

# Конспект JS-course

maxello

Published  
with GitBook



# Table of Contents

---

1. Конспект JS-course
2. Операторы, приведение типов, ветвление
  - i. Основные операторы
  - ii. Выражения и операторы
  - iii. Приоритеты операторов
  - iv. Задачи
3. Игра угадайка. Почитать про массивы, объекты, циклы
4. Циклы, массивы. Почитать про методы
  - i. Конспект по массивам, объектам, структурам данных
  - ii. Конспект. Методы
5. Функции
  - i. MDN Функции
  - ii. Функция - это значение
  - iii. Function Declaration и Function Expression
  - iv. Named Function Expression
  - v. Псевдо-массив arguments
  - vi. Именованные аргументы
  - vii. Задачи
    - i. Как в функции отличить отсутствующий аргумент от undefined?
    - ii. Напишите функцию `sum(...)`, которая возвращает сумму всех своих аргументов
    - iii. Функция, которая объединяет несколько строк
6. Функции и области видимости
  - i. Глобальный объект
  - ii. Замыкания, функции изнутри
  - iii. Хранение данных в замыкании, модули
  - iv. Выразительный JavaScript: Функции
  - v. Конспект. Функции и области видимости
  - vi. Задачи
    - i. Напишите функцию `sum`, которая работает так: `sum(a)(b) = a + b`.
    - ii. Функция `filter(arr, func)`
7. Функции. Продолжение
  - i. Строгий режим "use strict"
  - ii. Конспект. Функции, области видимости, замыкания
  - iii. Методы массивов
    - i. Метод `indexOf()`
    - ii. Метод `filter()`
    - iii. Метод `map()`
    - iv. Метод `forEach()`
    - v. Метод `every()`
    - vi. Метод `some()`
    - vii. Метод `reduce()`
8. Функции, методы, наследование
  - i. Свои объекты: конструкторы и методы
  - ii. Контекст `this` в деталях
  - iii. Тонкости ECMA-262-3. Часть 7.1. ООП: Общая теория
  - iv. Наследование и цепочка прототипов
  - v. Методы функций
    - i. Метод `call()`
    - ii. Метод `apply()`
  - vi. Конспект. Функции, `debugger`, `this`
  - vii. Задачи
    - i. Создайте объект `calculator` с тремя методами

- ii. Напишите функцию-конструктор Summator, которая создает объект с двумя методами
  - iii. Напишите функцию-конструктор Adder(startingValue).
- 9. Классы, наследование
  - i. Прототип: наследование и методы
  - ii. "Классы" в JavaScript
  - iii. Прототипное наследование
  - iv. Задачи
  - v. Конспект. Функции, конструкторы, прототипы
- 10. Массивы, чтение
  - i. Расширение встроенных прототипов
  - ii. Область применения наследования
  - iii. Тайная жизнь объектов
  - iv. Задачи
- 11. DOM (document object model)
  - i. JavaScript и браузер
  - ii. Окружение: DOM, BOM и JS
  - iii. BOM-объекты: navigator, screen, location, frames
  - iv. Работа с DOM из консоли
- 12. Читаем про html, css
- 13. Верстка
- 14. Верстка
- 15. Верстка
- 16. DOM
  - i. Дерево DOM
  - ii. Навигация в DOM, свойства-ссылки
  - iii. Свойства узлов: тип, тег, содержимое и другие
  - iv. Добавление и удаление узлов
  - v. Задачи
- 17. DOM events
  - i. Введение в браузерные события
  - ii. Всплытие и перехват
  - iii. Действия браузера по умолчанию
  - iv. Объект "событие" (event)
  - v. События мыши
    - i. Введение: клики, кнопка, координаты
    - ii. События движения: "mouseover/out/move/leave/enter"
  - vi. Задачи
- 18. DOM events
  - i. Делегирование событий
  - ii. Задачи

# Конспект JS-course

---

@maxellort

# Операторы, приведение типов, ветвление

---

Будем говорить про операторы (побитовые пропускаем), приведение типов, сравнение, ветвление.

## Операторы

---

Пропустить побитовые, выполнить для себя задания, обсудить задания на форуме.

<http://learn.javascript.ru/operators>

### Операторы:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)

### Порядок операторов:

[https://developer.mozilla.org/ru/docs/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/ru/docs/JavaScript/Reference/Operators/Operator_Precedence)

### Приведение типов:

<http://learn.javascript.ru/types-conversion>

### Логические операторы и сравнение:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)

<http://learn.javascript.ru/logical-ops>

<http://learn.javascript.ru/comparison>

### if-else:

<http://learn.javascript.ru/ifelse>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else>

# Основные операторы

Источник: <http://learn.javascript.ru/operators>

## Термины: «унарный», «бинарный», «операнд»

У операторов есть своя терминология, которая используется во всех языках программирования.

**Операнд** – то, к чему применяется оператор. Например: `5 * 2` – оператор умножения с левым и правым операндами. Другое название: «аргумент оператора». Унарным называется оператор, который применяется к одному выражению. Например, оператор унарный минус `-` меняет знак числа на противоположный:

```
var x = 1;

alert(-x);    // -1, унарный минус

alert(-(x + 2)); // -3, унарный минус применён к результату сложения x + 2

alert(-(-3)); // 3
```

Бинарным называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

```
var x = 1, y = 3;

alert(y - x); // 2, бинарный минус
```

Работа унарного `+` и бинарного `+` в JavaScript существенно различается.

Это действительно разные операторы. Бинарный плюс складывает операнды, а унарный – ничего не делает в арифметическом плане, но зато приводит операнд к числовому типу.

## Арифметические операторы

Базовые арифметические операторы знакомы нам с детства: это плюс `+`, минус `-`, умножить `*`, поделить `/`.

Например:

```
var i = 2;

i = (2 + i) * 3 / i;

alert(i); // 6
```

Более редкий арифметический оператор `%` интересен тем, что никакого отношения к процентам не имеет. Его результат `a % b` – это остаток от деления `a` на `b`.

```
alert(5 % 2); // 1, остаток от деления 5 на 2

alert(8 % 3); // 2, остаток от деления 8 на 3

alert(6 % 3); // 0, остаток от деления 6 на 3
```

## Сложение строк, бинарный `+`

Если бинарный оператор `+` применить к строкам, то он их объединяет в одну:

```
var a = "моя" + "строка";  
alert(a); // моястрока
```

Если хотя бы один аргумент является строкой, то второй будет также преобразован к строке!

```
alert('1' + 2); // "12"  
alert(2 + '1'); // "21"
```

Это приведение к строке – особенность бинарного оператора `+`.

Остальные арифметические операторы работают только с числами и всегда приводят аргументы к числу.

```
alert('1' - 2); // -1  
alert(6 / '2'); // 3
```

## Унарный плюс +

Унарный плюс как арифметический оператор ничего не делает. Тем не менее, он широко применяется, так как его «побочный эффект» – преобразование значения в число.

Например, у нас есть два числа, в форме строк, и нужно их сложить. Бинарный плюс сложит их как строки, поэтому используем унарный плюс, чтобы преобразовать к числу.

## Присваивание

Оператор присваивания выглядит как знак равенства `=`:

```
var i = 1 + 2;  
alert(i); // 3
```

Он вычисляет выражение, которое находится справа, и присваивает результат переменной. Это выражение может быть достаточно сложным и включать в себя любые другие переменные:

```
var a = 1;  
var b = 2;  
  
a = b + a + 3; // (*)  
  
alert(a); // 6
```

В строке `(*)` сначала произойдет вычисление, использующее текущее значение `a` (т.е. 1), после чего результат перезапишет старое значение `a`.

Возможно присваивание по цепочке:

```
var a, b, c;  
  
a = b = c = 2 + 2;  
  
alert(a); // 4  
alert(b); // 4
```

```
alert(c); // 4
```

Такое присваивание работает справа-налево, то есть сначала вычислится самое правое выражение `2 + 2`, присвоится в `c`, затем выполнится `b = c` и, наконец, `a = b`.

