единственный выбор — квадратные скобки.



Ограничения на имя свойства

- На имя свойства при доступе «через точку» наложены синтаксические ограничения — примерно те же, что и на имя переменной. Оно не может начинаться с числа, содержать пробелы и т.п.
- При обращении через квадратные скобки можно использовать любую строку.

Например:

```
person['день рождения!'] = '18.04.1982';  
// запись через точку "person.день рождения! = ..." невозможна
```


В обоих случаях, **имя свойства обязано быть строкой**. Если использовано значение другого типа — JavaScript приведет его к строке автоматически.

Объявление со свойствами

Объект можно заполнить значениями при создании, указав их в фигурных скобках: { ключ1: значение1, ключ2: значение2, ... }.

Такой синтаксис называется *литеральным* (оригинал - *literal*), например:

```
menuSetup = {  
  width: 300,  
  height: 200,  
  title: "Menu"  
}
```



key	value
width	300
height	200
title	Menu

Следующие два фрагмента кода создают одинаковый объект:

```
01 var menuSetup = {  
02   width: 300,  
03   height: 200,  
04   title: "Menu"  
05 };  
06  
07 // то же самое, что:  
08  
09 var menuSetup = {};  
10 menuSetup.width = 300;  
11 menuSetup.height = 200;  
12 menuSetup.title = 'Menu';
```

Названия свойств можно перечислять в кавычках или без, если они

удовлетворяют ограничениям для имён переменных.

Например:

```
1 var menuSetup = {  
2   width: 300,  
3   'height': 200,  
4   "мама мыла раму": true  
5 };
```



Задача: Первый объект

Мини-задача на синтаксис объектов. Напишите код, Важность: 3 по строке на каждое действие.

1. Создайте пустой объект `user`.
2. Добавьте свойство `name` со значением `Вася`.
3. Добавьте свойство `surname` со значением `Петров`.
4. Поменяйте значение `name` на `Сергей`.
5. Удалите свойство `name` из объекта.

[Решение задачи "Первый объект" »»](#)

Вложенные объекты в объявлении

Значением свойства может быть объект:

```
01 var user = {  
02   name: "Таня",  
03   age: 25,  
04   size: {  
05     top: 90,  
06     middle: 60,  
07     bottom: 90  
08   }  
09 }  
10  
11 alert( user.name ) // "Таня"  
12  
13 alert( user.size.top ) // 90
```

Здесь значением свойства `size` является объект `{top: 90, middle: 60, bottom: 90 }`.





Вывод объекта, Firefox

Для целей отладки иногда хочется вывести объект целиком. В Firefox для этого существует нестандартный метод `toSource` .

```
1 var user = {
2   name: "Таня",
3   size: {
4     top: 90,
5     middle: 60
6   }
7 }
8
9 alert( user.toSource() ); // работает только в
  Firefox
```

В других браузерах его нет, но объект можно посмотреть через инструменты разработчика: отладчик или `console.log`.

Перебор свойств и значений

Для перебора всех свойств из объекта используется цикл по свойствам `for...in`. Это специальная синтаксическая конструкция, которая работает не так, как обычный цикл `for (i=0;i<n;i++)`.

Синтаксис:

```
for (key in obj) {
  /* ... делать что-то с obj[key] ... */
}
```

При этом в `key` будут последовательно записаны имена свойств.



Объявление переменной в цикле

```
for (var key in obj)
```

Переменную можно объявить прямо в цикле:

```
for (var key in menu) {
  // ...
}
```

Так иногда пишут для краткости кода.

Например:

```
01 var menu = {
02     width: 300,
03     height: 200,
04     title: "Menu"
05 };
06
07 for (var key in menu) {
08     // этот код будет вызван для каждого свойства
09     // ..и выведет имя свойства и его значение
10
11     alert("Ключ: " + key + " значение:" + menu[key]);
12 }
```

Конечно, вместо `key` может быть любое другое имя переменной.

Количество свойств в объекте

Одного метода, который вернул бы количество свойств нет.

Кросс-браузерный способ — это сделать цикл по свойствам и посчитать:

```
1 function getKeysCount(obj) {
2     var counter = 0;
3     for (var key in obj) {
4         counter++;
5     }
6     return counter;
7 }
```

А вот так выглядит функция проверки на «пустоту»:

```
1 function isEmpty(obj) {
2     for (var key in obj) {
3         return false; // если цикл хоть раз сработал, то
4     }                // объект не пустой => false
5     // дошли до этой строки - значит цикл не нашёл ни одного
6     // свойства => true
7     return true;
8 }
```

Порядок перебора свойств

Упорядочены ли свойства в объектах? В теории, т.е. по спецификации — нет. На практике, всё сложнее.

Браузеры придерживаются важного соглашения о порядке перебора свойств. Оно, хоть и не прописано в стандарте, но достаточно надежно, т.к. много существующего кода зависит от него.

- Браузеры IE<9, Firefox, Safari перебирают ключи в том же порядке, в котором свойства присваивались.
- Опега, современный IE, Chrome гарантируют сохранение порядка только для *строковых* ключей. Численные ключи сортируются и идут до строковых.

То есть, если свойства в объекте строковые, то такой объект упорядочен. Свойства будут перебираться в порядке их присвоения:

```
1 | var user = {  
2 |   name: "Вася",  
3 |   surname: "Петров"  
4 | };  
5 | user.age = 25;  
6 |  
7 | for (var prop in user) {  
8 |   alert(prop); // name, surname, age  
9 | }
```

Рассмотрим теперь случай, когда ключи не строковые. Например, следующий объект задает список опций для выбора страны:

```
1 | var codes = {  
2 |   // телефонные коды в формате "код страны": "название"  
3 |   "7": "Россия",  
4 |   "38": "Украина",  
5 |   // ...  
6 |   "1": "США"  
7 | };
```

Мы хотели бы предложить их посетителю в том же порядке, то есть «Россия» в начале, а «США» — в конце.

Однако, при переборе их через `for...in` некоторые браузеры (IE9+,

Chrome, Opera) выведут ключи не в том порядке, в котором они заданы, а в отсортированном порядке:

```
1 var codes = {
2   "7": "Россия",
3   "38": "Украина",
4   "1": "США"
5 };
6
7 for (var key in codes) {
8   alert(key + ": " + codes[key]); // 1, 7, 38 в IE9+,
  Chrome, Opera
9 }
```

Нарушение порядка возникло, потому что ключи численные. Чтобы его обойти, можно применить небольшой хак, который заключается в том, что все ключи искусственно делаются строчными. Например, добавим им дополнительный первый символ '+':

В этом случае браузер сочтет ключи строковыми и сохранит порядок перебора:

```
01 var codes = {
02   "+7": "Россия", // передаём 7 в виде строки, чтобы
   браузер сохранил порядок
03   "+38": "Украина",
04   "+1": "США"
05 };
06
07 for (var key in codes ) {
08   var value = codes[key];
09   var id = +key; // ..если нам нужно именно число,
   преобразуем: "+7" -> 7
10
11   alert( id + ": " + value ); // 7, 38, 1 во всех
   браузерах
12 }
```

Передача по ссылке

Обычные значения: строки, числа, булевы значения, null/undefined копируются «по значению».

Это означает, что если в переменной message хранится значение "Привет", и её скопировали phrase = message, то значение копируются, и у нас будут две переменные, каждая из которых хранит значение:

"Привет".

Объекты присваиваются и передаются «по ссылке».

В переменной хранится не сам объект, а ссылка на его место в памяти. Чтобы лучше понять это — сравним с обычными переменными.

Например:

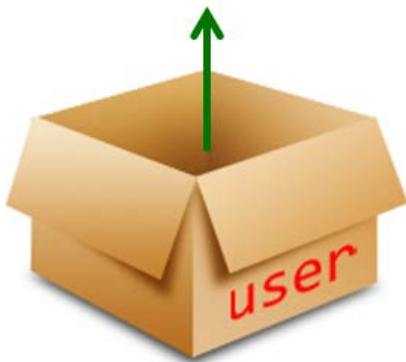
```
var message = "Привет"; // значение в переменной
```



А вот как выглядит переменная `user = { name: "Вася" }`.

Внимание: объект — вне переменной. В переменной — лишь ссылка («адрес места в памяти, где лежит объект»).

```
{  
  name: 'Вася'  
}
```

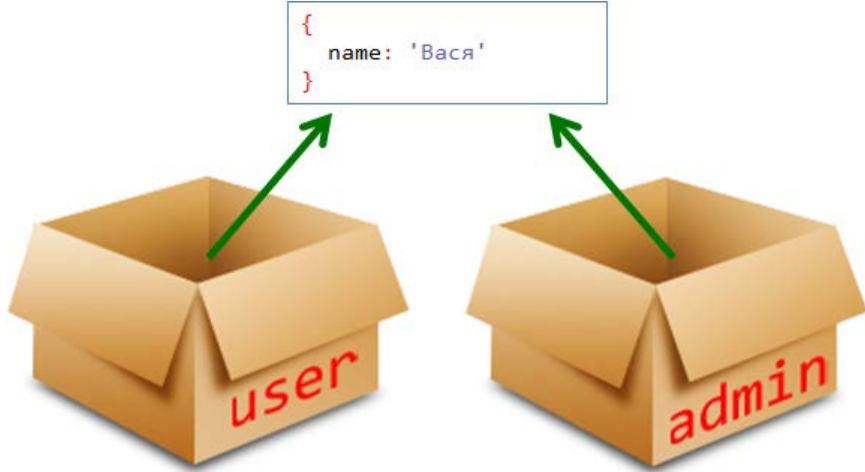


При копировании переменной с объектом — копируется эта ссылка, а объект по-прежнему остается в единственном экземпляре.

Получается, что несколько переменных *ссылаются* на один и тот же объект:

```
var user = { name: "Вася" }; // в переменной - ссылка
```

```
var admin = user; // скопировали ссылку
```



Так как объект всего один, то в какой бы переменной его не меняли — это отражается на других:

```
1 var user = { name: 'Вася' };  
2  
3 var admin = user;  
4  
5 admin.name = 'Петя'; // поменяли данные через admin  
6  
7 alert(user.name); // 'Петя', изменения видны в user
```



Переменная с объектом как «код» к сейфу с данными

Ещё одна аналогия: переменная, в которую присвоен объект, на самом деле хранит не сами данные, а код к сейфу, где они хранятся.

При передаче её в функцию, в локальные переменные копируется именно этот код, так что переменная может вносить в данные изменения.



«Настоящее» копирование объекта

Итак, при передаче объекта куда-либо, копируется лишь ссылка на него.

Чтобы скопировать сами данные, нужно достать их из объекта и скопировать на уровне примитивов.

Примерно так:

```
01 function clone(obj) {  
02     var obj2 = {};  
03
```

```
04 // если свойства могут быть объектами, то нужно
    перебирать и их
05     for(var key in obj) obj2[key] = obj[key];
06
07     return obj2;
08 }
09
10 var user = { name: 'Вася' }; // user - ссылка на
    объект
11
12 var admin = clone(user);
13
14 admin.name = 'Петя'; // поменяли данные в admin
15 alert(user.name); // 'Вася'
```



Задача: multiplyNumeric

Создайте функцию `multiplyNumeric`, которая получает объект и умножает все численные свойства на 2. Например:

Важность: 5

```
01 // до вызова
02 var menu = {
03     width: 200,
04     height: 300,
05     title: "My menu"
06 };
07
08 multiplyNumeric(menu);
09
10 // после вызова
11 menu = {
12     width: 400,
13     height: 600,
14     title: "My menu"
15 };
```

P.S. Для проверки на число используйте функцию:

```
function isNumeric(n) {
    return !isNaN(parseFloat(n)) && isFinite(n)
}
```

Решение задачи "multiplyNumeric" »»

Компактное представление объектов



Hardcore coders only

Эта секция относится ко внутреннему устройству структуры данных и требует специальных знаний. Она не обязательна к прочтению.

Объект, в таком виде как он описывается — занимает много места в памяти, например:

```
1 var user = {  
2   name: "Vasya",  
3   age: 25  
4 };
```

Здесь содержится информация о свойстве `name` и его строковом значении, а также о свойстве `age` и его численном значении. Представим, что таких объектов много.

Получится, что информация об именах свойств `name` и `age` дублируется в каждом объекте.

Чтобы избежать этого, браузеры используют специальное «компактное представление объектов».

При создании множества объектов одного и того же вида (с одинаковыми полями) интерпретатор запоминает, сколько у него полей и какие они, и хранит эти данные в отдельной структуре. А сам объект — в виде непрерывного массива данных.

Например, для объектов вида `{name: "Вася", age: 25}` будет создана структура, которая описывает данный вид объектов: «строка `name`, затем целое `age`», а сами объекты будут представлены в памяти данными: `Вася25`. Причём вместо строки обычно будет указатель на неё.

Браузер для доступа к свойству узнает из структуры, где оно лежит и перейдёт на нужную позицию в данных. Это очень быстро, и экономит память, когда на одно описание структуры приходится много однотипных объектов.

Что же происходит, если к объекту добавляется новое поле? Например:

```
user.admin = true;
```

В этом случае браузер смотрит, есть ли уже структура, под которую подходит такой объект («строка name, целое age, логическое admin»). Если нет — она создаётся. Изменившиеся данные объекта привязываются к ней.

Детали применения и реализации этого способа хранения варьируются от браузера к браузеру. Применяются дополнительные оптимизации.

Более подробно внутреннем устройстве типов вы можете узнать, например, из презентации [Know Your Engines \(Velocity 2011\)](#).

Итого

Объекты — это ассоциативные массивы с дополнительными возможностями:

→ Доступ к элементам осуществляется:

→ Напрямую по ключу `obj.prop = 5`

→ Через переменную, в которой хранится ключ:

```
var key = "prop";  
obj[key] = 5
```

→ Удаление ключей: `delete obj.name`.

→ Цикл по ключам: `for (key in obj)`, порядок перебора соответствует порядку объявления для строковых ключей, а для числовых - зависит от браузера.

→ Существование свойства может проверять оператор `in`:

`if ("prop" in obj)`, как правило, работает и просто сравнение `if (obj.prop !== undefined)`.

→ Переменная хранит не сам объект, а ссылку на него. Копирование такой переменной и передача её в функцию дублируют ссылку, но объект остаётся один.

См. также

→ [Wrong order in Object properties iteration](#)

Массивы с числовыми индексами

Массив с числовыми индексами - это коллекция данных, которая хранит сколько угодно значений, причем у каждого значения - свой уникальный номер.

Если переменная — это *коробка для данных*, то массив — это *шкаф с нумерованными ячейками*, в каждой из которых могут быть свои данные.

Например, при создании электронного магазина нужно хранить список товаров — для таких задач и придуман массив.

Объявление

Синтаксис для создания нового массива — квадратные скобки со списком элементов внутри.

Пустой массив:

```
var arr = [];
```

Массив `fruits` с тремя элементами:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];
```

Элементы нумеруются, начиная с нуля. Чтобы получить нужный элемент из массива — указывается его номер в квадратных скобках:

```
1 | var fruits = ["Яблоко", "Апельсин", "Слива"];
2 |
3 | alert(fruits[0]); // Яблоко
4 | alert(fruits[1]); // Апельсин
5 | alert(fruits[2]); // Слива
```

Элемент можно всегда заменить:

```
fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]
```

... Или добавить:

```
fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша",  
"Лимон"]
```

Общее число элементов, хранимых в массиве, содержится в его свойстве

length:

```
1 | var fruits = ["Яблоко", "Апельсин", "Груша"];
2 |
3 | alert(fruits.length); // 3
```

Через `alert` можно вывести и массив целиком. При этом его элементы будут перечислены через запятую:

```
1 | var fruits = ["Яблоко", "Апельсин", "Груша"];
2 |
3 | alert(fruits); // Яблоко,Апельсин,Груша
```

В массиве может храниться любое число элементов любого типа. В том числе, строки, числа, объекты и т.п.:

```
1 | // микс значений
2 | var arr = [ 1, 'Имя', { name: 'Петя' }, true ];
3 |
4 | // получить объект из массива и тут же -- его свойство
5 | alert( arr[2].name ); // Петя
```



Задача: Получить последний элемент массива

Как получить последний элемент из произвольного массива?

Важность: 5

У нас есть массив `goods`. Сколько в нем элементов — не знаем, но можем прочитать из `goods.length`. Напишите код для получения последнего элемента `goods`.

[Решение задачи "Получить последний элемент массива" »»](#)



Задача: Добавить новый элемент в массив

Как добавить элемент в конец произвольного массива?

Важность: 5

У нас есть массив `goods`. Напишите код для добавления в его конец значения «Компьютер».

[Решение задачи "Добавить новый элемент в массив" »»](#)

Методы `pop/push`, `shift/unshift`

Одно из применений массива — это **очередь**. В классическом программировании так называют упорядоченную коллекцию элементов, такую что элементы добавляются в конец, а обрабатываются — с начала. В реальной жизни эта структура данных встречается очень часто. Например, очередь сообщений, которые надо отослать.

Очень близка к очереди еще одна структура данных: **стек**. Это такая коллекция элементов, в которой новые элементы добавляются в конец и берутся с конца.

Например, стеком является колода карт, в которую новые карты кладутся сверху, и берутся — тоже сверху.

Для того, чтобы реализовывать эти структуры данных, и просто для более удобной работы с началом и концом массива существуют специальные методы.

Конец массива

pop

Удаляет *последний* элемент из массива и возвращает его:

```
1 | var fruits = ["Яблоко", "Апельсин", "Груша"];
2 |
3 | alert( fruits.pop() ); // удалили "Груша"
4 |
5 | alert(fruits); // Яблоко, Апельсин
```

push

Добавляет элемент *в конец* массива:

```
1 | var fruits = ["Яблоко", "Апельсин"];
2 |
3 | fruits.push("Груша");
4 |
5 | alert(fruits); // Яблоко, Апельсин, Груша
```

Является полным аналогом `fruits[fruits.length] =`

Начало массива

shift

Удаляет из массива *первый* элемент и возвращает его:

```

1 | var fruits = ["Яблоко", "Апельсин", "Груша"];
2 |
3 | alert( fruits.shift() ); // удалили Яблоко
4 |
5 | alert(fruits); // Апельсин, Груша

```

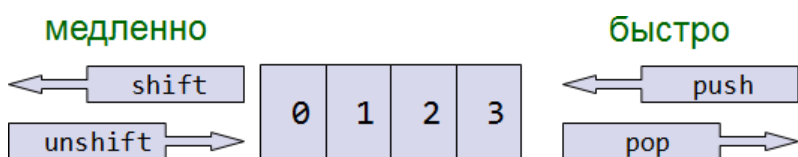
unshift

Добавляет элемент *в начало* массива:

```

1 | var fruits = ["Апельсин", "Груша"];
2 |
3 | fruits.unshift('Яблоко');
4 |
5 | alert(fruits); // Яблоко, Апельсин, Груша

```



Методы `push` и `unshift` могут добавлять сразу по несколько элементов:

```

1 | var fruits = ["Яблоко"];
2 |
3 | fruits.push("Апельсин", "Персик");
4 | fruits.unshift("Ананас", "Лимон");
5 |
6 | // результат: ["Ананас", "Лимон", "Яблоко", "Апельсин",
7 | "Персик"]
8 | alert(fruits);

```



Задача: Создание массива

Задача из 5 шагов-строк:

Важность: 5

1. Создайте массив `styles` с элементами «Джаз», «Блюз».
2. Добавьте в конец значение «Рок-н-Ролл»
3. Замените предпоследнее значение с конца на «Классика».
Код замены предпоследнего значения должен работать для массивов любой длины.
4. Удалите первое значение массива и выведите его `alert`.
5. Добавьте в начало значения «Рэп» и «Регги».

Массив в результате каждого шага:

- 1 Джаз, Блюз
- 2 Джаз, Блюз, Рок-н-Ролл
- 3 Джаз, Классика, Рок-н-Ролл
- 4 Классика, Рок-н-Ролл
- 5 Рэп, Регги, Классика, Рок-н-Ролл

[Решение задачи "Создание массива" »»](#)



Задача: Получить случайное значение из массива

Напишите код для вывода `alert` случайного значения из массива:

Важность: 3

```
var arr = ["Яблоко", "Апельсин", "Груша", "Лимон"];
```

P.S. Код для генерации случайного целого от `min` to `max` включительно:

```
var rand = min + Math.floor( Math.random() * (max+1-min) );
```

[Решение задачи "Получить случайное значение из массива" »»](#)



Задача: Создайте калькулятор для введённых значений

Напишите код, который:

Важность: 4

- Запрашивает по очереди значения при помощи `prompt` и сохраняет их в массиве.
- Заканчивает ввод, как только посетитель введёт пустую строку, не число или нажмёт «Отмена».
- Выводит сумму всех значений массива

На демо-страничке

<http://learn.javascript.ru/files/tutorial/intro/array/calculator.html>

показан результат.

[Решение задачи "Создайте калькулятор для введённых значений" »»](#)

Внутреннее устройство массива

Массив — это объект, где в качестве ключей выбраны цифры, с дополнительными методами и свойством `length`.

Так как это объект, то в функцию он передаётся по ссылке:

```
01 function eat(arr) {  
02   arr.pop();  
03 }  
04  
05 var arr = ["нам", "не", "страшен", "серый", "волк"]  
06  
07 alert(arr.length); // 5  
08 eat(arr);  
09 eat(arr);  
10 alert(arr.length); // 3, в функцию массив не скопирован, а  
   передана ссылка
```

Ещё одно следствие — можно присваивать в массив любые свойства.

Например:

```
1 var fruits = []; // создать массив  
2  
3 fruits[99999] = 5; // присвоить свойство с любым номером  
4  
5 fruits.age = 25; // назначить свойство со строковым именем
```

.. Но массивы для того и придуманы в JavaScript, чтобы удобно работать именно с *упорядоченными, нумерованными данными*. Для этого в них существуют специальные методы и свойство `length`.

Как правило, нет причин использовать массив как обычный объект, хотя технически это и возможно.



Вывод массива с «дырами»

Если в массиве есть пропущенные индексы, то при выводе в большинстве браузеров появляются «лишние» запятые, например:

```
1 var a = [];  
2 a[0] = 0;  
3 a[5] = 5;  
4  
5 alert(a); // 0,,,,,5
```

Эти запятые появляются потому, что алгоритм вывода массива идёт от 0 до `arr.length` и выводит всё через запятую. Отсутствие значений даёт несколько запятых подряд.

Влияние на быстродействие

Методы `push/pop` выполняются быстро, а `shift/unshift` — медленно.

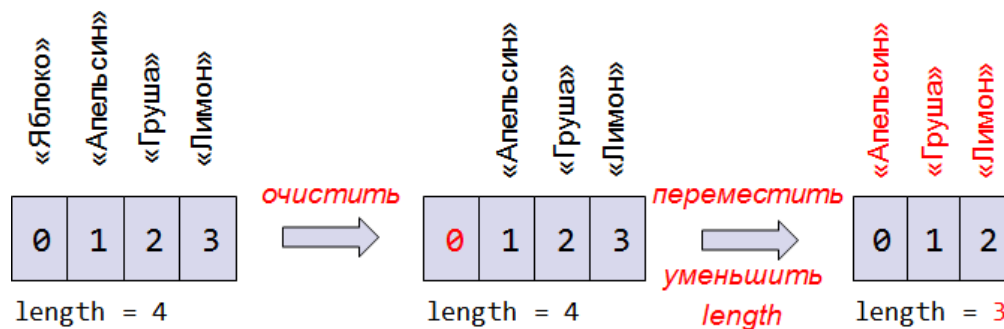
Чтобы понять, почему работать с концом массива быстрее, чем с его началом, разберём происходящее подробнее.

Операция `shift` выполняет два действия:

1. Удалить элемент в начале.
2. Обновить внутреннее свойство `length`.

При этом, так как все элементы находятся в своих ячейках, просто очистить ячейку с номером 0 недостаточно. Нужно еще и переместить все ячейки на 1 вниз (красным на рисунке подсвечены изменения):

`fruits.shift();` // убрать 1 элемент с начала

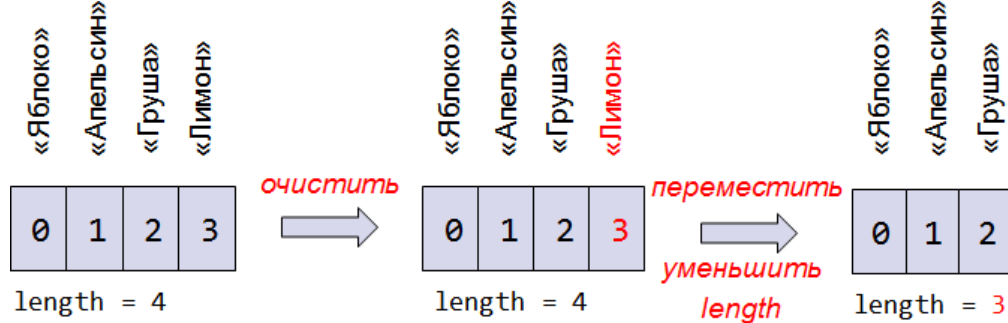


Чем больше элементов в массиве, тем дольше их перемещать.

Аналогично работает `unshift`: чтобы добавить элемент в начало массива, нужно сначала перенести все существующие.

У методов `push/pop` таких проблем нет. Для того, чтобы удалить элемент, метод `pop` очищает ячейку и укорачивает `length`.

`fruits.pop();` // убрать 1 элемент с конца



Аналогично работает push.

Перебор элементов

Для перебора элементов обычно используется цикл:

```
1 var arr = ["Яблоко", "Апельсин", "Груша"];
2
3 for (var i=0; i<arr.length; i++) {
4     alert( arr[i] );
5 }
```



Не используйте `for..in` для массивов

Так как массив является объектом, то возможен и вариант `for..in`:

```
1 var arr = ["Яблоко", "Апельсин", "Груша"];
2
3 for (var key in arr) {
4     alert( arr[key] ); // Яблоко, Апельсин, Груша
5 }
```

Недостатки этого способа:

→ Цикл `for..in` выведет *все свойства* объекта, а не только цифровые.

В браузере, при работе с объектами страницы, встречаются коллекции элементов, которые по виду как массивы, но имеют дополнительные нецифровые свойства, которые будут видны в цикле `for..in`.

Бывают и библиотеки, которые предоставляют такие коллекции. Например jQuery. С виду — массив, но есть дополнительные свойства. Для перебора только цифровых свойств нужен цикл `for(var i=0; i<arr.length...)`

→ Цикл `for (var i=0; i<arr.length; i++)` в современных браузерах выполняется в 10-100 раз быстрее. Казалось бы, по виду он сложнее, но браузер особым образом оптимизирует такие циклы.

Если кратко: цикл `for(var i=0; i<arr.length...)` надёжнее и быстрее.

Особенности работы `length`

Встроенные методы для работы с массивом автоматически обновляют его длину `length`.

Длина `length` — не количество элементов массива, а последний индекс + 1. Так уж оно устроено.

Это легко увидеть на следующем примере:

```
1 | var arr = [];  
2 | arr[1000] = true;  
3 |  
4 | alert(arr.length); // 1001
```

Вообще, если у вас элементы массива нумеруются случайно или с большими пропусками, то стоит подумать о том, чтобы использовать обычный объект.

Массивы предназначены именно для работы с непрерывной упорядоченной коллекцией элементов.

Используем `length` для укорачивания массива

Обычно нам не нужно самостоятельно менять `length`... Но есть один фокус, который можно повернуть.

При уменьшении `length` массив укорачивается. Причем этот процесс необратимый, т.е. даже если потом вернуть `length` обратно — значения не восстановятся:

```
1 | var arr = [1, 2, 3, 4, 5];  
2 |  
3 | arr.length = 2; // укоротить до 2 элементов
```

```
4 | alert(arr); // [1, 2]
5 |
6 | arr.length = 5; // вернуть length обратно, как было
7 | alert(arr[3]); // undefined: значения не вернулись
```

Самый простой способ очистить массив — это `arr.length=0`.

Создание вызовом `new Array`

`new Array()`

Существует еще один синтаксис для создания массива:

```
var arr = new Array("Яблоко", "Груша", "и т.п.");
```

Он редко используется, т.к. квадратные скобки `[]` короче.

Кроме того, у него есть одна особенность. Обычно

`new Array(элементы, ...)` создаёт массив из данных элементов, но если у него один аргумент, и это число — то он создает массив без элементов, но с заданной длиной.