**Оператор `=` возвращает значение.**

Все операторы возвращают значение. Вызов `x = выражение` записывает выражение в `x`, а затем возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
var a = 1;
var b = 2;

var c = 3 - (a = b + 1);

alert(a); // 3
alert(c); // 0
```

В примере выше результатом `(a = b + 1)` является значение, которое записывается в `a` (т.е. 3). Оно используется для вычисления `c`.

## Приоритет

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется **приоритетом**. Из школы мы знаем, что умножение в выражении `2 * 2 + 1` выполнится раньше сложения, т.к. его приоритет выше, а скобки явно задают порядок выполнения. Но в JavaScript – гораздо больше операторов, поэтому существует целая [таблица приоритетов](#).

В ней каждому оператору задан числовой приоритет. Тот, у кого число меньше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

## Инкремент/декремент: `++`, `--`

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу.

Для этого существуют даже специальные операторы:

**Инкремент `++`** увеличивает на 1:

```
var i = 2;
i++;      // более короткая запись для i = i + 1.
alert(i); // 3
```

**Декремент `--`** уменьшает на 1:

```
var i = 2;
i--;      // более короткая запись для i = i - 1.
alert(i); // 1
```

Инкремент/декремент можно применить только к переменной.

Вызывать эти операторы можно не только после, но и перед переменной: `i++` (называется «постфиксная форма») или `++i` («префиксная форма»).

Обе эти формы записи делают одно и то же: увеличивают на 1.



Тем не менее, между ними существует разница. Она видна только в том случае, когда мы хотим не только увеличить/уменьшить переменную, но и использовать результат в том же выражении.

```
var i = 1;
var a = ++i; // (*)

alert(a); // 2
```

В строке `(*)` вызов `++i` увеличит переменную, а затем вернёт её значение в `a`. То есть, в `a` попадёт значение `i` после увеличения.

Постфиксная форма `i++` отличается от префиксной `++i` тем, что возвращает старое значение, бывшее до увеличения.

В примере ниже в `a` попадёт старое значение `i`, равное 1:

```
var i = 1;
var a = i++; // (*)

alert(a); // 1
```

- Если результат оператора не используется, а нужно только увеличить/уменьшить переменную – без разницы, какую форму использовать:

```
var i = 0;
i++;
++i;
alert(i); // 2
```

- Если хочется тут же использовать результат, то нужна префиксная форма:

```
var i = 0;
alert( ++i ); // 1
```

- Если нужно увеличить, но нужно значение переменной до увеличения – постфиксная форма:

```
var i = 0;
alert( i++ ); // 0
```

**Инкремент/декремент можно использовать в любых выражениях.** При этом он имеет более высокий приоритет и выполняется раньше, чем арифметические операции.

## Оператор запятой

Запятая тоже является оператором. Ее можно вызвать явным образом, например:

```
a = (5, 6);

alert(a);
```

Запятая позволяет перечислять выражения, разделяя их запятой `,`. Каждое из них – вычисляется и отбрасывается, за исключением последнего, которое возвращается.

Запятая – единственный оператор, приоритет которого ниже присваивания. Обычно этот оператор используется в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке
for (a = 1, b = 3, c = a * b; a < 10; a++) {
  ...
}
```

Такие трюки используются во многих JavaScript-фреймворках для укорачивания кода.

# Выражения и операторы

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Expressions_and_Operators)

## Выражения

**Выражением** является любой валидный блок кода, который имеет ценность.

Концептуально, существуют два типа выражений: те которые присваивают переменной значение и те которые вычисляют значение без его присваивания куда-либо.

Все JavaScript выражения делятся на следующие категории:

1. Арифметические: равняются числу, для примера 3.14159 (Используют арифметические операторы).
2. Строчные: равняются текстовой строке, для примера, "Fred" или "234" (Используют строчные операторы).
3. Логические: равняются true или false (Используют логические операторы).
4. Объектные: равняются объекту.

## Операторы

В JavaScript имеются следующие типы операторов:

- Операторы присваивания
- Операторы сравнения
- Арифметические операторы
- Бинарные операторы
- Логические операторы
- Строчные операторы
- Специальные операторы

JavaScript поддерживает бинарные и унарные операторы, а также ещё один специальный тернарный оператор — условный оператор. Бинарная операция использует два операнда, один перед оператором (его знаком) и другой за ним:

```
operand1 operator operand2
```

В свою очередь унарная операция использует один операнд, перед или после оператора:

```
operator operand
```

или

```
operand operator
```

## Операторы присваивания

В результате операции присваивания операнду слева от оператора присваивания (знак `=`) устанавливается значение, которое берётся из правого операнда. Основным оператором присваивания является `=`, он присваивает значение правого операнда операнду находящемуся слева. Таким образом, выражение `x = y` означает, что `x` присваивает значение `y`.

Существуют также сокращенные операторы присваивания используемые для операций приведенных ниже в таблице:

Сокращенный оператор	Значение
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x &lt;= y</code>	<code>x = x &lt; y</code>
<code>x &gt;= y</code>	<code>x = x &gt; y</code>
<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
<code>x &amp;= y</code>	<code>x = x &amp; y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>

## Операторы сравнения

Оператор сравнения сравнивает свои операнды и возвращает логическое значение базируясь на равенстве или неравенстве значений операндов. Операнды могут быть числовыми, строчными, логическими или объектами. Строки сравниваются опираясь на стандартный лексикографический порядок, используя Unicode значения. В большинстве случаев, если операнды имеют разный тип, JavaScript пробует преобразовать их к подходящему типу для сравнения. Данное поведение обычно используется при сравнении числовых операндов. Исключением из этого правила является сравнение с использованием операторов `===` и `!==`, которые производят строгое сравнение на равенство или неравенство. Эти операторы не делают попытки преобразовать операнды перед их сравнением на равенство. Следующая таблица описывает операторы сравнения в контексте следующего примера кода:

```
var var1 = 3, var2 = 4;
```

Оператор	Описание	Примеры возвращающие true
Равно <code>==</code>	Возвращает true если операнды равны.	<code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code>
Не равно <code>!=</code>	Возвращает true если операнды не равны.	<code>var1 != 4</code> <code>var2 != "3"</code>
Строгое равно <code>===</code>	Возвращает true если операнды равны и имеют одинаковый тип.	<code>3 === var1</code>
Строго не равно <code>!==</code>	Возвращает true если операнды не равны и/или имеют разный тип.	<code>var1 !== "3"</code> <code>3 !== '3'</code>
Больше чем <code>&gt;</code>	Возвращает true если операнд слева больше операнда справа.	<code>var2 &gt; var1</code> <code>"12" &gt; 2</code>
Больше чем или равно <code>&gt;=</code>	Возвращает true если операнд слева больше или равен операнду справа.	<code>var2 &gt;= var1</code> <code>var1 &gt;= 3</code>
Меньше чем <code>&lt;</code>	Возвращает true если операнд слева меньше операнда	<code>var1 &lt; var2</code> <code>"2" &lt; "12"</code>

меньше чем <	справа.	var1 < var2 4 < 14
Меньше чем или равно <=	Возвращает true если операнд слева меньше или равен операнду справа.	var1 <= var2 var2 <= 5

## Арифметические операторы

Арифметические операторы работают с числами (иногда со строками или переменными других типов) в качестве своих операндов и возвращают единичный результат также число. Стандартными арифметическими операторами являются сложение + , вычитание - , умножение \* , и деление / .

```
console.log(1 / 2); /* напечатает 0.5 */
console.log(1 / 2 == 1.0 / 2.0); /* напечатает true */
```

# Приоритеты операторов

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

## Основная информация

Приоритет операторов определяет порядок, в котором операторы выполняются. Операторы с более высоким приоритетом выполняются первыми.

Пример:

```
3 + 4 * 5 // возвращает 23
```

Оператор умножения `*` имеет более высокий приоритет, чем оператор сложения `+` и, таким образом будет выполняться первым.

## Ассоциативность

Ассоциативность определяет порядок, в котором операторы с одинаковым приоритетом обрабатываются. Например, рассмотрим выражение:

```
a OP b OP c
```

Левая ассоциативность (left-to-right) означает, что оно обрабатывается как `(a OP b) OP c`, в то время как правая ассоциативность (right-to-left) означает, что оно интерпретируется как `a OP (b OP c)`. Операторы присваивания являются право-ассоциативными, так что Вы можете написать:

```
a = b = 5;
```

с ожидаемым результатом, что `a` и `b` будут равны 5. Это происходит, потому что оператор присваивания возвращает тот результат, который присваивает. Сначала `b` становится равным 5, затем `a` принимает значение `b`.

Данная таблица упорядочена с самого высокого приоритета (1) до самого низкого (18).