```
1 | var arr = new Array(2,3); // создает массив [2, 3]
2 |
3 | arr = new Array(2); // создаст массив [2] ?
4 |
5 | alert(arr[0]); // нет! у нас массив без элементов, длины 2
```

Что же такое этот «массив без элементов, но с длиной»? Как такое возможно?

Оказывается, очень даже возможно и соответствует объекту `{length: 2}`. Получившийся массив ведёт себя так, как будто его элементы равны `undefined`.

Это может быть неожиданным сюрпризом, поэтому обычно используют квадратные скобки.

Многомерные массивы

Массивы в JavaScript могут содержать в качестве элементов другие массивы. Это можно использовать для создания многомерных массивов,

например матриц:

```
1 var matrix = [  
2   [1, 2, 3],  
3   [4, 5, 6],  
4   [7, 8, 9]  
5 ];  
6  
7 alert(matrix[1][1]); // центральный элемент
```

Внутреннее представление массивов



Hardcore coders only

Эта секция относится ко внутреннему устройству структуры данных и требует специальных знаний. Она не обязательна к прочтению.

Числовые массивы, согласно спецификации, являются объектами, в которые добавили ряд свойств, методов и автоматическую длину `length`. Но внутри они, как правило, устроены по-другому.

Современные интерпретаторы стараются оптимизировать их и хранить в памяти не в виде хэш-таблицы, а в виде непрерывной области памяти, так же как в языке C. Операции с массивами также оптимизируются, особенно если массив хранит только один тип данных, например только числа. Порождаемый набор инструкций для процессора получается очень эффективным.

Как правило, у интерпретатора это получается, но при этом программист не должен мешать.

В частности:

- Не ставить массиву произвольные свойства, такие как `arr.test = 5`. То есть, работать именно как с массивом, а не как с объектом.
- Заполнять массив непрерывно. Как только браузер встречает необычное поведение массива, например устанавливается значение `arr[0]`, а потом сразу `arr[1000]`, то он начинает работать с ним, как с обычным объектом. Как правило, это влечёт преобразование его в хэш-таблицу.

Если следовать этим принципам, то массивы будут занимать меньше памяти и быстрее работать.

Итого

Массивы существуют для работы с упорядоченным набором элементов.

Объявление:

```
1 // предпочтительное
2 var arr = [ элемент1, элемент2... ];
3
4 // new Array
5 var arr = new Array( элемент1, элемент2... );
```

При этом `new Array(число)` создаёт массив заданной длины, без элементов. Чтобы избежать ошибок, предпочтителен первый синтаксис.

Свойство `length` — длина массива. Если точнее, то последний индекс массива плюс 1. Если её уменьшить вручную, то массив укоротится. Если `length` больше реального количества элементов, то отсутствующие элементы равны `undefined`.

Массив можно использовать как очередь или стек.

Операции с концом массива:

- `arr.push(элемент1, элемент2...)` добавляет элементы в конец.
- `var elem = arr.pop()` удаляет и возвращает последний элемент.

Операции с началом массива:

- `arr.unshift(элемент1, элемент2...)` добавляет элементы в начало.
- `var elem = arr.shift()` удаляет и возвращает первый элемент.

Эти операции перенумеровывают все элементы, поэтому работают медленно.

В следующей главе мы рассмотрим другие методы для работы с массивами.



Задача: Чему равен элемент массива?

Что выведет этот код?

Важность: 3

```
1 var arr = [1,2,3];
2
3 var a = arr;
4 a[0] = 5;
5
6 alert(arr[0]);
7 alert(a[0]);
```

[Решение задачи "Чему равен элемент массива?" »»](#)



Задача: Поиск в массиве

Создайте функцию `find(arr, value)`, которая ищет `value` в массиве `arr` и возвращает его номер, если найдено, или `-1`, если не найдено. Важность: 3

Например:

```
1 arr = [ "test", 2, 1.5, false ];
2
3 find(arr, "test"); // 0
4 find(arr, 2); // 1
5 find(arr, 1.5); // 2
6
7 find(arr, 0); // -1
```

[Решение задачи "Поиск в массиве" »»](#)



Задача: Фильтр диапазона

Создайте функцию `filterRange(arr, a, b)`, которая принимает массив чисел `arr` и возвращает новый массив, который содержит только числа из `arr` из диапазона от `a` до `b`. То есть, проверка имеет вид $a \leq arr[i] \leq b$. Важность: 3
Функция не должна менять `arr`.

Пример работы:

```
1 var arr = [5, 4, 3, 8, 0];
2
3 var filtered = filterRange(arr, 3, 5);
4 // теперь filtered = [5, 4, 3]
5 // arr не изменился
```

[Решение задачи "Фильтр диапазона" »»](#)



Задача: Решето Эратосфена

Целое число, большее 1, называется *простым*, если оно не делится нацело ни на какое другое, кроме себя и 1. Важность: 3

Древний алгоритм «Решето Эратосфена» для поиска всех простых чисел до n выглядит так:

1. Создать список последовательных чисел от 2 до n :
2, 3, 4, ..., n .
2. Пусть $p=2$, это первое простое число.
3. Зачеркнуть все последующие числа в списке с разницей в p , т.е. $2p$, $3p$, $4p$ и т.д. В случае $p=2$ это будут 4, 6, 8, ...
4. Поменять значение p на первое незачеркнутое число после p .
5. Повторить шаги 3-4 пока $p^2 < n$.
6. Все оставшиеся незачеркнутыми числа - простые.

Посмотрите также [анимацию алгоритма](#).

Реализуйте «Решето Эратосфена» в JavaScript. Найдите все простые числа до 100 и выведите их сумму.

[Решение задачи "Решето Эратосфена" »»](#)



Задача: Подмассив наибольшей суммы

На входе массив чисел, например:

`arr = [1, -2, 3, 4, -9, 6].`

Важность: 2

Задача — найти непрерывный подмассив `arr`, сумма элементов которого максимальна.

Ваша функция должна возвращать только эту сумму.

Например:

```
1 getMaxSubSum([-1, 2, 3, -9]) = 5 (сумма
   выделенных)
2 getMaxSubSum([2, -1, 2, 3, -9]) = 6
3 getMaxSubSum([-1, 2, 3, -9, 11]) = 11
4 getMaxSubSum([-2, -1, 1, 2]) = 3
```

```
5 | getMaxSubSum([100, -9, 2, -3, 5]) = 100
6 | getMaxSubSum([1, 2, 3]) = 6 (неотрицательные - берем всех)
```

Если все элементы отрицательные, то не берём ни одного элемента и считаем сумму равной нулю:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Постарайтесь придумать решение, которое работает за $O(n^2)$, а лучше за $O(n)$ операций.

[Решение задачи "Подмассив наибольшей суммы" »»](#)

См. также

→ [Array in Mozilla manual](#)

Массивы: методы

В этой главе мы рассмотрим встроенные методы массивов JavaScript.

`Object.keys(obj)`

В JavaScript есть встроенный метод `Object.keys(obj)`, который возвращает ключи объекта в виде массива. Он поддерживается везде, кроме IE<9:

```
1 | var user = {
2 |   name: "Петя",
3 |   age: 30
4 | }
5 |
6 | var keys = Object.keys(user);
7 |
8 | alert(keys); // name, age
```

В более старых браузерах аналогом будет цикл:

```
var keys = [];
for(var key in user) keys.push(key);
```



Задача: Оставить уникальные элементы массива

Пусть `strings` — массив строк.

Важность: 3

Напишите функцию `unique(strings)`, которая возвращает массив, содержащий только уникальные элементы `arr`.

Например:

```
1 function unique(arr) {  
2   /* ваш код */  
3 }  
4  
5 var strings = ["кришна", "кришна", "хапе", "хапе",  
6   "хапе", "хапе", "кришна", "кришна", "8-()"];  
7  
8 alert( unique(strings) ); // кришна, хапе, 8-()
```

Решение задачи "Оставить уникальные элементы массива" »»

Метод `split`

Ситуация из реальной жизни. Мы пишем сервис отсылки сообщений и посетитель вводит имена тех, кому его отправить:

Маша, Петя, Марина, Василий.... Но нам-то гораздо удобнее работать с массивом имен, чем с одной строкой.

К счастью, есть метод `split(s)`, который позволяет превратить строку в массив, разбив ее по разделителю `s`. В примере ниже таким разделителем является строка из запятой и пробела.

```
1 var names = 'Маша, Петя, Марина, Василий';  
2  
3 var arr = names.split(', ');  
4  
5 for (var i=0; i<arr.length; i++) {  
6   alert('Вам сообщение ' + arr[i]);  
7 }
```



Второй аргумент `split`

У метода `split` есть необязательный второй аргумент — ограничение на количество элементов в массиве. Если их больше, чем указано — остаток массива будет отброшен:

```
1 alert( "a,b,c,d".split(',', 2) ); // a,b
```

На практике он используется редко.



Разбивка по буквам

Вызов `str.split('')` разобьёт строку на буквы:

```
1 | var str = "тест";
2 |
3 | alert( str.split('') ); // т,е,с,т
```

Метод join

Вызов `arr.join(str)` делает в точности противоположное `split`. Он берет массив и склеивает его в строку, используя `str` как разделитель.

Например:

```
1 | var arr = ['Маша', 'Петя', 'Марина', 'Василий'];
2 |
3 | var str = arr.join(';');
4 |
5 | alert(str); // Маша;Петя;Марина;Василий
```



`new Array` + `join` = Повторение строки

Код для повторения строки 3 раза:

```
1 | alert( new Array(4).join("ля") ); // ляляля
```

Как видно, `new Array(4)` делает массив без элементов длины 4, который `join` объединяет в строку, вставляя *между его элементами* строку "ля".

В результате, так как элементы пусты, получается повторение строки. Такой вот небольшой трюк.



Задача: Добавить класс в строку

В объекте есть свойство `className`, которое содержит список «классов» - слов, разделенных пробелом:

Важность: 5

```
var obj = {  
  className: 'open menu'  
}
```

Создайте функцию `addClass(obj, cls)`, которая добавляет в список класс `cls`, но только если его там еще нет:

```
1 addClass(obj, 'new'); // obj.className='open menu  
  new'  
2 addClass(obj, 'open'); // без изменений (класс уже  
  существует)  
3 addClass(obj, 'me'); // obj.className='open menu  
  new me'  
4  
5 alert(obj.className); // "open menu new me"
```

P.S. Ваша функция не должна добавлять лишних пробелов.

[Решение задачи "Добавить класс в строку" »»](#)



Задача: Перевести текст вида `border-left-width` в `borderLeftWidth`

Напишите функцию `camelize(str)`, которая преобразует строки вида «`my-short-string`» в «`myShortString`».

Важность: 3

То есть, дефисы удаляются, а все слова после них получают заглавную букву.

Например:

```
camelize("background-color") == 'backgroundColor';  
camelize("list-style-image") == 'listStyleImage';
```

Такая функция полезна при работе с CSS.

P.S. Вам пригодятся методы строк `charAt`, `split` и `toUpperCase`.

[Решение задачи "Перевести текст вида border-left-width в borderLeftWidth" »»](#)

Удаление из массива

Так как массивы являются объектами, то для удаления ключа можно воспользоваться обычным `delete`:


```
1 var arr = ["Я", "иду", "домой"];
2
3 delete arr[1]; // значение с индексом 1 удалено
4
5 // теперь arr = ["Я", undefined, "домой"];
6 alert(arr[1]); // undefined
```

Да, элемент удален из массива, но не так, как нам этого хочется. Образовалась «дырка».

Это потому, что оператор `delete` удаляет пару «ключ-значение». Это - все, что он делает. Обычно же при удалении из массива мы хотим, чтобы оставшиеся элементы сдвинулись и заполнили образовавшийся промежуток.

Поэтому для удаления используются специальные методы - из начала `shift`, из конца - `pop`, а из середины - `splice`, с которым мы сейчас познакомимся.

Метод `splice`

Метод `splice` - это универсальный раскладной нож для работы с массивами. Умеет все: удалять элементы, вставлять элементы, заменять элементы - по очереди и одновременно.

Его синтаксис:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

Удалить `deleteCount` элементов, начиная с номера `index`, а затем вставить `elem1, ..., elemN` на их место.

Посмотрим примеры.

```
1 var arr = ["Я", "изучаю", "JavaScript"];
2
3 arr.splice(1, 1); // начиная с позиции 1, удалить 1
  элемент
4
5 alert(arr); // осталось ["Я", "JavaScript"]
```

Здесь продемонстрировано, как **использовать `splice` для удаления одного элемента****. Следующие за удаленным элементы сдвигаются, чтобы заполнить его место.

```
1 var arr = ["Я", "изучаю", "JavaScript"];
2
3 arr.splice(0, 1); // удалить 1 элемент, начиная с позиции
4 0
5 alert( arr[0] ); // "изучаю" стал первым элементом
```

Следующий пример показывает, как *заменять элементы*:

```
1 var arr = ["Я", "сейчас", "изучаю", "JavaScript"];
2
3 // удалить 3 первых элемента и добавить другие вместо них
4 arr.splice(0, 3, "Мы", "изучаем")
5
6 alert( arr ) // теперь ["Мы", "изучаем", "JavaScript"]
```

Метод `splice` возвращает массив из удаленных элементов:

```
1 var arr = ["Я", "сейчас", "изучаю", "JavaScript"];
2
3 // удалить 2 первых элемента
4 var removed = arr.splice(0, 2);
5
6 alert( removed ); // "Я", "сейчас" <-- array of removed
  elements
```

Метод `splice` также может вставлять элементы без удаления, для этого достаточно установить `deleteCount` в 0:

```
1 var arr = ["Я", "изучаю", "JavaScript"];
2
3 // с позиции 2
4 // удалить 0
5 // вставить "сложный", "язык"
6 arr.splice(2, 0, "сложный", "язык");
7
8 alert(arr); // "Я", "изучаю", "сложный", "язык",
  "JavaScript"
```

Допускается использование отрицательного номера позиции, которая в этом случае отсчитывается с конца:

```
1 var arr = [1, 2, 5]
2
3 // начиная с позиции индексом -1 (предпоследний элемент)
4 // удалить 0 элементов,
5 // затем вставить числа 3 и 4
```

```
6 arr.splice(-1, 0, 3, 4);  
7  
8 alert(arr); // результат: 1,2,3,4,5
```



Задача: Функция removeClass

У объекта есть свойство `className`, которое хранит список «классов» - слов, разделенных пробелами: Важность: 5

```
var obj = {  
  className: 'open menu'  
}
```

Напишите функцию `removeClass(obj, cls)`, которая удаляет класс `cls`, если он есть:

```
removeClass(obj, 'open'); // obj.className='menu'  
removeClass(obj, 'blabla'); // без изменений (нет  
такого класса)
```

P.S. Дополнительное усложнение. Функция должна корректно обрабатывать дублирование класса в строке:

```
obj = { className: 'my menu menu' };  
removeClass(obj, 'menu');  
alert(obj.className); // 'my'
```

Лишних пробелов после функции образовываться не должно.

[Решение задачи "Функция removeClass" »»](#)



Задача: Фильтрация массива "на месте"

Создайте функцию Важность: 4
`filterRangeInPlace(arr, a, b)`, которая получает массив с числами `arr` и удаляет из него все числа вне диапазона `a..b`.

То есть, проверка имеет вид $a \leq arr[i] \leq b$. Функция должна менять сам массив и ничего не возвращать.

Например:

```
1 arr = [5, 3, 8, 1];  
2  
3 filterRangeInPlace(arr, 1, 4); // удалены числа вне
```

диапазона 1..4

4

5 alert(arr); // массив изменился: остались [3, 1]

Решение задачи "Фильтрация массива "на месте"»»

Метод slice

Метод slice(begin, end) копирует участок массива от begin до end, не включая end. Исходный массив при этом не меняется.

Например:

```
1 var arr = ["Почему", "надо", "учить", "JavaScript"];
2
3 var arr2 = arr.slice(1,3); // элементы 1, 2 (не включая 3)
4
5 alert(arr2); // надо, учить
```

Аргументы ведут себя так же, как и в строковом slice:

→ Если не указать end — копирование будет до конца массива:

```
1 var arr = ["Почему", "надо", "учить", "JavaScript"];
2
3 alert( arr.slice(1) ); // взять все элементы, начиная с
  номера 1
```

→ Можно использовать отрицательные индексы, они отсчитываются с конца:

```
var arr2 = arr.slice(-2); // копировать от 2го элемента с
конца и дальше
```

→ Если вообще не указать аргументов — скопируется весь массив:

```
var fullCopy = arr.slice();
```

Сортировка, метод sort(fn)

Метод sort() сортирует массив *на месте*. Например:

```
1 var arr = [ 1, 2, 15 ];
2
3 arr.sort();
```

```
4  
5 alert( arr ); // 1, 15, 2
```

Не заметили ничего странного в этом примере?

Порядок стал 1, 15, 2. Это произошло потому, что **sort сортирует, преобразуя элементы к строке**. Поэтому и порядок у них строковый, ведь "2" > "15".

Свой порядок сортировки

Внутренняя реализация метода arr.sort(fn) умеет сортировать любые массивы, если указать функцию fn от двух элементов, которая умеет сравнивать их.

Если эту функцию не указать, то элементы сортируются как строки.

Например, укажем эту функцию явно, отсортируем элементы массива как числа:

```
01 function compareNumeric(a, b) {  
02     if (a > b) return 1;  
03     if (a < b) return -1;  
04 }  
05  
06 var arr = [ 1, 2, 15 ];  
07  
08 arr.sort(compareNumeric);  
09  
10 alert(arr); // 1, 2, 15
```

Обратите внимание, мы передаём в sort() именно саму функцию compareNumeric, без вызова через скобки. Был бы ошибкой следующий код:

```
arr.sort( compareNumeric() ); // не работает
```

К функции, передаваемой sort, есть всего одно требование.

Алгоритм сортировки, встроенный в JavaScript, будет передавать ей для сравнения элементы массива. Она должна возвращать:

- Положительное значение, если $a > b$,
- Отрицательное значение, если $a < b$,
- Если равны — не важно, что возвращать, их взаимный порядок не

имеет значения:



Алгоритм сортировки

В методе `sort`, внутри самого интерпретатора JavaScript, реализован универсальный алгоритм сортировки. Как правило, это "**быстрая сортировка**", дополнительно оптимизированная для небольших массивов.

Ему совершенно неважно, что сортировать — строки, числа, яблоки с апельсинами или посетителей. Это и не должно быть важно, алгоритм сортировки просто сортирует абстрактные «элементы массива». Всё, что ему нужно о них знать — как эти элементы сравнивать между собой.

Для этого мы передаём в `sort` функцию сравнения. А там уже алгоритм решает, что с чем сравнивать, чтобы отсортировать побыстрее.

Кстати, те значения, с которыми `sort` вызывает функцию сравнения, можно увидеть, если вставить в неё `alert`:

```
1 | [1, -2, 15, 2, 0, 8].sort(function(a, b) {  
2 |     alert(a + " <> " + b);  
3 | });
```

Функцию `compareNumeric` для сравнения элементов-чисел можно упростить до одной строчки. Как?

Функция должна возвращать положительное число, если `a > b`, отрицательное, если наоборот, и, например, `0`, если числа равны.

Всем этим требованиям удовлетворяет функция:

```
function compareNumeric(a, b) {  
    return a - b;  
}
```



Задача: Сортировать в обратном порядке

Как отсортировать массив чисел в обратном

Важность: 5

порядке?

```
1 | var arr = [ 5, 2, 1, -10, 8];  
2 |  
3 | // отсортируйте?  
4 |  
5 | alert(arr); // 8, 5, 2, 1, -10
```

[Решение задачи "Сортировать в обратном порядке" »»](#)



Задача: Скопировать и отсортировать массив

Есть массив строк `arr`. Создайте массив `arrSorted` — из тех же элементов, но отсортированный. Важность: 5

Исходный массив не должен меняться.

```
1 | var arr = [ "HTML", "JavaScript", "CSS" ];  
2 |  
3 | // ... ваш код ...  
4 |  
5 | alert(arrSorted); // CSS, HTML, JavaScript  
6 | alert(arr); // HTML, JavaScript, CSS (без  
  | изменений)
```

Постарайтесь сделать код как можно короче.

[Решение задачи "Скопировать и отсортировать массив" »»](#)



Задача: Случайный порядок в массиве

Используйте функцию `sort` для того, чтобы «перетрясти» элементы массива в случайном порядке. Важность: 3

```
1 | var arr = [1, 2, 3, 4, 5];  
2 |  
3 | arr.sort(ваша функция);  
4 |  
5 | alert(arr); // элементы в случайном порядке,  
  | например [3,5,1,2,4]
```

[Решение задачи "Случайный порядок в массиве" »»](#)



Задача: Сортировка объектов

Напишите код, который отсортирует массив объектов `people` по полю `age`.

Важность: 5

Например:

```
01 var vasya = { name: "Вася", age: 23 };
02 var masha = { name: "Маша", age: 18 };
03 var vovochka = { name: "Вовочка", age: 6 };
04
05 var people = [ vasya , masha , vovochka ];
06
07 ... ваш код ...
08
09 // теперь people: [vovochka, masha, vasya]
10 alert(people[0].age) // 6
```

Выведите список имён в массиве после сортировки.

[Решение задачи "Сортировка объектов" »»](#)

reverse

Метод `arr.reverse()` меняет порядок элементов в массиве на обратный.

```
1 var arr = [1,2,3];
2 arr.reverse();
3
4 alert(arr); // 3,2,1
```



Задача: Вывести односвязный список

Односвязный список — это структура данных, которая состоит из *элементов*, каждый из которых хранит ссылку на следующий. Последний элемент может не иметь ссылки, либо она равна `null`.

Важность: 5

Например, объект ниже задаёт односвязный список, в `next` хранится ссылка на следующий элемент:

```
01 var list = {
02   value: 1,
03   next: {
04     value: 2,
05     next: {
06       value: 3,
07       next: {
```

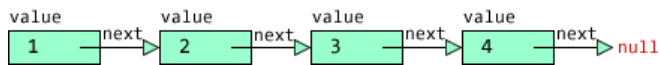


```

08         value: 4,
09         next: null
10     }
11 }
12 }
13 };

```

Графическое представление этого списка:



Альтернативный способ создания:

```

1  var list = { value: 1 };
2  list.next = { value: 2 };
3  list.next.next = { value: 3 };
4  list.next.next.next = { value: 4 };

```

Такая структура данных интересна тем, что можно очень быстро разбить список на части, объединить списки, удалить или добавить элемент в любое место, включая начало. При использовании массива такие действия требуют обширных перенумерований.

Задачи:

1. Напишите функцию `printList(list)`, которая выводит элементы списка по очереди.
2. Напишите функцию `printReverseList(list)`, которая выводит элементы списка в обратном порядке, используя рекурсию.
Для списка выше она должна вывести 4,3,2,1.
3. Напишите `printReverseList(list)` без использования рекурсии. Какой вариант быстрее? Почему?

[Решение задачи "Вывести односвязный список" »»](#)

concat

Метод `arr.concat(value1, value2, ... valueN)` создаёт новый массив, в который копируются элементы из `arr`, а также `value1, value2, ... valueN`.

Например:

```
1 | var arr = [1,2];
2 | var newArr = arr.concat(3,4);
3 |
4 | alert(newArr); // 1,2,3,4
```

Если `value` — массив, то `concat` добавляет его элементы.

Например:

```
1 | var arr = [1,2];
2 | var newArr = arr.concat( [3,4], 5); // то же самое, что
   | arr.concat(3,4,5)
3 |
4 | alert(newArr); // 1,2,3,4,5
```

indexOf/lastIndexOf

Эти методы не поддерживаются в IE<9. Для их поддержки подключите библиотеку [ES5-shim](#).

Метод `arr.indexOf(searchElement[, fromIndex])` возвращает номер элемента `searchElement` в массиве `arr` или `-1`, если его нет.

Поиск начинается с номера `fromIndex`, если он указан. Если нет — с начала массива.

Для поиска используется строгое сравнение `===`.

Например:

```
1 | var arr = [ 1, 0, false ];
2 |
3 | alert( arr.indexOf(0) ); // 1
4 | alert( arr.indexOf(false) ); // 2
5 | alert( arr.indexOf(null) ); // -1
```

Как вы могли заметить, по синтаксису он полностью аналогичен методу `indexOf` для строк .

Метод `arr.lastIndexOf(searchElement[, fromIndex])` ищет справа-налево: с конца массива или с номера `fromIndex`, если он указан.

 Методы `indexOf/lastIndexOf` осуществляют поиск перебором

Если нужно проверить, существует ли значение в массиве — его нужно перебрать. Только так. Внутренняя реализация `indexOf/lastIndexOf` осуществляет полный перебор, аналогичный циклу `for` по массиву. Чем длиннее массив, тем дольше он будет работать.



Коллекция уникальных элементов

Рассмотрим задачу — есть коллекция строк, и нужно быстро проверять: есть ли в ней какой-то элемент. Массив для этого не подходит из-за медленного `indexOf`. Но подходит объект! Доступ к свойству объекта осуществляется очень быстро, так что можно сделать все элементы ключами объекта и проверять, есть ли уже такой ключ.

Например, организуем такую проверку для коллекции строк `"div"`, `"a"` и `"form"`:

```
1  var store = { }; // объект для коллекции
2
3  var items = ["div", "a", "form"];
4
5  for(var i=0; i<items.length; i++) {
6      var key = items[i]; // для каждого элемента
7      store[ key ] = true; // значение здесь не важно
8  }
```

Теперь для проверки, есть ли ключ `key`, достаточно выполнить `if (store[key])`. Если есть — можно использовать значение, если нет — добавить.

Такое решение работает только со строками, но применимо к любым элементам, для которых можно вычислить строковый «уникальный ключ».



Задача: Отфильтровать анаграммы

Анаграммы — слова, состоящие из одинакового количества одинаковых букв, но в разном порядке.

Например:

Важность: 4

воз - зов

киборг - гробик

корсет - костер - сектор

Напишите функцию `aclean(arr)`, которая возвращает массив слов, очищенный от анаграмм.

Например:

```
var arr = ["воз", "киборг", "корсет", "зов", "гробик",  
"костер", "сектор"];
```

```
alert( aclean(arr) ); // "воз,киборг,корсет" или  
"зов,гробик,сектор"
```

Из каждой группы анаграмм должно остаться только одно слово, не важно какое.

Решение задачи "Отфильтровать анаграммы" »»

Итого

Методы:

- `push/pop`, `shift/unshift`, `splice` — для добавления и удаления элементов.
- `join/split` — для преобразования строки в массив и обратно.
- `sort` — для сортировки массива. Если не передать функцию сравнения — сортирует элементы как строки.
- `reverse` — меняет порядок элементов на обратный.
- `concat` — объединяет массивы.
- `indexOf/lastIndexOf` — возвращают позицию элемента в массиве (не поддерживается в IE<9).

Изученных нами методов достаточно в 95% случаях, но существуют и другие. Для знакомства с ними рекомендуется заглянуть в справочник [Array](#) и [Array в Mozilla Developer Network](#) .

См. также

- [Array in Mozilla manual](#)

Дата и Время

Для работы с датой и временем в JavaScript используются объекты `Date` .

Создание

Для создания нового объекта типа `Date` используется один из синтаксисов:

`new Date()`

Создает объект `Date` с текущей датой и временем:

```
1 | var now = new Date();  
2 | alert(now);
```

`new Date(milliseconds)`

Создает объект `Date`, значение которого равно количеству миллисекунд (1/1000 секунды), прошедших с 1 января 1970 года GMT+0.

```
1 | // 24 часа после 01.01.1970 GMT+0  
2 | var Jan02_1970 = new Date(3600*24*1000);  
3 | alert( Jan02_1970 );
```

`new Date(datestring)`

Если единственный аргумент - строка, используется вызов `Date.parse` для ее разбора.

`new Date(year, month, date, hours, minutes, seconds, ms)`

Дату можно создать, используя компоненты в местной временной зоне. Для этого формата обязательны только первые два аргумента. Отсутствующие параметры, начиная с `hours` считаются равными нулю, а `date` — единице.

Заметим, что год `year` должен быть из 4 цифр, а отсчет месяцев `month` начинается с нуля 0. Например:

```
new Date(2011, 0, 1) // 1 января 2011, 00:00:00 в местной  
временной зоне  
new Date(2011, 0) // то же самое, date по умолчанию равно 1  
new Date(2011, 0, 1, 0, 0, 0, 0); // то же самое
```

Дата задана с точностью до миллисекунд:

```
1 | var d = new Date(2011, 0, 1, 2, 3, 4, 567);  
2 | alert(d); // 1.01.2011, 02:03:04.567
```



Задача: Создайте дату

Создайте объект Date для даты: 20 февраля 2012 года, 3 часа 12 минут.

Важность: 5

Временная зона — местная. Выведите его на экран.

[Решение задачи "Создайте дату" »»](#)

Получение компонент даты

Для доступа к компонентам даты-времени объекта Date используются следующие методы:

getFullYear()

Получить год(из 4 цифр)

getMonth()

Получить месяц, от 0 до 11.

getDate()

Получить число месяца, от 1 до 31.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Получить соответствующие компоненты.



Устаревший getYear()

Некоторые браузеры реализуют нестандартный метод `getYear()`. Где-то он возвращает только две цифры из года, где-то четыре. Так или иначе, этот метод отсутствует в стандарте JavaScript. Не используйте его. Для получения года есть `getFullYear()`.

Дополнительно можно получить день недели:

getDay()

Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом от 0(воскресенье) до 6(суббота).

Все методы, указанные выше, возвращают результат для местной временной зоны.

Существуют также UTC-варианты этих методов, возвращающие день, месяц, год и т.п. для зоны GMT+0 (UTC): `getUTCFullYear()`, `getUTCMonth()`, `getUTCDay()`. То есть, сразу после "get" вставляется "UTC".

Если ваше локальное время сдвинуто относительно UTC, то следующий код покажет разные часы:

```
1 var date = new Date();
2
3 alert( date.getHours() ); // час в вашей зоне для даты
  date
4 alert( date.getUTCHours() ); // час в зоне GMT+0 для даты
  date
```

Кроме описанных выше, существуют два специальных метода без UTC-варианта:

`getTime()`

Возвращает число миллисекунд, прошедших с 01.01.1970 00:00:00 UTC. Это то же число, которое используется в конструкторе `new Date(milliseconds)`.

`getTimezoneOffset()`

Возвращает разницу между местным и UTC-временем, в минутах.