Precedence	Operator type	Associativity	Individual operators
1	member	left-to-right	<code>.</code> <code>[]</code>
1	new	right-to-left	<code>new</code>
2	function call	left-to-right	<code>()</code>
3	increment	n/a	<code>++</code>
3	decrement	n/a	<code>--</code>
4	logical-not	right-to-left	<code>!</code>
4	bitwise not	right-to-left	<code>~</code>
4	unary +	right-to-left	<code>+</code>
4	unary negation	right-to-left	<code>-</code>
4	typeof	right-to-left	<code>typeof</code>

4	void	right-to-left	void
4	delete	right-to-left	delete
5	multiplication	left-to-right	*
5	division	left-to-right	/
5	modulus	left-to-right	%
6	addition	left-to-right	+
6	subtraction	left-to-right	-
7	bitwise shift	left-to-right	<< >> >>>
8	relational	left-to-right	< <= > >=
8	in	left-to-right	in
8	instanceof	left-to-right	instanceof
9	equality	left-to-right	== != === !==
10	bitwise-and	left-to-right	&
11	bitwise-xor	left-to-right	^
12	bitwise-or	left-to-right	
13	logical-and	left-to-right	&&
14	logical-or	left-to-right	
15	conditional	right-to-left	?:
16	yield	right-to-left	yield
17	assignment	right-to-left	= += -= *= /= %= <<= >>= >>>= &= ^=  =
18	comma	left-to-right	,

# Задачи

---

## Задача 1

Посмотрите, понятно ли вам, почему код ниже работает именно так?

```
var a = 1, b = 1, c, d;

c = ++a; alert(c); // 2
d = b++; alert(d); // 1

c = (2 + ++a); alert(c); // 5
d = (2 + b++); alert(d); // 4

alert(a); // 3
alert(b); // 3
```

## Пояснение

```
var a = 1, b = 1, c, d;

// префиксная форма сначала увеличивает a до 2, а потом возвращает
c = ++a; alert(c); // 2

// постфиксная форма увеличивает, но возвращает старое значение
d = b++; alert(d); // 1

// сначала увеличили a до 3, потом использовали в арифметике
c = (2 + ++a); alert(c); // 5

// увеличили b до 3, но в этом выражении оставили старое значение
d = (2 + b++); alert(d); // 4

// каждую переменную увеличили по 2 раза
alert(a); // 3
alert(b); // 3
```



# Игра угадайка. Почитать про массивы, объекты, циклы

---

Будем говорить про массивы, объекты, циклы

## Объекты:

<http://learn.javascript.ru/object>

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working\\_with\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects) (дальше Using a constructor function не надо читать)

## Массивы:

<http://learn.javascript.ru/array>

<http://learn.javascript.ru/array-methods>

## Циклы:

<http://learn.javascript.ru/while-for>

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Statements#Loop\\_Statements](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Statements#Loop_Statements) (label не обязательно понимать, используется не так часто)

# Циклы, массивы. Почитать про методы

---

## Почитать:

На следующей встрече продолжаем говорить о массивах. А так же говорим о методах (строк, массивов, объектов)

Опустите детали как работают методы, которые получают аргументом функции ( `sort` , `replace` , `map` , `filter` ). Мы вернемся к этим методам, когда дойдем до функций.

<http://learn.javascript.ru/properties-and-methods>

<http://learn.javascript.ru/array-methods>

<http://habrahabr.ru/post/240813/>

# Конспект по массивам, объектам, структурам данных

Источник: <http://forum.jscourse.com/t/konspekt-po-massivam-obektam-strukturam-dannyh-video-03/551>

Автор конспекта: @Nikolay

Правки: @dmitry

## Ошибки в коде

Интерпретатор работает следующим образом: подаем ("скармливаем") интерпретатору текст, он его считывает, строит абстрактное синтаксическое дерево (разбивает на составляющие) и потом выполняет. То есть присутствуют 2 действия:

1. Считывание;
2. Выполнение.

Ошибки могут быть:

- **синтаксические** (когда интерпретатор не смог считать код). Их отловить проще всего;
- **ошибки runtime-ма** (или ошибки, которые возникли в ходе выполнения). Возникают по мере выполнения кода. Например, в каком-то куске вызываете функцию, а ее не существует. Тогда ошибка возникнет, когда интерпретатор дойдет до этого куска кода.

Интерпретатор не сообщает о месте, где ошибка, он сообщает о том месте, где у него возникла проблема. Но чаще всего эти два места совпадают. При обнаружении JShint -ом ошибки очень важно понять почему ошибка произошла, а не просто исправить указанную строчку.

## Массивы

**Массивы** - тип данных, который представляет множество однородных предметов.

Массивы используются когда нужно использовать последовательную структуру данных, т.е. в которых есть направление (векторы, очереди), где позиция элемента имеет значение (1, 2, 3).

Никогда не используйте для создания массивов конструкторы (!) (только литеральная запись).

Массивы в JavaScript'е нетипизированные. Можно создать массив как из чисел, так и из строк, и в перемешку, и массив из массивов (необходимо для матрицы). Функции и объекты также могут быть элементами массива (!).

```
var matrix = [['o', 'x', 'o'],
              [' ', 'x', 'o'],
              [' ', 'o', 'x']];

//Чтобы получить координату - используем:

console.log(matrix[1][1]) // 'x' ;
console.log(matrix[1])   // [' ', 'x', 'o'] ;

//Пробежимся в цикле по всем элементам:

for (var row = 0; row < matrix.length; row += 1) {
    for (var column = 0; column < matrix[row].length; column += 1) {
        console.log(row, column, matrix[row][column]);
    }
}
result:
0 0 "o"
0 1 "x"
0 2 "o"
```

```
1 0 " "  
...
```

Обращение к несуществующему элементу вычисляется в `undefined`. Очень частая ошибка - обращение к элементу, который выходит за рамки массива (например, ошибка в цикле с операторами `<`, `>`).

Нужно обязательно понимать когда получается `undefined` (!) (при этом не углубляясь в суть самого `undefined`). Поэтому, чтобы легче обнаружить где возвращается `undefined` - лучше писать как можно меньше замысловатый код.

## Объекты

Ключи в объектах всегда имеют строковые значения. Свойства - любые величины. Что интересно, `Math` - это объект с огромным количеством свойств-функций.

Ключи всегда должны быть уникальными. Иначе - ошибка при выполнении кода неизбежна (интерпретатор возьмет только последний ключ из 2-х одинаковых).

Стоит использовать литеральную нотацию.

Чтобы вывести все ключи объекта удобней использовать `console.dir`. Объекты являются проекцией чего-то существующего в код (например, адресной книги).

## Передача данных по ссылке и по значению

Следует запомнить, что **все непримитивные типы данных (как объекты, массивы, функции) хранятся и передаются по ссылке (!)**

Это означает, что в переменную (в данном случае `student1`) записывается не сам объект, а ссылка на него (с объекта). То есть, если мы создадим другую переменную и запишем в нее ссылку на этот же объект, то значение этих переменных (студентов) будет равно.

```
var mmAcademy = {  
  teacherRoot: "/Users/podgorniy/Dropbox/Academy/2014_11/teacher/",  
  studentsRoot: "/Users/podgorniy/Dropbox/Academy/2014_11/students/",  
  folderToCopyForStudent: ["examples", "task"],  
  students: [{  
    name: "Afanasiy Sergeevich",  
    skype: "afanas",  
    email: "afanas@gmail.com"  
  }, {  
    name: "Petro Kush",  
    skype: "petrokush",  
    email: "petrokush@gmail.com"  
  }  
]  
// Получаем информацию о первом студенте (первом элементе массива)  
  
var student1 = mmAcademy.students[0];  
// output:  
//Object { name: "Afanasiy Sergeevich", skype: "afanas", email: "afanas@gmail.com" }
```

То есть, если мы создадим другую переменную, и запишем в нее ссылку на этот же объект, то значение этих переменных (студентов) будет равно.

```
var student2 = mmAcademy.students[0];  
console.log(student1 === student2); // true
```

Но, если создать точно такой же объект (эквивалентный по составу), то:

```
var student2 = {  
  name: "Afanasiy Sergeevich",
```

```
    skype: "afanas",
    email: "afanas@gmail.com"};

console.log(student1 === student2); // false
```

Несмотря на то, что объекты одинаковы, и выглядят одинаковыми, они - разные, с точки зрения JS. Отож, сравнение и передача данных происходит по ссылке (!).

Тоже самое касается массивов.

```
var arr1 = [];
var arr2 = [];
console.log(arr1 === arr2); // false
```

Это происходит потому что в каждом массиве находятся ссылки на разные области памяти (!).

Еще один пример (объект `mmAcademy` тот же, что и выше). Есть две переменные: в одну записана ссылка на объект, и во вторую записана ссылка на этот же объект.

```
var student1 = mmAcademy.students[0];
var student2 = mmAcademy.students[0];
```

При создании нового свойства `mark` в объекте `mmAcademy`, в которое, к примеру, будет записано значение, изменения также отобразятся и в переменной `student2`, т.к. она содержит ссылку на тот же объект!

```
student1.mark = 4.3;
console.log(student2);

//Output:
//Object { name: "Afanasiy Sergeevich", skype: "afanas", email: "afanas@gmail.com", mark: 4.3 }
```

Это критично для понимания, поскольку передача по ссылке происходит довольно часто. Поэтому, если функция принимает какой-то аргумент, не стоит его менять, поскольку кто-то другой может использовать этот же аргумент в другом месте кода.