```
1 alert( new Date().getTimezoneOffset() ); // Для GMT-1
  выведет 60
```



Задача: Имя дня недели

Создайте функцию `getWeekDay(date)`, которая выводит текущий день недели в коротком формате 'пн', 'вт', ... 'вс'.

Важность: 5

Например:

```
var date = new Date(2012,0,3); // 3 января 2012
alert( getWeekDay(date) );      // Должно вывести 'вт'
```

[Решение задачи "Имя дня недели" »»](#)



Задача: День недели в европейской нумерации

Напишите функцию, `getLocalDay(date)` которая возвращает день недели для даты `date`.

Важность: 5

День нужно вернуть в европейской нумерации, т.е. понедельник имеет номер 1, вторник номер 2, ..., воскресенье - номер 7.

```
var date = new Date(2012, 0, 3); // 3 янв 2012
alert( getLocalDay(date) );      // вторник, выведет 2
```

Решение задачи "День недели в европейской нумерации" »»

Установка компонент даты

Следующие методы позволяют устанавливать компоненты даты и времени:

- `setFullYear(year [, month, date])`
- `setMonth(month [, date])`
- `setDate(date)`
- `setHours(hour [, min, sec, ms])`
- `setMinutes(min [, sec, ms])`
- `setSeconds(sec [, ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (устанавливает всю дату по миллисекундам с 01.01.1970 UTC)

Все они, кроме `setTime()`, обладают также UTC-вариантом, например: `setUTCHours()`.

Как видно, некоторые методы могут устанавливать несколько компонент даты одновременно, в частности, `setHours`. При этом если какая-то компонента не указана, она не меняется. Например:

```
1 var today = new Date;
2
3 today.setHours(0);
4 alert( today ); // сегодня, но час изменён на 0
```



```
5  
6 today.setHours(0, 0, 0, 0);  
7 alert (today ); // сегодня, ровно 00:00:00.
```

Автоисправление даты

Автоисправление — очень удобное свойство объектов Date. Оно заключается в том, что можно устанавливать заведомо некорректные компоненты (например 32 января), а объект сам себя поправит.

```
1 var d = new Date(2013, 0, 32); // 32 января 2013 ?!?  
2 alert(d); // ... это 1 февраля 2013!
```

Неправильные компоненты даты автоматически распределяются по остальным.

Например, нужно увеличить на 2 дня дату «28 февраля 2011». Может быть так, что это будет 2 марта, а может быть и 1 марта, если год високосный. Но нам обо всем этом думать не нужно. Просто прибавляем два дня. Остальное сделает Date:

```
1 var d = new Date(2011, 1, 28);  
2 d.setDate( d.getDate() + 2 );  
3  
4 alert(d); // 2 марта, 2011
```


Также это используют для получения даты, отдаленной от имеющейся на нужный промежуток времени. Например, получим дату на 70 секунд большую текущей:

```
1 var d = new Date();  
2 d.setSeconds( d.getSeconds()+70);  
3  
4 alert(d); // выведет корректную дату
```

Можно установить и нулевые, и даже отрицательные компоненты. Например:

```
1 var d = new Date;  
2  
3 d.setDate(1); // поставить первое число месяца  
4 alert(d);  
5  
6 d.setDate(0); // нулевого числа нет, будет последнее число  
  предыдущего месяца
```

```
7 | alert(d);  
1 | var d = new Date;  
2 |  
3 | d.setDate(-1); // предпоследнее число предыдущего месяца  
4 | alert(d);
```


 **Задача:** День 100 дней назад

Какое число месяца было 100 дней назад? Какой день недели?

Используйте JavaScript, чтобы вывести эту информацию. День недели выводите двухбуквенном виде, т.е. одно значение из (пн, вт, ср, ..., вс).

Важность: 4

[Решение задачи "День 100 дней назад" »»](#)

 **Задача:** Последний день месяца?

Напишите функцию `getLastDayInMonth(year, month)`, которая возвращает последний день месяца.

Параметры:

- `year` — 4-значный год, например 2012.
- `month` — месяц от 0 до 11.

Например, `getLastDayInMonth(2012, 1) = 29` (високосный год, февраль).

Важность: 5

[Решение задачи "Последний день месяца?" »»](#)

Преобразование к числу, разность дат

Когда объект `Date` используется в числовом контексте, он преобразуется в количество миллисекунд:

```
1 | alert( +new Date ) // +date то же самое, что:  
   | +date.valueOf()
```

Важный побочный эффект: даты можно вычитать, результат вычитания объектов `Date` — их временная разница, в

миллисекундах.

Это используют для измерения времени:

```
01 var start = new Date; // засекали время
02
03 // что-то сделать
04 for (var i=0; i<100000; i++) {
05     var doSomething = i*i*i;
06 }
07
08 var end = new Date; // конец измерения
09
10 alert("Цикл занял " + (end-start) + " ms");
```



Задача: Сколько секунд - до завтра?

Напишите код, который выводит:

Важность: 5

1. Сколько секунд прошло с начала сегодняшнего дня.
2. Сколько осталось до конца дня.

Скрипт должен работать в любой день, т.е. в нём не должно быть конкретного значения сегодняшней даты.

[Решение задачи "Сколько секунд - до завтра?" >>>](#)

Бенчмаркинг

Допустим, у нас есть несколько вариантов решения задачи, каждый описан функцией.

Как узнать, какой быстрее?

Для примера возьмем две функции, которые округляют число:

```
1 function floorMath(x) {
2     return Math.floor(x);
3 }
4
5 function floorXor(x) {
6     return x^0; // побитовое исключающее ИЛИ (XOR) всегда
7     округляет число
8 }
```

Чтобы померять, какая из них быстрее, нельзя запустить один раз `floorMath`, один раз `floorXor` и замерить разницу. Одноразовый запуск

ненадежен, любая мини-помеха исказит результат.

Для правильного бенчмаркинга функция запускается много раз, чтобы сам тест занял существенное время. Это сведет влияние помех к минимуму. Сложную функцию можно запускать 100 раз, простую — 1000 раз...

Померяем, какая из функций округления быстрее:

```
01 function floorMath(x) { return Math.floor(x); }
02 function floorXor(x) { return x^0; }
03
04 function bench(f) {
05     var d = new Date();
06     for (var i=0.5; i<1000000; i++) f(i);
07     return new Date() - d;
08 }
09
10 alert('Время floorMath: ' + bench(floorMath) + 'мс');
11 alert('Время floorXor: ' + bench(floorXor) + 'мс');
```

В зависимости от браузера, может быть быстрее как floorXor так и floorMath.

Многократное повторение функции — обязательно.

Иначе измерения сильно подвержены помехам. Ещё более точные результаты можно получить, если сам пакет тестов, т.е два теста в данном случае, тоже прогоняется много раз.

Форматирование

Встроенные в Date методы форматирования используются редко, и преимущественно, для отладки.

toString(), toDateString(), toTimeString()

Возвращают стандартное строчное представление, не указанное в стандарте, а зависящее от браузера. Единственное требование - читаемость человеком. Метод toString возвращает дату целиком, toDateString() и toTimeString() - только дату и время соответственно.

```
1 var d = new Date();
2
3 alert( d.toString() ); // вывод, похожий на 'Wed Jan 26
  2011 16:40:50 GMT+0300'
```

toLocaleString(), toLocaleDateString(), toLocaleTimeString()

То же самое, но строка должна быть с учетом локальных настроек и языка посетителя.

```
1 | var d = new Date();  
2 |  
3 | alert( d.toLocaleString() ); // дата на языке посетителя
```

toUTCString()

То же самое, что toString(), но дата в зоне UTC.

toISOString()

Возвращает дату в формате ISO Детали формата будут далее.

Поддерживается современными браузерами, не поддерживается IE<9.

```
1 | var d = new Date();  
2 |  
3 | alert( d.toISOString() ); // вывод, похожий на '2011-01-  
   | 26T13:51:50.417Z'
```

Встроенные методы форматирования Date не допускают указание собственного формата.

Поэтому, как правило, любой вывод, кроме отладочного, форматируется своей, а не встроенной функцией.



Задача: Вывести текущую дату

Напишите функцию `formatDate(date)`, которая выводит дату `date` в формате `дд.мм.гг`:

Важность: 3

Например:

```
var d = new Date(2011, 0, 30); // 30 января 2011  
alert( formatDate(d) ); // '30.01.11'
```

P.S. Обратите внимание, ведущие нули должны присутствовать, то есть 1 января 2011 должно быть 01.01.11, а не 1.1.11.

[Решение задачи "Вывести текущую дату" »»](#)



Задача: Относительное форматирование даты

Напишите функцию `formatDate(date)`, которая

Важность: 4

форматирует дату date так:

- Если со времени date прошло менее секунды, то возвращает "только что".
- Иначе если со времени date прошло менее минуты, то "n сек. назад".
- Иначе если прошло меньше часа, то "m мин. назад".
- Иначе полная дата в формате "дд.мм.гг чч:мм".

Например:

```
1 function formatDate(date) { /* ваш код */ }
2
3 alert( formatDate( new Date(new Date - 1) ) ); //
  "только что"
4
5 alert( formatDate( new Date(new Date - 30*1000) )
  ); // "30 сек. назад"
6
7 alert( formatDate( new Date(new Date - 5*60*1000) )
  ); // "5 мин. назад"
8
9 alert( formatDate( new Date(new Date - 86400*1000)
  ) ); // вчерашняя дата в формате "дд.мм.гг чч:мм"
```

[Решение задачи "Относительное форматирование даты" »»](#)

Разбор строки, Date.parse

Все современные браузеры, включая IE9+, понимают даты в упрощённом формате ISO 8601 Extended.

Этот формат выглядит так: YYYY-MM-DDTHH:mm:ss.sssZ. Для разделения даты и времени в нем используется символ 'T'. Часть 'Z' обозначает (необязательную) временную зону — она может отсутствовать, тогда зона UTC, либо может быть символ z — тоже UTC, или зона в формате +-hh:mm.

Также возможны упрощенные варианты, к примеру:

YYYY

YYYY-MM

YYYY-MM-DD

Метод `Date.parse(str)` разбирает строку `str` в таком формате и возвращает соответствующее ей количество миллисекунд. Если это невозможно, `Date.parse` возвращает `NaN`.

На момент написания некоторые браузеры (Safari) воспринимали формат без `'Z'` как дату в локальной таймзоне (по стандарту UTC), поэтому пример ниже в них работает некорректно:

```
1 var msNoZone = Date.parse('2012-01-26T13:51:50.417'); //  
  без зоны, значит UTC  
2  
3 alert(msNoZone); // 1327571510417 (число миллисекунд)  
4  
5 var msZ = Date.parse('2012-01-26T13:51:50.417z'); // зона  
  z означает UTC  
6 alert(msZ == msNoZone); // true, если браузер правильный
```

С таймзоной `-07:00 GMT` в конце все современные браузеры работают правильно:

```
1 var ms = Date.parse('2012-01-26T13:51:50.417-07:00');  
2  
3 alert(ms); // 1327611110417 (число миллисекунд)
```



Формат дат для IE8-

До появления спецификации EcmaScript 5 формат не был стандартизован, и браузеры, включая IE8-, имели свои собственные форматы дат. Частично, эти форматы пересекаются.

Например, код ниже работает везде, включая старые IE:

```
1 var ms = Date.parse("January 26, 2011 13:51:50");  
2  
3 alert(ms);
```

Вы также можете почитать о старых форматах IE в документации к методу [MSDN Date.parse](#) .

Конечно же, сейчас лучше использовать современный формат, если цель — современные браузеры, а если дополнительно нужны IE8-, то либо передавать даты через миллисекунды, а не строки, либо добавить библиотеку типа [es5-shim](#), которая

добавит `Date.parse` в старые IE.

Итого

- Дата и время представлены в JavaScript одним объектом: `Date` .
Создать «только время» при этом нельзя, оно должно быть с датой.
Список методов `Date` вы можете найти в справочнике `Date` или выше.
- Объект `Date` удобен тем, что автокорректируется. Благодаря этому легко сдвигать даты.
- Объекты `Date` можно вычитать, результатом будет разница в мс.

Преобразование типов

Система преобразования типов в JavaScript очень проста, но отличается от других языков. Поэтому она часто служит «камнем преткновения» для приходящих из других языков программистов.

Всего есть три преобразования:

1. Строковое преобразование.
2. Числовое преобразование.
3. Преобразование к логическому значению.

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, его производит функция `alert`.