## Копирование объекта

Самый удобный способ скопировать объект - это превратить этот объект в строку, и потом обратно в объект. Для этого есть формат JSON с методами `stringify` и `parse`.

**JSON.stringify()** возвращает объект в строку. При этом объект не должен содержать ссылки на самого себя.

```
JSON.stringify({
  name: "Afanasiy Sergeevich",
  skype: "afanas",
  email: "afanas@gmail.com"
});

//Output:
//{"name":"Afanasiy Sergeevich","skype":"afanas","email":"afanas@gmail.com"}
```

**JSON.parse()** возвращает строку в объект.

```
JSON.parse('{"name":"Afanasiy Sergeevich","skype":"afanas","email":"afanas@gmail.com"}')

//Output:
//Object { name: "Afanasiy Sergeevich", skype: "afanas", email: "afanas@gmail.com" }
```

Заметьте, что JSON возвращает значение, которое можно записать в переменную.

# Конспект. Методы

Источник: <http://forum.jscourse.com/t/04-konspekt-metody/579>

Автор конспекта: @Nikolay

Правки: @dmitry

## Методы массивов

**Метод** — это какое-то действие которое связано с объектом и выполняет определенную функцию. Методы привязаны к объектам. Например все числа имеют метод `toString`, все строки — метод `split`.

```
Math.round(10.10); // вызов метода
```

## Метод объекта Math

Разные методы могут как принимать аргументы ((10.10) - аргумент), так и не принимать. Также следует учитывать, что конкретный метод может вести себя по разному в зависимости от того, что ему передано в качестве аргументов.

## Метод split

Если метод `split` вызывается с пустой строкой (без аргумента), то он разбивает строку посимвольно, иначе - использует заданный аргумент и разбивает по нему.

```
var str = 'http://forum.jscourse.com/t/02-zadanie-igra-ugadajka-pochitat-pro-massivy-obekty-czikly/482';
str.split('/');

Output:
[ "http:", "", "forum.jscourse.com", "t", "02-zadanie-igra-ugadajka-pochitat-pro-massivy-obekty-czikly", "482" ]
```

Метод может изменять объект/значение с которым он вызван. Например метод `push`.

```
var arr = [];
arr.push('one');
console.log(arr); // ["one"]
```

Следует очень внимательно использовать методы, которые изменяют объекты, переданные им для операций. Это важно потому что этот переданный объект может быть использован в другой части кода, что приведет к ошибкам. Лучше сделать копию объекта/значения с которым планируется работать.

## Копирование массивов

Для копирования массивов чаще всего используется метод `slice`. Синтаксис - `arrayObj.slice( start[, end] )`

```
var obj = {};
var arr = [obj, 'two', 'three', 'four', 'five'];
var arrCopy = arr.slice();
var arrCopy1 = arr.slice(1);
var arrCopy2 = arr.slice(1, 2);
console.log('arrCopy', arrCopy); // arrCopy [{}, "two", "three", "four", "five"]
console.log('arrCopy1', arrCopy1); // arrCopy1 ["two", "three", "four", "five"]
console.log('arrCopy2', arrCopy2); //arrCopy2 ["two"]
Если мы сравним объекты - то они разные

arr === arrCopy // false
```

А если сравним первые элементы этих объектов, то они одинаковые

```
arr[0] === arrCopy[0]
// true
```

Копию объекта также можно сделать с помощью следующих методов: с помощью `JSON.stringify` — превратить объект в строку, а с помощью метода `JSON.parse` — обратно в объект.

Добавить элемент в массив (с конца) можно использовать следующие способы:

```
arr[arr.length - 1] = 'item'; // или
arr.push('item');
```

Лучше использовать метод `push` - он понятней

## Метод sort

Метод `sort` изменяет массив и если его вызывать без аргумента, то он пытается привести все элементы массива к строкам и сравнивает их. Синтаксис - `arrayObj.sort( [sortFunction] )`

## PolyFill

Если вы хотите использовать метод, которые не поддерживаются той или иной версией браузера, то необходимо использовать PolyFill (полизаполнения).

**PolyFill** - это отрывок кода (скрипт) который вставляется в браузер, и который реализует возможность использовать конкретный метод (функцию) в браузерах которые ее не поддерживают.

Следует обращать внимание, какие методы являются стандартными и поддерживаться всеми браузерами, а какие — нет.

Для выполнения такой проверки есть полезный ресурс - <http://caniuse.com/>



# Функции

---

Домой почитать, выполнить задания с сайта [learn.javascript.ru](http://learn.javascript.ru)

## About functions:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Predefined\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Predefined_functions)

## Про функции:

<http://learn.javascript.ru/function-is-value>

<http://learn.javascript.ru/function-declaration-expression>

<http://learn.javascript.ru/arguments-pseudarray>

<http://learn.javascript.ru/arguments-named>

## Продвинутое чтение:

<http://dmitrysoshnikov.com/ecmascript/ru-chapter-5-functions/>

# MDN Функции

Источник: <https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Functions>

**Функции - это один из фундаментальных строительных блоков JavaScript.**

Функция - это JavaScript процедура, представляющая собой набор инструкций, которые выполняют какое либо действие или вычисляют значение.

## Объявление функций

Объявление функции состоит из ключевого слова `function`, далее следует:

1. Имя функции;
2. Список аргументов функции, в скобках вида `()` разделенных запятыми (или параметров функции по другому);
3. Инструкции JavaScript которые и составляют тело функции обрамляются, скобками вида `{ }`.

Пример, следующий код объявляет функцию `square`:

```
function square(number) {  
    return number * number;  
}
```

Инструкция `return` указывает значение возвращаемое функцией.

Параметры примитивов (например, такие как числа) передаются в функцию по значению; значение передается функции, но если функция изменит значение параметра, это изменение не будет иметь глобального эффекта.

Если вы передаете в функцию объект (непримитивное значение, такой как Array или определенный пользователем объект) как параметр, и функция изменяет свойства объекта, то эти изменения будут видимы вне функции, как продемонстрировано это в след. примере:

```
function myFunc(theObject) {  
    theObject.make = "Toyota";  
}  
  
var mycar = {make: "Honda", model: "Accord", year: 1998},  
    x,  
    y;  
  
x = mycar.make;    // x gets the value "Honda"  
  
myFunc(mycar);  
y = mycar.make;    // y gets the value "Toyota"  
                  // (the make property was changed by the function)
```

**Метод** это функция являющаяся членом объекта.

Определения функции приведенные выше являются набором синтаксической инструкцией, но функции также могут быть созданы с помощью выражения функции. Такие функции могут быть анонимными; функция не обязательно должна иметь имя. Например функция `square` может быть определена как:

```
var square = function(number) {return number * number};
```

Функции выражения очень удобны когда надо передать в функцию другую функцию как аргумент.

## Вызов функций

Определение функций не выполняет их. Определение функции только дает имя функции и инструкции что делать если функция будет вызвана. Вызов функции в действительности выполняет определенные действия с переданными параметрами.

```
square(5);
```

Предыдущая инструкция вызывает функцию с аргументом равным 5. Функция выполняет свои инструкции и возвращает значение 25. Функции должны быть в области видимости, когда они вызываются.

Аргументы функции не ограничены числами и строками. Вы также можете передавать объекты и функции.

## Область видимости функции

Переменные определенные внутри функции невидимы вне этой функции, так как переменные определяются в области видимости внутри функции. Как бы там ни было сама функция имеет доступ ко всем переменным и другим функциям определенным в той же области видимости где и сама функция была определена. Другими словами, функция определенная в глобальной области видимости имеет доступ ко всем переменным определенным в глобальной области видимости. Функция, определенная внутри другой функции имеет доступ ко всем переменным определенным в функции родителе и любой другой переменной к которой функция родитель имеет доступ.

```
// The following variables are defined in the global scope
var num1 = 20,
    num2 = 3,
    name = "Chamahk";

// This function is defined in the global scope
function multiply() {
    return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore () {
    var num1 = 2,
        num2 = 3;

    function add() {
        return name + " scored " + (num1 + num2);
    }

    return add();
}

getScore(); // Returns "Chamahk scored 5"
```

## Замыкания

Замыкания одна из самых мощных особенностей JavaScript. JavaScript позволяет наследовать функции и, вдобавок, дает полный доступ внутренним функциям ко всем переменным и функциям определенным в функции родителе (и всем переменным и функциям к которым имеет доступ функция родитель). Как бы там ни было, внешняя функция не имеет доступа к переменным и функциям определенным внутри внутренней функции. Это обеспечивает в некотором виде безопасность переменных внутренней функции.

```
var pet = function(name) { // The outer function defines a variable called "name"
    var getName = function() { // The inner function has access to the "name" variable of the outer function
        return name;
    }

    return getName; // Return the inner function, thereby exposing it to outer scopes
},
```

```
myPet = pet("Vivie");

myPet(); // Returns "Vivie"
```

Объект содержащий методы для изменения внутренних переменных внешней функции может быть возвращен.

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex == "string" && (newSex.toLowerCase() == "male" || newSex.toLowerCase() == "female")) {
        sex = newSex;
      }
    }
  }
}

var pet = createPet("Vivie");
pet.getName(); // Vivie

pet.setName("Oliver");
pet.setSex("male");
pet.getSex(); // male
pet.getName(); // Oliver
```

# Функция - это значение

---

Источник: <http://learn.javascript.ru/function-is-value>

Объявление создает функцию и записывает ссылку на неё в переменную.

Функция — не просто значение, это объект.

```
function sayHi(){};
sayHi.test = 5;

alert(sayHi.test); // 5
```

Функцию можно скопировать в другую переменную.

```
function sayHi(person) {
  alert('Привет, ' + person);
}

var func = sayHi;

func('Вася'); // выведет 'Привет, Вася'
sayHi('Маша'); // и так по-прежнему работает: 'Привет, Маша'
```

Так как функция — это объект, то и копируется она «по ссылке».

# Function Declaration и Function Expression

Источник: <http://learn.javascript.ru/function-declaration-expression>

При объявлении функции создаётся переменная со значением-функцией. В частности, невозможно иметь функцию и переменную с одинаковым именем.

## Время создания Function Declaration

Функции, объявленные как Function Declaration, создаются интерпретатором до выполнения кода.

Пример объявления Function Declaration:

```
sayHi("Вася");

function sayHi(name) {
  alert("Привет, " + name);
}
```

Условно объявить функцию через Function Declaration нельзя.

## Объявление Function Expression

Функцию можно создать и присвоить переменной как самое обычное значение. Такое объявление называется Function Expression и выглядит так:

```
var sayHi = function(person) {
  alert("Привет, " + person);
};

sayHi('Вася'); // Привет, Вася
```

```
var arr = [1, 2, function(a) { alert(a) }, 3, 4];
var fun = arr[2];

fun(1); // 1
```

В отличие от объявлений Function Declaration, которые создаются заранее, до выполнения кода, объявления Function Expression создают функцию, когда до них доходит выполнение. Благодаря этому свойству Function Expression можно (и даже нужно) использовать для условного объявления функции.

## Функция с вызовом «на месте»

Такая функция объявляется — и тут же вызывается, вот так:

```
(function() {
  var a = 1, b = 2; // переменные для нашего скрипта
  // код скрипта
})();
```

Задача такой функции обёртки — создать отдельную область видимости для скрипта.

- Если браузер видит function в основном потоке кода - он считает, что это Function Declaration.
- Если же function идёт в составе более сложного выражения, то он считает, что это Function Expression.

Скобки нужны, чтобы показать, что у нас Function Expression, который по правилам JavaScript можно вызвать «на месте». Скобки не нужны, если это и так Function Expression, например в таком вызове:

```
// функция объявлена и тут же вызвана
var res = function(a,b) { return a+b }(2,2);

alert(res); // 4
```

Но при вызове «на месте» лучше ставить скобки и для Expression.

Функция здесь создаётся как часть выражения присваивания, поэтому является Function Expression и может быть вызвана «на месте». При этом, так как сама функция нигде не сохраняется, то она исчезнет, выполнившись, останется только её результат.

## Итого

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

- Если функция объявлена в основном потоке кода, то это Function Declaration.
- Если функция создана как часть выражения, то это Function Expression.

Между этими двумя основными способами создания функций есть следующие различия:

	Function Declaration	Function Expression
Время создания	До выполнения первой строчки кода	Когда управление достигает строки с функцией
Можно вызвать до объявления	Да (т.к. создается заранее)	Нет
Можно объявить в if	Нет (т.к. создается заранее)	Да
Можно вызывать «на месте»	Нет (ограничение синтаксиса JavaScript)	Да

Используйте Function Expression только там, где это действительно нужно. Например, для объявления функции в зависимости от условий.

# Named Function Expression

---

Источник: <http://learn.javascript.ru/named-function-expression>

Обычно то, что называют «именем функции» — на самом деле, является именем переменной, в которую присвоена функция. Если функцию переместить в другую переменную — она сменит «имя». В JavaScript есть способ указать имя, действительно привязанное к функции. Оно называется Named Function Expression (NFE) или, по-русски, именованное функциональное выражение.

Простейший пример NFE выглядит так:

```
var f = function sayHi(...) { /* тело функции */ };
```

Имя функционального выражения (`sayHi`) имеет особый смысл. Оно доступно только изнутри самой функции.

```
var f = function sayHi(name) {  
    alert(sayHi); // изнутри функции - видно (выведет код функции)  
};  
  
alert(sayHi); // снаружи - не видно (ошибка: undefined variable 'sayHi')
```

Ещё одно принципиальное отличие имени от обычной переменной заключается в том, что его нельзя перезаписать.

NFE используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

**Внутреннее имя позволяет функции надёжно обращаться к самой себе, где бы она ни находилась.**



# Псевдо-массив arguments

---

Источник: <http://learn.javascript.ru/arguments-pseudoarray>

Доступ к значениям аргументов осуществляется через «псевдо-массив» `arguments`. Он содержит список аргументов по номерам: `arguments[0]`, `arguments[1]`... , а также свойство `length` .

```
function sayHi() {  
  for (var i = 0; i < arguments.length; i++) {  
    alert("Привет, " + arguments[i]);  
  }  
}  
  
sayHi("Винни", "Пятачок"); // 'Привет, Винни', 'Привет, Пятачок'
```

Если не используется "use strict" внутри функции можно менять параметры, например: `arguments[0] = 5` , но рекомендуется никогда не изменять `arguments`.

# Именованные аргументы

Источник: <http://learn.javascript.ru/arguments-named>

Именованные аргументы не имеют отношения к `arguments`. Это альтернативная техника работы с аргументами, которая позволяет обращаться к ним по имени, а не по номеру.

В JavaScript именованные параметры реализуются при помощи объекта. Вместо списка аргументов передается объект с параметрами, вот так:

```
function showWarning(options) {  
  var width = options.width || 200; // по умолчанию  
  var height = options.height || 100;  
  var title = options.title || "Предупреждение";  
  // ...  
}
```

Вызвать такую функцию очень легко. Достаточно передать объект аргументов, указав в нем только нужные:

```
showWarning({  
  contents: "Вы вызвали функцию",  
  showYesNo: true  
});
```

Еще один бонус кроме красивой записи — возможность повторного использования объекта аргументов:

```
var opts = {  
  width: 400,  
  height: 200,  
  contents: "Текст",  
  showYesNo: true  
};  
  
showWarning(opts);  
opts.contents = "Другой текст";
```

# Задачи

---

## Список задач:

1. Как в функции отличить отсутствующий аргумент от undefined? [READ](#)
2. Напишите функцию `sum(...)`, которая возвращает сумму всех своих аргументов. [READ](#)
3. Функция, которая объединяет несколько строк. [READ](#)

# Как в функции отличить отсутствующий аргумент от undefined?

---

```
function f(x) {  
  // ..ваш код..  
  // выведите 1, если первый аргумент есть, и 0 - если нет  
}  
  
f(undefined); // 1  
f(); // 0
```

## Решение:

Узнать количество реально переданных аргументов можно по значению `arguments.length` :

```
function f(x) {  
  alert(arguments.length ? 1 : 0);  
}  
  
f(undefined);  
f();
```

## Напишите функцию `sum(...)`, которая возвращает сумму всех своих аргументов

---

```
sum(); // 0
sum(1); // 1
sum(1, 2); // 3
sum(1, 2, 3); // 6
sum(1, 2, 3, 4); // 10
```

### Решение:

```
function sum() {
  var result = 0;
  for(var i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }
  return result;
}
```

# Функция, которая объединяет несколько строк

---

Написать функцию, которая по заданному разделителю объединяет несколько строк.

```
// returns "red, orange, blue, "  
myConcat(",", "red", "orange", "blue");  
  
// returns "elephant; giraffe; lion; cheetah; "  
myConcat("; ", "elephant", "giraffe", "lion", "cheetah");  
  
// returns "sage. basil. oregano. pepper. parsley. "  
myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley");
```

## Решение:

```
function myConcat(separator) {  
  var result = "", // initialize list  
      i;  
  // iterate through arguments  
  for (i = 1; i < arguments.length; i++) {  
    result += arguments[i] + separator;  
  }  
  return result;  
}
```

# Функции и области видимости

---

Будем говорить про области видимости, замыкания.

Домой **почитать** (и да, выполнить задание, которые есть на [learn.javascript.ru](http://learn.javascript.ru))

<http://learn.javascript.ru/global-object>

<http://learn.javascript.ru/closures>

<http://learn.javascript.ru/closures-usage>

<http://habrahabr.ru/post/240349/>

# Глобальный объект

Источник: <http://learn.javascript.ru/global-object>

**Глобальными** называют переменные и функции, которые не находятся внутри какой-то функции.

В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object).

В браузере этот объект явно доступен под именем window. Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами window.