```
1 | var a = true;  
2 |  
3 | alert(a); // "true"
```

Можно также осуществить преобразование явным вызовом `String(val)`:

```
1 | alert( String(null) === "null" ); // true
```

Также для явного преобразования применяется оператор `+`, у которого один из аргументов строка. В этом случае он приводит к строке и другой

аргумент, например:

```
1 | alert( true + "test" ); // "truetest"
2 | alert( "123" + undefined ); // "123undefined"
```

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений `===`, `!==`).

Для преобразования к числу в явном виде можно вызвать `Number(val)`, либо, что короче, поставить перед выражением оператор `+`:

```
var a = +"123\n"; // 123
var a = Number("123\n"); // 123, тот же эффект
```

Значение Преобразуется в...

undefined	NaN
null	0
true / false	1 / 0
Строка	Пробелы по краям обрезаются. Далее, если остаётся пустая строка, то 0. Из непустой строки «считывается» число, при ошибке результат: NaN.

Примеры:

→ Логические значения:

```
1 | alert( +true ); // 1
2 | alert( +false ); // 0
```

→ Сравнение разных типов — значит численное преобразование:

```
1 | alert( "\n0\n" == 0 ); // true
```

При этом строка `"\n0\n"` преобразуется к числу — начальные и конечные пробелы игнорируются, получается 0.

→ Ещё пример:

```
1 | alert( "\n" == false );
```

Здесь сравнение `==` снова приводит обе части к числу. И слева и справа получается `0`.

По аналогичной причине верно равенство `"1" == true`.

Специальные значения

Посмотрим на поведение специальных значений более внимательно.

Интуитивно, значения `null/undefined` ассоциируются с нулём, но при преобразованиях ведут себя иначе.

Специальные значения преобразуются к числу так:

Значение

Преобразуется в...

<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>

Это преобразование осуществляется при арифметических операциях и сравнениях `>` `>=` `<` `<=`, но не при проверке равенства `==`. Алгоритм проверки равенства для этих значений в спецификации прописан отдельно (пункт [11.9.3](#)). В нём считается, что `null` и `undefined` равны `"=="` между собой, но эти значения не равны никакому другому значению.

Это ведёт к забавным последствиям.

Например, `null` не подчиняется законам математики — он «больше либо равен нулю»: `null>=0`, но не больше и не равен:

```
1 alert(null >= 0); // true, т.к. null преобразуется к 0
2 alert(null > 0); // false (не больше), т.к. null
  преобразуется к 0
3 alert(null == 0); // false (и не равен!), т.к. ==
  рассматривает null особо.
```

Значение `undefined` вообще вне сравнений:

```
1 alert(undefined > 0); // false, т.к. undefined -> NaN
2 alert(undefined == 0); // false, т.к. это undefined (без
  преобразования)
3 alert(undefined < 0); // false, т.к. undefined -> NaN
```

Для более очевидной работы кода и во избежание ошибок лучше не давать специальным значениям участвовать в сравнениях

`>` `>=` `<` `<=`.

Используйте в таких случаях переменные-числа или приводите к числу явно.

Логическое преобразование

Преобразование к `true/false` происходит в логическом контексте, таком как `if(obj)`, `while(obj)` и при применении логических операторов.

Все значения, которые интуитивно «пусты», становятся `false`. Их несколько: `0`, пустая строка, `null`, `undefined` и `NaN`.

Остальное, в том числе и любые объекты — `true`.

Полная таблица преобразований:

Значение	Преобразуется в...
<code>undefined</code> , <code>null</code>	<code>false</code>
Числа	Все <code>true</code> , кроме <code>0</code> , <code>NaN</code> — <code>false</code> .
Строки	Все <code>true</code> , кроме пустой строки <code>"</code> — <code>false</code>
Объекты	Всегда <code>true</code>

Для явного преобразования используется двойное логическое отрицание `!!value` или вызов `Boolean(value)`.

 Обратите внимание: строка `"0"` становится `true`

В отличие от многих языков программирования (например PHP), `"0"` в JavaScript является `true`, как и строка из пробелов:

```
1 alert( !! "0" ); // true
2 alert( !! " " ); // любые непустые строки, даже из
  пробелов - true!
```

 Пустой объект или массив тоже `true`

Также, в отличие от ряда других языков программирования, пустой объект `{}` или массив `[]` являются `true`:

```
1 if ( {} ) {
2   alert("{} -> true");
3 }
4
5 if ( [] ) {
```

```
6 | alert("[ ] -> true");  
7 | }
```

Логическое преобразование интересно тем, как оно сочетается с численным.

Два значения могут быть равны, но одно из них в логическом контексте `true`, другое — `false`.

Например, равенства в следующем примере верны, так как происходит численное преобразование:

```
1 | alert( 0 == "\n0\n" ); // true  
2 | alert( false == " " ); // true
```

...А в логическом контексте левая часть даст `false`, правая — `true`:

```
1 | if ( "\n0\n" ) {  
2 |     alert("true, совсем не как 0!");  
3 | }
```

С точки зрения преобразования типов в JavaScript это совершенно нормально. При равенстве — численное преобразование, а в `if` — логическое, только и всего.

Итого

В JavaScript есть три преобразования:

1. Строковое: происходит обычно при выводе.
2. Численное: его делают математические операторы и функции, а также сравнения и проверки равенства, кроме строгих `===` и `!==`.
3. Логическое: происходит по таблице.

Сравнение не осуществляет преобразование типов в следующих случаях:

- При сравнении объектов. Две переменные, которые являются объектами равны только, когда ссылаются на один и тот же объект.
- При сравнении двух строк. Там отдельный алгоритм сравнения. А вот если хоть один операнд — не строка, то значения будут приведены:

`true > "000"` станет `1 > 0`.

→ При проверке равенства с `null` и `undefined`. Они равны друг другу, но не равны чему бы то ни было ещё, этот случай прописан особо в спецификации.

Код для явного преобразования типов:

Преобразование к числу

`+value` или `Number(value)`

Преобразование к строке

`'' + value` или `String(value)`

Преобразование к логическому значению

`!!value` или `Boolean(value)`



Задача: Вопросник по преобразованиям, для примитивов

Подумайте, какой результат будет у выражений ниже. Тут не только преобразования типов. Когда закончите — сверьтесь с решением.

Важность: 5

```
01  "" + 1 + 0
02  "" - 1 + 0
03  true + false
04  6 / "3"
05  "2" * "3"
06  4 + 5 + "px"
07  "$" + 4 + 5
08  "4" - 2
09  "4px" - 2
10  7 / 0
11  parseInt("09")
12  "  -9\n" + 5
13  "  -9\n" - 5
14  5 && 2
15  2 && 5
16  5 || 0
17  0 || 5
18  null + 1
19  undefined + 1
```

Решение задачи "Вопросник по преобразованиям, для примитивов" »»

Решения задач



Решение задачи: Сделать первый символ заглавным

Мы не можем просто заменить первый символ, т.к. строки в JavaScript неизменяемы.

Единственный способ - пересоздать строку на основе существующей, но с заглавным первым символом:

```
01 function ucFirst(str) {  
02     var newStr = str.charAt(0).toUpperCase();  
03  
04     for(var i=1; i<str.length; i++) {  
05         newStr += str.charAt(i);  
06     }  
07  
08     return newStr;  
09 }  
10  
11 alert( ucFirst("вася") );
```

P.S. Возможны и другие решения, использующие метод `str.slice` и `str.replace`.



Решение задачи: Проверьте спам

Метод `indexOf` ищет совпадение с учетом регистра. То есть, в строке `'xXx'` он не найдет `'XXX'`.

Для проверки приведем к нижнему регистру и строку `str` и то, что будем искать:

```
1 function checkSpam(str) {  
2     str = str.toLowerCase();  
3  
4     return str.indexOf('viagra') >= 0 ||  
5         str.indexOf('xxx') >= 0;  
6 }
```

Полное решение:

<http://learn.javascript.ru/play/tutorial/intro/checkSpam.html>.



Решение задачи: Усечение строки

Так как окончательная длина строки должна быть `maxlength`, то нужно её обрезать немного короче, чтобы дать место для троеточия.

```
01 function truncate(str, maxlength) {  
02     if (str.length > maxlength) {  
03         return str.slice(0, maxlength - 3) + '...';  
04         // итоговая длина равна maxlength  
05     }  
06  
07     return str;  
08 }  
09  
10 alert(truncate("Вот, что мне хотелось бы сказать на  
    эту тему:", 20));  
11 alert(truncate("Всем привет!", 20));
```

Ещё лучшим вариантом будет использование вместо трёх точек специального символа «троеточие»: `... (…)`, тогда можно отрезать один символ.

```
01 function truncate(str, maxlength) {  
02     if (str.length > maxlength) {  
03         return str.slice(0, maxlength - 1) + '...';  
04     }  
05  
06     return str;  
07 }  
08  
09 alert(truncate("Вот, что мне хотелось бы сказать на  
    эту тему:", 20));  
10 alert(truncate("Всем привет!", 20));
```

Можно было бы написать этот код ещё короче:

```
1 function truncate(str, maxlength) {  
2     return (str.length > maxlength) ?  
3         str.slice(0, maxlength - 1) + '...' : str;  
4 }  
5  
6 alert(truncate("Вот, что мне хотелось бы сказать на  
    эту тему:", 20));  
7 alert(truncate("Всем привет!", 20));
```



Решение задачи: Интерфейс sum

<http://learn.javascript.ru/play/tutorial/intro/sum.html>



Решение задачи: Почему 6.35.toFixed(1) == 6.3?

Во внутреннем двоичном представлении 6.35 является бесконечной двоичной дробью. Хранится она с потерей точности.. А впрочем, посмотрим сами:

```
1 | alert( 6.35.toFixed(20) ); // 6.34999999999999964473
```

Интерпретатор видит число как 6.34..., поэтому и округляет вниз.



Решение задачи: Сложение цен

Есть два основных подхода.

1. Можно хранить сами цены в «копейках» (центах и т.п.). Тогда они всегда будут целые и проблема исчезнет. Но при показе и при обмене данными нужно будет это учитывать и не забывать делить на 100.
2. При операциях, когда необходимо получить окончательный результат — округлять до 2го знака после запятой. Все, что дальше — ошибка округления:

```
1 | var price1 = 0.1, price2 = 0.2;  
2 | alert( +(price1 + price2).toFixed(2) );
```



Решение задачи: Бесконечный цикл по ошибке

Потому что `i` никогда не станет равным 10.

Запустите, чтобы увидеть *реальные* значения `i`:

```
1 | var i = 0;  
2 | while(i < 11) {  
3 |   i += 0.2;  
4 |   if (i>9.8 && i<10.2) alert(i);
```



```
5 | }
```

Ни одно из них в точности не равно 10.



Решение задачи: Как получить дробную часть числа?

Функция

Функция может быть такой:

```
1 | function getDecimal(num) {  
2 |     return num - Math.floor(num);  
3 | }  
4 |  
5 | alert( getDecimal(12.5) ); // 0.5  
6 | alert( getDecimal(6.25) ); // 0.25
```

...Или, гораздо проще, такой:

```
function getDecimal(num) {  
    return num % 1;  
}
```

Числа

Обычно функция работает неправильно из-за неточных алгоритмов работы с дробями.

Например:

```
1 | alert( 1.2 % 1 ); // 0.19999999999999996  
2 | alert( 1.3 % 1 ); // 0.30000000000000004  
3 | alert( 1.4 % 1 ); // 0.3999999999999999
```

Исправление

Можно добавить округление `toFixed`, которое отбросит лишние знаки:

```
1 | function getDecimal(num) {  
2 |     return +(num % 1).toFixed(6);  
3 | }  
4 |  
5 | alert( getDecimal(1.2) ); // 0.2  
6 | alert( getDecimal(1.3) ); // 0.3  
7 | alert( getDecimal(1.4) ); // 0.4
```



Решение задачи: Формула Бине

Вычисление по следствию из формулы Бине:

```
1 function fib(n) {  
2   var phi = (1 + Math.sqrt(5)) / 2;  
3   return Math.round( Math.pow(phi, n) / Math.sqrt(5)  
4   );  
5 }  
6 alert( fib(2) ); // 1, верно  
7 alert( fib(8) ); // 21, верно  
8 alert( fib(77)); // 5527939700884755 !=  
   5527939700884757, неверно!
```

Обратите внимание — при вычислении используется округление `Math.round`, т.к. нужно именно *ближайшее целое*.

Результат вычисления F_{77} неправильный!

Он отличается от вычисленного другим способом. Причина — в ошибках округления, ведь $\sqrt{5}$ — бесконечная дробь.

Ошибки округления при вычислениях множатся и, в итоге, дают расхождение.



Решение задачи: Случайное из интервала (0, b)

Сгенерируем значение в диапазоне `0..1` и умножим на `max`:

```
1 var max = 10;  
2  
3 alert( Math.random()*max );
```



Решение задачи: Случайное из интервала (a, b)

Сгенерируем значение из интервала `0..max-min`, а затем сдвинем на `min`:

```
1 var min=5, max = 10;  
2  
3 alert( min + Math.random()*(max-min) );
```



Решение задачи: Случайное целое от min до max

Очевидное неверное решение (round)

Самый простой, но неверный способ - это сгенерировать значение в интервале `min..max` и округлить его `Math.round`, вот так:

```
var rand = min + Math.random()*(max-min)
rand = Math.round(rand);
```

Оно работает. Но при этом вероятность получить крайние значения `min` и `max` будет в два раза меньше, чем любые другие. Например, давайте найдем значения между 1 и 3 этим способом:

```
// случайное число от 1 до 3, не включая 3
var rand = 1 + Math.random()*(3-1)
```

Вызов `Math.round()` округлит значения следующим образом:

значения из диапазона 1	...	1.499+	станут 1
значения из диапазона 1.5	...	2.499+	станут 2
значения из диапазона 2.5	...	2.999+	станут 3

Отсюда уже видно, что в 1 (как и 3) попадает диапазон в два раза меньший, чем в 2. Так что 1 будет выдаваться в два раза реже, чем 2.

Верное решение с round

Правильный способ:

`Math.round(случайное от min-0.5 до max+0.5)`

```
1 var min = 1, max = 3;
2
3 var rand = min - 0.5 + Math.random()*(max-min+1)
4 rand = Math.round(rand);
5
6 alert(rand);
```

В этом случае диапазон будет тот же (`max-min+1`), но учтена механика округления `round`.

Решение с floor

Альтернативный путь - применить округление `Math.floor()` к случайному числу от `min` до `max+1`.

Например, для генерации целого числа от 1 до 3, создадим вспомогательное случайное значение от 1 до 4 (не включая 4).

Тогда `Math.floor()` округлит их так:

```
1 ... 1.999+ станет 1
2 ... 2.999+ станет 2
3 ... 3.999+ станет 3
```

Все диапазоны одинаковы.

Итак, код:

```
1 var min=5, max=10;
2 var rand = min + Math.random()*(max+1-min);
3 rand = rand^0; // округление битовым оператором
4 alert(rand);
```



Решение задачи: Первый объект

```
1 var user = {};
2 user.name = "Вася";
3 user.surname = "Петров";
4 user.name = "Сергей";
5 delete user.name;
```



Решение задачи: multiplyNumeric

```
01 var menu = {
02   width: 200,
03   height: 300,
04   title: "My menu"
05 };
06
07 function isNumeric(n) {
08   return !isNaN(parseFloat(n)) && isFinite(n);
09 }
10
11 function multiplyNumeric(obj) {
12   for(var key in obj) {
13     if (isNumeric( obj[key] )) {
14       obj[key] *= 2;
15     }
16   }
17 }
```

```
16 }  
17 }  
18  
19 multiplyNumeric(menu);  
20  
21 alert("menu width="+menu.width+"  
height="+menu.height+" title="+menu.title);
```



Решение задачи: Получить последний элемент массива

Последний элемент имеет индекс на 1 меньший, чем длина массива.

Например:

```
var fruits = ["Яблоко", "Груша", "Слива"];
```

Длина массива этого массива `fruits.length` равна 3. Здесь «Яблоко» имеет индекс 0, «Груша» — индекс 1, «Слива» — индекс 2.

То есть, для массива длины `goods`:

```
var lastItem = goods[goods.length-1]; // получить  
последний элемент
```



Решение задачи: Добавить новый элемент в массив

Текущий последний элемент имеет индекс `goods.length-1`.

Значит, индексом нового элемента будет `goods.length`:

```
goods[goods.length] = 'Компьютер'
```



Решение задачи: Создание массива

```
1 var styles = ["Джаз", "Блюз"];  
2 styles.push("Рок-н-Ролл");  
3 styles[styles.length-2] = "Классика";  
4 alert( styles.shift() );  
5 styles.unshift( "Рэп", "Регги " );
```





Решение задачи: Получить случайное значение из массива

Для вывода нужен случайный номер от 0 до `arr.length-1` включительно.