```
var a = 5; // объявление var создаёт свойство window.a

alert(window.a); // 5
```

Выполнение скрипта происходит в две фазы:

1. На первой фазе происходит инициализация, подготовка к запуску. Во время инициализации скрипт сканируется на предмет объявления функций вида Function Declaration, а затем — на предмет объявления переменных var. Каждое такое объявление добавляется в window. **Функции, объявленные как Function Declaration, создаются сразу работающими, а переменные — равными undefined.**
2. На второй фазе — собственно, выполнение. Присваивание (=) значений переменных происходит на второй фазе, когда поток выполнения доходит до соответствующей строки кода.

```
// По окончании инициализации, до выполнения кода:
// window = { f: function, a: undefined, g: undefined }

var a = 5; // при инициализации даёт: window.a=undefined

function f(arg) { /*...*/ } // при инициализации даёт: window.f = function

var g = function(arg) { /*...*/ }; // при инициализации даёт: window.g = undefined
```



# Замыкания, функции изнутри

Источник: <http://learn.javascript.ru/closures>

## Лексическое окружение

Все переменные внутри функции — это свойства специального внутреннего объекта **LexicalEnvironment** (лексическое окружение или просто объект переменных).

В отличие от `window`, объект `LexicalEnvironment` является внутренним, он скрыт от прямого доступа.

## Пример:

Посмотрим пример, чтобы лучше понимать, как это работает:

```
function sayHi(name) {  
  var phrase = "Привет, " + name;  
  alert(phrase);  
}  
  
sayHi('Вася');
```

При вызове функции: о выполнения первой строчки её кода, **на стадии инициализации**, интерпретатор создает пустой объект `LexicalEnvironment` и заполняет его. В данном случае туда попадает аргумент `name` и единственная переменная `phrase`:

```
function sayHi(name) {  
  // LexicalEnvironment = { name: 'Вася', phrase: undefined }  
  var phrase = "Привет, " + name;  
  alert(phrase);  
}  
  
sayHi('Вася');
```

**Во время выполнения** происходит присвоение локальной переменной `phrase`, то есть, другими словами, присвоение свойству `LexicalEnvironment.phrase` нового значения:

```
function sayHi(name) {  
  // LexicalEnvironment = { name: 'Вася', phrase: undefined }  
  var phrase = "Привет, " + name;  
  // LexicalEnvironment = { name: 'Вася', phrase: 'Привет, Вася'}  
  alert(phrase);  
}  
  
sayHi('Вася');
```

**В конце выполнения** функции объект с переменными обычно выбрасывается и память очищается.

## Доступ ко внешним переменным

Из функции мы можем обратиться не только к локальной переменной, но и к внешней:

```
var a = 5;  
  
function f() {  
  alert(a); // 5  
}
```

Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем `LexicalEnvironment`, а затем, если её нет — ищет во внешнем объекте переменных. В данном случае им является `window`. Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется `[[Scope]]`.

1. Каждая функция при создании получает ссылку `[[Scope]]` на объект с переменными, в контексте которого была создана. В нашем случае `f. [[Scope]] = window`.
2. При запуске функции создается новый объект с переменными. В него копируется ссылка на внешний объект из `[[Scope]]`.
3. При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом — по этой ссылке. Благодаря этому в функции доступны внешние переменные.

## Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции.

```
function sayHi(person) {  
  
    var message = makeMessage(person);  
    alert(message);  
  
    // ----- вспомогательные функции -----  
  
    function getHello(age) {  
        return age >= 18 ? 'Здравствуйте' : 'Привет';  
    }  
  
    function makeMessage(person) {  
        return getHello(person.age) + ', ' + person.name;  
    }  
}  
  
sayHi({  
    name: 'Петька',  
    age: 17  
}); // привет, Петька
```

Вложенные функции могут быть объявлены и как Function Declaration и как Function Expression.

Вложенные функции обрабатываются в точности так же, как и глобальные. Единственная разница — они создаются в объекте переменных внешней функции, а не в `window`. В примере выше при запуске `sayHi(person)` будет создан такой `LexicalEnvironment`:

```
LexicalEnvironment = {  
    person: переданный аргумент,  
    message: undefined,  
    getHello: function...,  
    makeMessage: function...  
}
```

Вложенная функция имеет доступ к внешним переменным через `[[Scope]]`.

Вложенную функцию можно вернуть. Например, пусть `sayHi` не выдаёт `alert` тут же, а возвращает функцию, которая это делает:

```
function sayHi(person) {  
  
    return function() { // (*)  
        var message = makeMessage(person); // (**)  
        alert(message);  
    };  
};
```

```
// ----- вспомогательные функции -----

function getHello(age) {
    return age >= 18 ? 'Здравствуйте' : 'Привет';
}

function makeMessage() {
    return getHello(person.age) + ', ' + person.name;
}

var sayHiPete = sayHi({ name: 'Петька', age: 17 });
var sayHiOther = sayHi({ name: 'Василий Иванович', age: 35 });

sayHiPete(); // эта функция может быть вызвана позже
```

Возвращаемая функция (\*) при запуске будет иметь полный доступ к аргументам внешней функции, а также к другим вложенным функциям `makeMessage` и `getHello`, так как при создании она получает ссылку `[[Scope]]`, которая указывает на текущий `LexicalEnvironment`. Переменные, которых нет в ней, например, `person`, будут взяты из него.

В частности, функция `makeMessage` при вызове в строке (\*\*) будет взята из внешнего объекта переменных. **Замыканием** функции называется сама эта функция, плюс вся цепочка `LexicalEnvironment`, которая при этом образуется.

Можно сказать и по-другому: **«замыкание - это функция и все внешние переменные, которые ей доступны»**.

## Управление памятью

JavaScript устроен так, что любой объект и, в частности, функция, существует до тех пор, пока на него есть ссылка, пока он как-то доступен для вызова, обращения. Отсюда следует важное следствие при работе с замыканиями.

**Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство `[[Scope]]`.**

## [[Scope]] для new Function

Есть одно исключение из общего правила присвоения `[[Scope]]`.

```
var sum = new Function('a', 'b', 'return a+b;');

var result = sum(1,2);
alert(result); // 3
```

То есть, функция создаётся вызовом `new Function (params, code)`, где:

**params** - параметры функции через запятую в виде строки;

**code** - код функции в виде строки.

При создании функции с использованием `new Function`, её свойство `[[Scope]]` ссылается не на текущий `LexicalEnvironment`, а на `window`.

## Итого

1. Все переменные и параметры функций являются свойствами объекта переменных `LexicalEnvironment`. Каждый запуск функции создает новый такой объект. На верхнем уровне роль `LexicalEnvironment` играет «глобальный объект», в браузере это `window`.
2. При создании функция получает системное свойство `[[Scope]]`, которое ссылается на `LexicalEnvironment`, в котором она была создана (кроме `new Function`).
3. При запуске функции её `LexicalEnvironment` ссылается на внешний, сохраненный в `[[Scope]]`. Переменные сначала ищутся в своём объекте, потом — в объекте по ссылке и так далее, вплоть до `window`.



# Хранение данных в замыкании, модули

Источник: <http://learn.javascript.ru/closures-usage>

В этой главе мы рассмотрим примеры использования замыканий для хранения данных и задачи на эту тему.

## Данные для счётчика

В примере ниже makeCounter создает функцию, которая считает свои вызовы:

```
function makeCounter() {
  var currentCount = 0;

  return function() {
    currentCount++;
    return currentCount;
  };
}

var counter = makeCounter();

// каждый вызов увеличивает счётчик
counter();
counter();
alert(counter()); // 3
```

Хранение текущего числа вызовов осуществляется в переменной currentCount внешней функции.

При этом, так как каждый вызов makeCounter создает новый объект переменных, то все создаваемые функции-счётчики взаимно независимы.

```
var c1 = makeCounter();

var c2 = makeCounter();

alert(c1()); // 1
alert(c2()); // 1, счётчики независимы
```

Добавим счётчику аргумент, который, если передан, устанавливает значение:

```
function makeCounter() {
  var currentCount = 0;

  return function(newCount) {
    if (newCount !== undefined) { // есть аргумент?
      currentCount = +newCount; // сделаем его новым значением счётчика
      // вернём текущее значение, счётчик всегда возвращает его (это удобно)
      return currentCount;
    }

    currentCount++;
    return currentCount;
  };
}

var counter = makeCounter();

alert(counter()); // 1
alert(counter(3)); // 3
alert(counter()); // 4
```

## Объект счётчика

Можно пойти дальше и вернуть полноценный объект с функциями управления счётчиком:

- **getNext()** — получить следующее значение, то, что раньше делал вызов `counter()` ;
- **set(value)** — поставить значение;
- **reset()** — обнулить счётчик.