```
1 var arr = ["Яблоко", "Апельсин", "Груша", "Лимон"];
2
3 var rand = Math.floor( Math.random() * arr.length
4 );
5 alert(arr[rand]);
```



Решение задачи: Создайте калькулятор для введенных значений

Решение:

<http://learn.javascript.ru/play/tutorial/intro/array/calculator.html>

Перед `prompt` стоит `+` для преобразования к числу, иначе калькулятор будет складывать строки.



Решение задачи: Чему равен элемент массива?

```
1 var arr = [1,2,3];
2
3 var a = arr; // (*)
4 a[0] = 5;
5
6 alert(arr[0]);
7 alert(a[0]);
```

Код выведет 5 в обоих случаях, так как массив является объектом. В строке (*) в переменную `a` копируется ссылка на него, а сам объект в памяти по-прежнему один, в нём отражаются изменения, внесенные через `a` или `arr`.

Если нужно именно скопировать массив, то это можно сделать, например, так:

```
var a = [];
for(var i=0; i<arr.length; i++) a[i] = arr[i];
```





Решение задачи: Поиск в массиве

Возможное решение:

```
1 function find(array, value) {  
2  
3     for(var i=0; i<array.length; i++) {  
4         if (array[i] == value) return i;  
5     }  
6  
7     return -1;  
8 }
```

Однако, в нем ошибка, т.к. сравнение `==` не различает `0` и `false`.

Поэтому лучше использовать `===`. Кроме того, в современном стандарте JavaScript существует встроенная функция

`Array#indexOf`, которая работает именно таким образом. Имеет смысл ей воспользоваться, если браузер ее поддерживает.

```
01 function find(array, value) {  
02     if (array.indexOf) { // если метод существует  
03         return array.indexOf(value);  
04     }  
05  
06     for(var i=0; i<array.length; i++) {  
07         if (array[i] === value) return i;  
08     }  
09  
10     return -1;  
11 }  
12  
13 var arr = ["a", -1, 2, "b"];  
14  
15 var index = find(arr, 2);  
16  
17 alert(index);
```

... Но еще лучшим вариантом было бы определить `find` по-разному в зависимости от поддержки браузером метода `indexOf`:

```
01 // создаем пустой массив и проверяем поддерживается  
    // ли indexOf  
02 if ( [].indexOf ) {  
03  
04     var find = function(array, value) {
```

```

05     return array.indexOf(value);
06 }
07
08 } else {
09     var find = function(array, value) {
10         for(var i=0; i<array.length; i++) {
11             if (array[i] === value) return i;
12         }
13
14         return -1;
15     }
16
17 }

```

Этот способ - лучше всего, т.к. не требует при каждом запуске find проверять поддержку indexOf.



Решение задачи: Фильтр диапазона

Решение, шаг 1

Алгоритм решения:

1. Создайте временный пустой массив `var results = []`.
2. Пройдите по элементам `arr` в цикле и заполните его.
3. Возвратите `results`.

Решение, шаг 2

Код: <http://learn.javascript.ru/play/tutorial/intro/array/filterRange.html>.



Решение задачи: Решето Эратосфена

Их сумма равна 1060.

Решение: <http://learn.javascript.ru/play/tutorial/intro/array/sieve.html>.



Решение задачи: Подмассив наибольшей суммы

Подсказка для $O(n^2)$

Можно просто посчитать для каждого элемента массива все суммы, которые с него начинаются.

Например, для [-1, 2, 3, -9, 11]:

```
01 // Начиная с -1:
02 -1
03 -1 + 2
04 -1 + 2 + 3
05 -1 + 2 + 3 + (-9)
06 -1 + 2 + 3 + (-9) + 11
07
08 // Начиная с 2:
09 2
10 2 + 3
11 2 + 3 + (-9)
12 2 + 3 + (-9) + 11
13
14 // Начиная с 3:
15 3
16 3 + (-9)
17 3 + (-9) + 11
18
19 // Начиная с -9
20 -9
21 -9 + 11
22
23 // Начиная с -11
24 -11
```

Сделайте вложенный цикл, который на внешнем уровне бежит по элементам массива, а на внутреннем — формирует все суммы элементов, которые начинаются с текущей позиции.

Решение для $O(n^2)$

Решение через вложенный цикл:

```
01 function getMaxSubSum(arr) {
02     var maxSum = 0; // если совсем не брать элементов,
    то сумма 0
03
04     for(var i=0; i<arr.length; i++) {
05         var sumFixedStart = 0;
06         for(var j=i; j<arr.length; j++) {
07             sumFixedStart += arr[j];
08             maxSum = Math.max(maxSum, sumFixedStart);
09         }
10     }
11
12     return maxSum;
```

```

13 }
14
15 alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
16 alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
17 alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
18 alert( getMaxSubSum([1, 2, 3]) ); // 6
19 alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100

```

Алгоритм для $O(n)$

Будем идти по массиву и накапливать в некоторой переменной s текущую частичную сумму. Если в какой-то момент s окажется отрицательной, то мы просто присвоим $s=0$. Утверждается, что максимум из всех значений переменной s , случившихся за время работы, и будет ответом на задачу.

Докажем этот алгоритм.

В самом деле, рассмотрим первый момент времени, когда сумма s стала отрицательной. Это означает, что, стартовав с нулевой частичной суммы, мы в итоге пришли к отрицательной частичной сумме — значит, и весь этот префикс массива, равно как и любой его суффикс имеют отрицательную сумму.

Следовательно, от всего этого префикса массива в дальнейшем не может быть никакой пользы: он может дать только отрицательную прибавку к ответу.

Решение за $O(n)$

```

01 function getMaxSubSum(arr) {
02     var maxSum = 0, partialSum = 0;
03     for (var i=0; i<arr.length; i++) {
04         partialSum += arr[i];
05         maxSum = Math.max(maxSum, partialSum);
06         if (partialSum < 0) partialSum = 0;
07     }
08     return maxSum;
09 }
10
11
12 alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
13 alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
14 alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
15 alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
16 alert( getMaxSubSum([1, 2, 3]) ); // 6

```

```
17 | alert( getMaxSubSum([-1, -2, -3]) ); // 0
```

Информацию об алгоритме вы также можете прочитать здесь: http://e-maxx.ru/algorithm/maximum_average_segment и здесь: [Maximum subarray problem](#) .



Решение задачи: Оставить уникальные элементы массива

Решение перебором (медленное)

Пройдём по массиву вложенным циклом.

Для каждого элемента мы будем искать, был ли такой уже. Если был — игнорировать:

```
01 | function unique(arr) {
02 |     var obj = {};
03 |     var result = [];
04 |
05 |     nextInput:
06 |     for(var i=0; i<arr.length; i++) {
07 |         var str = arr[i];           // для
каждого элемента
08 |         for(var j=0; j<result.length; j++) { // ищем,
был ли он уже?
09 |             if (result[j] == str) continue nextInput; //
если да, то следующий
10 |         }
11 |         result.push(str);
12 |     }
13 |
14 |     return result;
15 | }
16 |
17 | var strings = ["кришна", "кришна", "хапе", "хапе",
18 |     "хапе", "хапе", "кришна", "кришна", "8-()"];
19 |
20 | alert( unique(strings) ); // кришна, хапе, 8-()
```

Давайте посмотрим, насколько быстро он будет работать.

Предположим, в массиве 100 элементов. Если все они одинаковые, то `result` будет состоять из одного элемента и вложенный цикл будет выполняться сразу. В этом случае всё хорошо.

А если все, или почти все элементы разные?

В этом случае для каждого элемента понадобится обойти весь текущий массив результатов, после чего — добавить в этот массив.

1. Для первого элемента — это обойдётся в 0 операций доступа к элементам `result` (он пока пустой).
2. Для второго элемента — это обойдётся в 1 операцию доступа к элементам `result`.
3. Для третьего элемента — это обойдётся в 2 операции доступа к элементам `result`.
4. ...Для n -го элемента — это обойдётся в $n-1$ операций доступа к элементам `result`.

Всего $0 + 1 + 2 + \dots + n-1 = (n-1)*n/2 = n^2/2 - n/2$ (как сумма арифметической прогрессии), то есть количество операций растёт примерно как квадрат от n .

Это очень быстрый рост. Для 100 элементов — 4950 операций, для 1000 — 499500 (по формуле выше).

Поэтому такое решение подойдёт только для небольших массивов. Вместо вложенного `for` можно использовать и `arr.indexOf`, ситуация от этого не поменяется, так как `indexOf` тоже ищет перебором.

Решение с объектом (быстрое)

Наилучшая техника для выбора уникальных строк — использование вспомогательного объекта. Ведь название свойства в объекте, с одной стороны — строка, а с другой — всегда уникально. Повторная запись в свойство с тем же именем перезапишет его.

Например, если `"харе"` попало в объект один раз (`obj["харе"] = true`), то второе такое же присваивание ничего не изменит.

Решение ниже создаёт объект `obj = {}` и записывает в него все строки как имена свойств. А затем собирает свойства из объекта

в массив через `for...in`. Дубликатов уже не будет.

```
01 function unique(arr) {
02   var obj = {};
03
04   for(var i=0; i<arr.length; i++) {
05     var str = arr[i];
06     obj[str] = true; // запомнить строку в виде
                        свойства объекта
07   }
08
09   return Object.keys(obj); // или собрать ключи
                        перебором для IE<9
10 }
11
12 var strings = ["кришна", "кришна", "харе", "харе",
13               "харе", "харе", "кришна", "кришна", "8-()"];
14
15 alert( unique(strings) ); // кришна, харе, 8-()
```

Так что можно положить все значения как ключи в объект, а потом достать.



Решение задачи: Добавить класс в строку

Решение заключается в превращении `obj.className` в массив при помощи `split`.

После этого в нем можно проверить наличие класса, и если нет - добавить.

```
01 function addClass(obj, cls) {
02   var classes = obj.className ?
03     obj.className.split(' ') : [];
04
05   for(var i=0; i<classes.length; i++) {
06     if (classes[i] == cls) return; // класс уже есть
07   }
08   classes.push(cls); // добавить
09
10   obj.className = classes.join(' '); // и обновить
                        СВОЙСТВО
11 }
12
13 var obj = { className: 'open menu' };
14
```

```
15 | addClass(obj, 'new');
16 | addClass(obj, 'open');
17 | addClass(obj, 'me');
18 | alert(obj.className) // open menu new me
```

P.S. «Альтернативный» подход к проверке наличия класса вызовом `obj.className.indexOf(cls)` был бы неверным. В частности, он найдёт `cls = "menu"` в строке классов `obj.className = "open mymenu"`.

P.P.S. Проверьте, нет ли в вашем решении присвоения `obj.className += " " + cls`. Не добавляет ли оно лишний пробел в случае, если изначально `obj.className = ""`?



Решение задачи: Перевести текст вида `border-left-width` в `borderLeftWidth`

Идея

Задача может быть решена несколькими способами. Один из них — разбить строку по дефису `str.split('-')`, затем последовательно сконструировать новую.

Решение

Разобьём строку в массив, а затем преобразуем его элементы и сольём обратно:

```
01 | function camelize(str) {
02 |     var arr = str.split('-');
03 |
04 |     for(var i=1; i<arr.length; i++) {
05 |         // преобразовать: первый символ с большой буквы
06 |         arr[i] = arr[i].charAt(0).toUpperCase() +
07 |         arr[i].slice(1);
08 |     }
09 |     return arr.join('');
10 | }
```

Демо: <http://learn.javascript.ru/play/tutorial/intro/array/camelize.html>.



Решение задачи: Функция `removeClass`

Решение заключается в том, чтобы разбить `className` в массив классов, а затем пройтись по нему циклом. Если класс есть - удаляем его `splice`, заново объединяем массив в строку и присваиваем объекту.

```
01 function removeClass(obj, cls) {
02   var classes = obj.className.split(' ');
03
04   for(i=0; i<classes.length; i++) {
05     if (classes[i] == cls) {
06       classes.splice(i, 1); // удалить класс
07       i--; // (*)
08     }
09   }
10   obj.className = classes.join(' ');
11
12 }
13
14 var obj = { className: 'open menu menu' }
15
16 removeClass(obj, 'blabla');
17 removeClass(obj, 'menu')
18 alert(obj.className) // open
```

В примере выше есть тонкий момент. Элементы массива проверяются один за другим. При вызове `splice` удаляется текущий, `i`-й элемент, и те элементы, которые идут дальше, сдвигаются на его место.

Таким образом, на месте `i` оказывается новый, непроверенный элемент.

Чтобы это учесть, строчка `(*)` уменьшает `i`, чтобы следующая итерация цикла заново проверила элемент с номером `i`. Без нее функция будет работать с ошибками.



Решение задачи: Фильтрация массива "на месте"

<http://learn.javascript.ru/play/tutorial/intro/array/filterRangeInPlace.html>.



Решение задачи: Сортировать в обратном порядке

```
1 var arr = [ 5, 2, 1, -10, 8];
2
```

```
3 function compareReversed(a, b) {  
4   return b - a;  
5 }  
6  
7 arr.sort(compareReversed);  
8  
9 alert(arr);
```



Решение задачи: Скопировать и отсортировать массив

Для копирования массива используем `slice()`, и тут же — сортировку:

```
1 var arr = [ "HTML", "JavaScript", "CSS" ];  
2  
3 var arrSorted = arr.slice().sort();  
4  
5 alert(arrSorted);  
6 alert(arr);
```



Решение задачи: Случайный порядок в массиве

Решение, шаг 1

Функция сортировки должна возвращать случайный результат сравнения. Используйте для этого `Math.random`.

Решение, шаг 2

Решение:

Обычно `Math.random()` возвращает результат от 0 до 1. Вычтем 0.5, чтобы область значений стала `[-0.5 ... 0.5]`.