```
function makeCounter() {
  var currentCount = 0;

  return {
    getNext: function() {
      return ++currentCount;
    },

    set: function(value) {
      currentCount = value;
    },

    reset: function() {
      currentCount = 0;
    }
  };
}

var counter = makeCounter();

alert(counter.getNext()); // 1
alert(counter.getNext()); // 2

counter.reset();
alert(counter.getNext()); // 1
```

Теперь `counter` — объект с методами, которые при работе используют `currentCount` . Снаружи никак иначе, кроме как через эти методы, к `currentCount` получить доступ нельзя, так как это локальная переменная.

## Объект счётчика + функция

К сожалению, пропал короткий красивый вызов `counter()` , вместо него теперь `counter.getNext()` . Но он ведь был таким коротким и удобным... Так что давайте вернём его:

```
function makeCounter() {
  var currentCount = 0;

  // возвращаемся к функции
  function counter() {
    return ++currentCount;
  }

  // ...и добавляем ей методы!
  counter.set = function(value) {
    currentCount = value;
  };

  counter.reset = function() {
    currentCount = 0;
  };

  return counter;
}

var counter = makeCounter();

alert(counter()); // 1
alert(counter()); // 2

counter.reset();
alert(counter()); // 1
```

Тот факт, что объект — функция, вовсе не мешает добавить к нему сколько угодно методов.

## Приём проектирования «Модуль»

Цель приема - объявить функции так, чтобы ненужные подробности их реализаций были скрыты. В том числе: временные переменные, константы, вспомогательные мини-функции и т.п. Оформление кода в модуль предусматривает следующие шаги:

1. Создаётся функция-обёртка, которая выполняется «на месте»:

```
(function() {  
  ...  
})();
```

2. Внутри этой функции пишутся локальные переменные и функции, которые пользователю модуля не нужны, но нужны самому модулю:

```
(function() {  
  var count = 0;  
  
  function helper() { ... }  
})();
```

Они будут доступны только изнутри.

3. Те функции, которые нужны пользователю, «экспортируются» во внешнюю область видимости. Если функция одна — это можно сделать явным возвратом `return` :

```
var func = (function() {  
  var count = 0;  
  function helper() { ... }  
  
  return function() { ... }  
})();
```

Если функций много — можно присвоить их напрямую в `window` :

```
(function() {  
  
  function helper() { ... }  
  
  window.createMenu = function() { ... };  
  window.createDialog = function() { ... };  
})();
```

Или, что ещё лучше, вернуть объект с ними:

```
var MyLibrary = (function() {  
  
  function helper() { ... }  
  
  return {  
    createMenu: function() { ... },  
    createDialog: function() { ... }  
  };  
})();  
// использование  
MyLibrary.createMenu();
```

Все функции модуля будут через замыкание иметь доступ к другим переменным и внутренним функциям. Но снаружи программист, использующий модуль, может обращаться напрямую только к тем, которые экспортированы.



# Выразительный JavaScript: Функции

Источник: <http://habrahabr.ru/post/240349/>

## Определение функции

**Определение функции** — обычное определение переменной, где значение, которое получает переменная, является функцией.

Например, следующий код определяет переменную `square`, которая ссылается на функцию, подсчитывающую квадрат заданного числа:

```
var square = function(x) {  
    return x * x;  
};  
  
console.log(square(12)); // → 144
```

Функция создаётся выражением, начинающимся с ключевого слова `function`. У функций есть набор параметров (в данном случае, только `x`), и тело, содержащее инструкции, которые необходимо выполнить при вызове функции. Тело функции всегда заключают в фигурные скобки, даже если оно состоит из одной инструкции.

У функции может быть несколько параметров, или вообще их не быть. В следующем примере `makeNoise` не имеет списка параметров, а у `power` их целых два:

```
var makeNoise = function() {  
    console.log("Хрясь!");  
};  
  
makeNoise(); // → Хрясь!
```

```
var power = function(base, exponent) {  
    var result = 1;  
    for (var count = 0; count < exponent; count++)  
        result *= base;  
    return result;  
};  
  
console.log(power(2, 10));  
// → 1024
```

Некоторые функции возвращают значение, как `power` и `square`, другие не возвращают, как `makeNoise`, которая производит только побочный эффект. Инструкция `return` определяет значение, возвращаемое функцией. Когда обработка программы доходит до этой инструкции, она сразу же выходит из функции, и возвращает это значение в то место кода, откуда была вызвана функция. `return` без выражения возвращает значение `undefined`.

## Параметры и область видимости

**Параметры функции** — такие же переменные, но их начальные значения задаются при вызове функции, а не в её коде.

Важное свойство функций в том, что переменные, созданные внутри функции (включая параметры), локальны внутри этой функции. Это означает, что в примере с `power` переменная `result` будет создаваться каждый раз при вызове функции, и эти отдельные её инкарнации никак друг с другом не связаны.

Эта локальность переменных применяется только к параметрам и созданным внутри функций переменным.

Переменные, заданные снаружи какой бы то ни было функции, называются глобальными, поскольку они видны на протяжении всей программы. Получить доступ к таким переменным можно и внутри функции, если только вы не объявили локальную переменную с тем же именем.

Следующий код иллюстрирует это. Он определяет и вызывает две функции, которые присваивают значение переменной `x`. Первая объявляет её как локальную, тем самым меняя только локальную переменную. Вторая не объявляет, поэтому работа с `x` внутри функции относится к глобальной переменной `x`, заданной в начале примера.

```
var x = "outside";

var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x); // → outside

var f2 = function() {
  x = "inside f2";
};
f2();
console.log(x); // → inside f2
```

## Вложенные области видимости

JavaScript различает не только глобальные и локальные переменные. Функции можно задавать внутри функций, что приводит к нескольким уровням локальности.

В каждой локальной области видимости можно увидеть все области, которые её содержат. Набор переменных, доступных внутри функции, определяется местом, где эта функция описана в программе. Все переменные из блоков, окружающих определение функции, видны — включая и те, что определены на верхнем уровне в основной программе. Этот подход к областям видимости называется лексическим.

## Функции как значения

Имена функций обычно используют как имя для кусочка программы. Такая переменная однажды задаётся и не меняется. Так что легко перепутать функцию и её имя.

Но это — две разные вещи. Вызов функции можно использовать, как простую переменную — например, использовать их в любых выражениях. Возможно хранить вызов функции в новой переменной, передавать её как параметр другой функции, и так далее. Также переменная, хранящая вызов функции, остаётся обычной переменной и её значение можно поменять.

## Объявление функций

Есть более короткая версия выражения `var square = function...`. Ключевое слово `function` можно использовать в начале инструкции:

```
function square(x) {
  return x * x;
}
```

Это объявление функции. Инструкция определяет переменную `square` и присваивает ей заданную функцию.

## Необязательные аргументы

Официально функция принимает один аргумент. Однако, при таком вызове она не жалуется. Она игнорирует остальные аргументы и показывает «Здрасьте».

JavaScript очень лоялен по поводу количества аргументов, передаваемых функции. Если вы передадите слишком много, лишние будут проигнорированы. Слишком мало – отсутствующим будет назначено значение `undefined`.

## Замыкания

В примере объявляется функция `wrapValue`, которая создаёт локальную переменную. Затем она возвращает функцию, которая читает эту локальную переменную и возвращает её значение.

```
function wrapValue(n) {
  var localVariable = n;
  return function() { return localVariable; };
}

var wrap1 = wrapValue(1);
var wrap2 = wrapValue(2);

console.log(wrap1()); // → 1
console.log(wrap2()); // → 2
```

Это допустимо и работает так, как должно – доступ к переменной остаётся. Более того, в одно и то же время могут существовать несколько экземпляров одной и той же переменной, что ещё раз подтверждает тот факт, что с каждым вызовом функции локальные переменные пересоздаются.

Эта возможность работать со ссылкой на какой-то экземпляр локальной переменной называется **замыканием**. Функция, замыкающая локальные переменные, называется **замыкающей**. Она не только освобождает вас от забот, связанных с временем жизни переменных, но и позволяет творчески использовать функции.

С небольшим изменением мы превращаем наш пример в функцию, умножающую числа на любое заданное число.

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

var twice = multiplier(2);
console.log(twice(5)); // → 10
```

Отдельная переменная вроде `localVariable` из примера с `wrapValue` уже не нужна. Так как параметр – сам по себе локальная переменная.