```
1 var arr = [1, 2, 3, 4, 5];  
2  
3 function compareRandom(a, b) {  
4   return Math.random() - 0.5;  
5 }  
6  
7 arr.sort(compareRandom);  
8  
9 alert(arr); // элементы в случайном порядке,  
             // например [3,5,1,2,4]
```




Решение задачи: Сортировка объектов

Для сортировки объявим передадим в `sort` анонимную функцию, которая сравнивает объекты по полю `age`:

```
01 // Наша функция сравнения
02 function compareAge(personA, personB) {
03     return personA.age - personB.age;
04 }
05
06 // проверка
07 var vasya = { name: "Вася", age: 23 };
08 var masha = { name: "Маша", age: 18 };
09 var vovochka = { name: "Вовочка", age: 6 };
10
11 var people = [ vasya , masha , vovochka ];
12
13 people.sort(compareAge);
14
15 // вывести
16 for(var i=0; i<people.length; i++) {
17     alert(people[i].name); // Вовочка Маша Вася
18 }
```



Решение задачи: Вывести односвязный список

Вывод списка

```
01 var list = {
02     value: 1, next: {
03         value: 2, next: {
04             value: 3, next: {
05                 value: 4, next: null
06             }
07         }
08     }
09 };
10
11 function printList(list) {
12     var tmp = list;
13
14     while(tmp) {
15         alert( tmp.value );
16         tmp = tmp.next;
17     }
18 }
```

```
19 }  
20  
21 printList(list);
```

Обратите внимание, что для прохода по списку используется временная переменная `tmp`, а не `list`. Можно было бы и бегать по списку, используя входной параметр функции:

```
1 function printList(list) {  
2  
3   while(list) {  
4     alert( list.value );  
5     list = list.next;  
6   }  
7  
8 }
```

...Но при этом мы в будущем не сможем расширить функцию и сделать со списком что-то ещё, ведь после окончания цикла начало списка уже нигде не хранится.

Поэтому и используется временная переменная — чтобы сделать код расширяемым, и, кстати, более понятным, ведь роль `tmp` — исключительно обход списка, как `i` в цикле `for`.

Обратный вывод с рекурсией

```
01 var list = {  
02   value: 1, next: {  
03     value: 2, next: {  
04       value: 3, next: {  
05         value: 4, next: null  
06       }  
07     }  
08   }  
09 };  
10  
11 function printReverseList(list) {  
12  
13   if (list.next) {  
14     printReverseList(list.next);  
15   }  
16  
17   alert(list.value);  
18 }  
19  
20 printReverseList(list);
```

Обратный вывод без рекурсии

```
01 var list = {
02   value: 1, next: {
03     value: 2, next: {
04       value: 3, next: {
05         value: 4, next: null
06       }
07     }
08   }
09 };
10
11
12 function printReverseList(list) {
13   var arr = [];
14   var tmp = list;
15
16   while(tmp) {
17     arr.push(tmp.value);
18     tmp = tmp.next;
19   }
20
21   for( var i = arr.length-1; i>=0; i-- ) {
22     alert( arr[i] );
23   }
24 }
25
26 printReverseList(list);
```

Обратный вывод без рекурсии быстрее.

По сути, рекурсивный вариант и нерекурсивный работают одинаково: они проходят список и запоминают его элементы, а потом выводят в обратном порядке.

В случае с массивом это очевидно, а для рекурсии запоминание происходит в стеке (внутренней специальной структуре данных): когда вызывается вложенная функция, то интерпретатор сохраняет в стек текущие параметры. Вложенные вызовы заполняют стек, а потом он выводится в обратном порядке.

При этом, при рекурсии в стеке сохраняется не только элемент списка, а другая вспомогательная информация, необходимая для возвращения из вложенного вызова. Поэтому тратится больше памяти. Все эти расходы отсутствуют во варианте без рекурсии,

так как в массиве хранится именно то, что нужно.

Преимущество рекурсии, с другой стороны — более короткий и, зачастую, более простой код.



Решение задачи: Отфильтровать анаграммы

Чтобы обнаружить анаграммы, разобьём каждое слово на буквы и отсортируем их. В отсортированном по буквам виде все анаграммы одинаковы.

Например:

```
воз, зов -> взо  
киборг, гробик -> бгикор  
...
```

По такой последовательности будем делать массив уникальным. Для этого воспользуемся вспомогательным объектом, в который будем записывать слова по отсортированному ключу:

```
01 function aclean(arr) {  
02     // этот объект будем использовать для уникальности  
03     var obj = {};  
04  
05     for(var i=0; i<arr.length; i++) {  
06         // разбить строку на буквы, отсортировать и  
07         // слить обратно  
08         var sorted =  
09         arr[i].toLowerCase().split('').sort().join(''); //  
10         (*)  
11  
12         obj[sorted] = arr[i]; // сохраняет только одно  
13         значение с таким ключом  
14     }  
15  
16     var result = [];  
17  
18     // теперь в obj находится для каждого ключа ровно  
19     // одно значение  
20     for(var key in obj) result.push(obj[key]);  
21  
22     return result;  
23 }  
24  
25 var arr = ["воз", "киборг", "корсет", "зов",
```

```
21 "гробик", "костер", "сектор"];  
22 alert( aclean(arr) );
```

Приведение слова к сортированному по буквам виду осуществляется цепочкой вызовов в строке (*).

Для удобства комментирования разобьём её на несколько строк (JavaScript это позволяет):

```
1 var sorted = arr[i] // 30В  
2   .toLowerCase(); // 30в  
3   .split('') // ['з','о','в']  
4   .sort() // ['в','з','о']  
5   .join('');
```

Получится, что два разных слова '30В' и 'воз' получат одинаковую отсортированную форму 'взо'.

Следующая строка:

```
obj[sorted] = arr[i];
```

В объект `obj` будет записано сначала первое из слов `obj['взо'] = "воз"`, а затем `obj['взо'] = '30В'`.

Обратите внимание, ключ — отсортирован, а само слово — в исходной форме, чтобы можно было потом получить его из объекта.

Вторая запись по тому же ключу перезапишет первую, то есть в объекте останется ровно одно слово с таким набором букв.



Решение задачи: Создайте дату

Дата в местной временной зоне создается при помощи `new Date`.

Месяцы начинаются с нуля, так что февраль имеет номер 1.

Параметры можно указывать с точностью до минут:

```
1 var d = new Date(2012, 1, 20, 3, 12);  
2 alert(d);
```



Решение задачи: Имя дня недели

Метод `getDay()` позволяет получить номер дня недели, начиная с воскресенья.

Запишем имена дней недели в массив, чтобы можно было их достать по номеру:

```
1 function getWeekDay(date) {  
2     var days = ['вс', 'пн', 'вт', 'ср', 'чт', 'пт', 'сб'] ;  
3  
4     return days[ date.getDay() ];  
5 }  
6  
7 var date = new Date(2012,0,3); // 3 января 2012  
8 alert( getWeekDay(date) ); // 'вт'
```



Решение задачи: День недели в европейской нумерации

Решение - в использовании встроенной функции `getDay`. Она полностью подходит нашим целям, но для воскресенья возвращает 0 вместо 7:

```
01 function getLocalDay(date) {  
02  
03     var day = date.getDay();  
04  
05     if ( day == 0 ) { // день 0 становится 7  
06         day = 7;  
07     }  
08  
09     return day;  
10 }  
11  
12 alert( getLocalDay(new Date(2012,0,3)) ); // 2
```

Если удобнее, чтобы день недели начинался с нуля, то можно возвращать в функции `day - 1`, тогда дни будут от 0 (пн) до 6(вс).



Решение задачи: День 100 дней назад

Создадим текущую дату и вычтем 100 дней:

```
1 var d = new Date;  
2 d.setDate( d.getDate() - 100 );  
3
```

```
4 | alert( d.getDate() );
5 |
6 | var dayNames =
  | ['вс', 'пн', 'вт', 'ср', 'чт', 'пт', 'сб'];
7 | alert( dayNames[d.getDay()] );
```

Объект `Date` авто-исправит себя и выдаст правильный результат.

Обратите внимание на массив с именами дней недели.

«Нулевой» день — воскресенье.



Решение задачи: Последний день месяца?

Создадим дату из следующего месяца, но день не первый, а «нулевой» (т.е. предыдущий):

```
1 | function getLastDayOfMonth(year, month) {
2 |     var date = new Date(year, month+1, 0);
3 |     return date.getDate();
4 | }
5 |
6 | alert( getLastDayOfMonth(2012, 1) ); // 29
```



Решение задачи: Сколько секунд - до завтра?

Первая часть.

Для вывода достаточно сгенерировать `date`, соответствующий началу дня, т.е. «сегодня» 00 часов 00 минут 00 секунд.

Разница между текущей датой и началом дня — это количество миллисекунд от начала дня. Его можно легко перевести в секунды:

```
1 | var now = new Date();
2 |
3 | // создать объект из текущей даты, без
  | часов-минут-секунд
4 | var today = new Date(now.getFullYear(),
  | now.getMonth(), now.getDate());
5 |
6 | var diff = now - today; // разница в миллисекундах
7 | alert( Math.round(diff / 1000) ); // перевести в
  | секунды
```

Вторая часть

Для получения оставшихся до конца дня секунд нужно из «завтра 00ч 00мин 00сек» вычесть текущее время.

Чтобы сгенерировать «завтра», нужно увеличить текущий день на 1:

```
1 var now = new Date();
2
3 // создать объект из даты, без часов-минут-секунд
4 var tomorrow = new Date(now.getFullYear(),
5   now.getMonth(), now.getDate()+1);
6
7 var diff = tomorrow - now; // разница в
  миллисекундах
8 alert( Math.round(diff / 1000) ); // перевести в
  секунды
```



Решение задачи: Вывести текущую дату

Получим компоненты один за другим.

1. День можно получить как `date.getDate()`. При необходимости добавим ведущий ноль:

```
var dd = date.getDate();
if (dd<10) dd= '0'+dd;
```

2. `date.getMonth()` возвратит месяц, начиная с нуля. Увеличим его на 1:

```
var mm = date.getMonth() + 1; // месяц 1-12
if (mm<10) mm= '0'+mm;
```

3. `date.getFullYear()` вернет год в 4-значном формате. Чтобы сделать его двузначным - воспользуемся оператором взятия остатка `'%'`:

```
var yy = date.getFullYear() % 100;
if (yy<10) yy= '0'+yy;
```

Заметим, что год, как и другие компоненты, может понадобиться дополнить нулем слева, причем возможно что

yy == 0 (например, 2000 год). При сложении со строкой 0+'0' == '00', так что будет все в порядке.

Полный код:

```
01 function formatDate(date) {
02
03     var dd = date.getDate()
04     if ( dd < 10 ) dd = '0' + dd;
05
06     var mm = date.getMonth()+1
07     if ( mm < 10 ) mm = '0' + mm;
08
09     var yy = date.getFullYear() % 100;
10     if ( yy < 10 ) yy = '0' + yy;
11
12     return dd+'.'+mm+'.'+yy;
13 }
14
15 var d = new Date(2011, 0, 30); // 30 Jan 2011
16 alert( formatDate(d) ); // '30.01.11'
```



Решение задачи: Относительное форматирование даты

Для того, чтобы узнать время от date до текущего момента - используем вычитание дат.

```
01 function formatDate(date) {
02     var diff = new Date() - date; // разница в
    миллисекундах
03
04     if (diff < 1000) { // прошло менее 1 секунды
05         return 'только что';
06     }
07
08     var sec = Math.floor( diff / 1000 ); // округлить
    diff до секунд
09
10     if (sec < 60) {
11         return sec + ' сек. назад';
12     }
13
14     var min = Math.floor( diff / 60000 ); // округлить
    diff до минут
15     if (min < 60) {
16         return min + ' мин. назад';
17     }
18 }
```

```

18
19 // форматировать дату, с учетом того, что месяцы
    начинаются с 0
20 var d = date;
21 d =
    ['0'+d.getDate(), '0'+(d.getMonth()+1), ''+d.getFullYear(),
    '0'+d.getHours(), '0'+d.getMinutes() ];
22 for(var i=0; i<d.length; i++) {
23     d[i] = d[i].slice(-2);
24 }
25
26 return d.slice(0,3).join('.')+'
    '+d.slice(3).join(':');
27 }
28
29 alert( formatDate( new Date( new Date - 1) ) ); //
    ТОЛЬКО ЧТО
30
31 alert( formatDate( new Date( new Date - 30*1000) )
    ); // 30 сек. назад
32
33 alert( formatDate( new Date( new Date- 5*60*1000) )
    ); // 5 мин. назад
34
35 alert( formatDate( new Date( new Date - 86400*1000)
    ) ); // вчерашняя дата в формате "дд.мм.гг чч:мм"

```



Решение задачи: Вопросник по преобразованиям, для примитивов

```

01 "" + 1 + 0 = "10" // (1)
02 "" - 1 + 0 = -1 // (2)
03 true + false = 1
04 6 / "3" = 2
05 "2" * "3" = 6
06 4 + 5 + "px" = "9px"
07 "$" + 4 + 5 = "$45"
08 "4" - 2 = 2
09 "4px" - 2 = NaN
10 7 / 0 = Infinity
11 parseInt("09") = "0" или "9" // (3)
12 " -9\n" + 5 = " -9\n5"
13 " -9\n" - 5 = -14
14 5 && 2 = 2
15 2 && 5 = 5
16 5 || 0 = 5
17 0 || 5 = 5

```

```
18 | null + 1 = 1 // (4)
19 | undefined + 1 = NaN // (5)
```

1. Оператор "+" в данном случае прибавляет 1 как строку, и затем 0.
2. Оператор "-" работает только с числами, так что он сразу приводит "" к 0.
3. В некоторых браузерах parseInt без второго аргумента интерпретирует 09 как восьмиричное число.
4. null при численном преобразовании становится 0
5. undefined при численном преобразовании становится NaN