Потребуется практика, чтобы начать мыслить подобным образом. Хороший вариант мысленной модели – представлять, что функция замораживает код в своём теле и обёртывает его в упаковку. Когда вы видите `return function(...) {...}`, представляйте, что это пульт управления куском кода, замороженным для употребления позже.

В нашем примере `multiplier` возвращает замороженный кусок кода, который мы сохраняем в переменной `twice`. Последняя строка вызывает функцию, заключённую в переменной, в связи с чем активируется сохранённый код (`return number * factor;`). У него всё ещё есть доступ к переменной `factor`, которая определялась при вызове `multiplier`, к тому же у него есть доступ к аргументу, переданному во время разморозки (`5`) в качестве числового параметра.

## Рекурсия

Функция вполне может вызывать сама себя, если она заботится о том, чтобы не переполнить стек. Такая функция называется рекурсивной. Вот пример альтернативной реализации возведения в степень:

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
```

```
    return base * power(base, exponent - 1);  
  }  
  
  console.log(power(2, 3)); // → 8
```

## Выращиваем функции

Существует два более-менее естественных способа ввода функций в программу.

Первый – вы пишете схожий код несколько раз. Этого нужно избегать – больше кода означает больше места для ошибок и больше материала для чтения тех, кто пытается понять программу. Так что мы берём повторяющуюся функциональность, подбираем ей хорошее имя и помещаем её в функцию.

Второй способ – вы обнаруживаете потребность в некоей новой функциональности, которая достойна помещения в отдельную функцию. Вы начинаете с названия функции, и затем пишете её тело. Можно даже начинать с написания кода, использующего функцию, до того, как сама функция будет определена.

## Функции и побочные эффекты

Функции можно грубо разделить на те, что вызываются из-за своих побочных эффектов, и те, что вызываются для получения некоторого значения. Конечно, возможно и объединение этих свойств в одной функции.

**Чистая функция** – особый вид функции, возвращающей значения, которая не только не имеет побочных эффектов, но и не зависит от побочных эффектов остального кода – к примеру, не работает с глобальными переменными, которые могут быть случайно изменены где-то ещё. Чистая функция, будучи вызванной с одними и теми же аргументами, возвращает один и тот же результат (и больше ничего не делает) – что довольно приятно. С ней просто работать. Вызов такой функции можно мысленно заменять результатом её работы, без изменения смысла кода. Когда вы хотите проверить такую функцию, вы можете просто вызвать её, и быть уверенным, что если она работает в данном контексте, она будет работать в любом. Не такие чистые функции могут возвращать разные результаты в зависимости от многих факторов, и иметь побочные эффекты, которые сложно проверять и учитывать.

Однако, не надо стесняться писать не совсем чистые функции, или начинать священную чистку кода от таких функций. Побочные эффекты часто полезны. Нет способа написать чистую версию функции `console.log`, и эта функция весьма полезна. Некоторые операции легче выразить, используя побочные эффекты.

## Итого

Эта глава показала вам, как писать собственные функции. Когда ключевое слово `function` используется в виде выражения, возвращает указатель на вызов функции. Когда оно используется как инструкция, вы можете объявлять переменную, назначая ей вызов функции.

```
// Создаём f со ссылкой на функцию  
var f = function(a) {  
  console.log(a + 2);  
};  
  
// Объявляем функцию g  
function g(a, b) {  
  return a * b * 3.5;  
}
```

Ключевой момент в понимании функций – локальные области видимости. Параметры и переменные, объявленные внутри функции, локальны для неё, пересоздаются каждый раз при её вызове, и не видны снаружи. Функции, объявленные внутри другой функции, имеют доступ к её области видимости.

Очень полезно разделять разные задачи, выполняемые программой, на функции. Вам не придётся повторяться, функции делают код более читаемым, разделяя его на смысловые части, так же, как главы и секции книги помогают в организации обычного текста.



# Конспект. Функции и области видимости

Источник: <http://forum.jscourse.com/t/05-konspekt-funkczii/584>

Автор конспекта: @Nikolay

Правки: @dmitry

## Функции (описание)

**Функции** - это некоторые действия. Функция позволяет структурировать программу, разбивать ее на маленькие части (подпрограммы) и таким образом организовывать код.

При создании функции необходимо ее реализовывать так, чтобы функция имела как можно меньше побочных эффектов.

Если можно избежать в функции обращение к внешним данным (находящимся за пределами функции), и реализовать все через аргументы и возвращаемые значения, то функцию следует реализовывать именно таким образом.

**Абстракция** - этим термином описывают технику, когда за одним действием скрывается несколько других действий.

## Вызов функции

При создании функции, к примеру `f`, мы можем вызвать ее с помощью `console.log(f())`:

```
function f() {  
  return 999;  
}  
console.log(f()); // 999
```

Также мы можем записать эту функцию в переменную:

```
var a = f;
```

Присвоение, сравнение функций происходит "по ссылке". То есть переменная `a` и переменная `f`, ссылаются на одну и ту же функцию.

Если мы обратимся к переменной `a` или `f`, то они вычисляются в ту функцию, в которую ссылаются `a`

```
console.log(a === f); // true
```

Чтобы вызвать функцию следует использовать круглые скобки:

```
a();
```

## Функции - данные

**Функция** - это значения, и поэтому мы можем передавать их в качестве аргументов в другие функции. Такая техника используется часто в асинхронном программировании и при работе с API.

Создадим функцию которая вызывает аргумент 10 раз:

```
function run10Times(func) { // принимает аргументом объект функции (func)
  for (var i = 0; i < 10; i += 1) { // вызывает объект этой функции 10 раз
    func();
  }
}
```

Объявляем функцию `run10Times` и передаем в качестве аргумента функцию `f` (а если точнее - ссылку на функцию).

```
function f(timeCalled) {
  console.log(999, timeCalled);
}

run10Times(f); // <<
```

Когда вызывается функция с аргументами, и подставляем другую функцию/объект (`f`) в качестве аргумента, то первым делом запускается тело этой (передаваемой) функции (`f`), и те аргументы которые были подставлены при вызове записываются в переменные которые были объявлены при объявлении функции.

То есть, внутри `run10Times` есть переменная `func`, которая ссылается на ту же функцию, что и `f`.

## Объявление функции

### Function declaration

```
function f() {
  console.log(arguments);
}
```

функция может быть вызвана и работать с любым количеством параметров:

```
f('mama'); // [ "mama" ]
f('mama', 'mila'); // ["mama", "mila"]
```

Об аргументах функции удобно думать следующим образом: "Когда функция `f` будет вызвана, то в переменную `first` внутри функции будет записано то значение, которое передано в качестве первого аргумента при вызове".

**Примитивные типы данных (строки, числа, булевые) передается аргументами в функцию по значению.**

**Объекты, массивы, функции передаются по ссылке(!).** Если функцию объявлена с 3-я параметрами, а при вызове одно из значений не было передано, то в значение непереданных аргументом будет записано значение `undefined`.

```
function f(first, second, third) {
  console.log(first, second, third);
  console.log('arguments', arguments);
}
f('first'); // "first" undefined undefined
           //arguments ["first"]
```

## Аргументы и функции

Объект `arguments` создается в момент вызова функции.

**arguments** является "массивоподобным" объектом (имеет числовой индекс и обладает свойством `length` (и только)).

Если нужно работать с `arguments`, как с массивом (вызвать метод массива) объект `arguments` нужно превратить в массив. При этом по `arguments` все так же можно итерировать циклом `for`. Для преобразования массивоподобного объекта в массив можно использовать **абстракцию**.

```
// функция которая преобразует объект arguments в массив
function objectToArray(arrayLikeObject) {
    var res = [];
    for (var i = 0; i < arrayLikeObject.length; i += 1) {
        res.push(arrayLikeObject[i]);
    }
    return res;
}

//рабочая функция
function f(first, second, third) {
    var args = objectToArray(arguments); // обращаемся к функции arrayLikeObject для преобразования объекта arguments в массив
    console.log('undeinfed index', args.indexOf(undefined));
}

// Результат
f('mama', 'mila', undefined); // "undeinfed index" 2
```

## Анонимные функции

Напишем функцию `forEach` которая будет принимать аргументом массив и функцию `action` (пример с книги Eloquent JS). Функция `forEach` вызывает для каждого элемента массива функцию `action`, и передает туда *i*-тый аргумент массива. (далее рассмотрим как реализовать аналогичную программу с использованием анонимной функции)

```
function forEach(array, action) {
    for (var i = 0; i < array.length; i++) {
        action(array[i]);
    }
}

function log(val) {
    console.log(val);
}
forEach(["Тили", "Мили", "Трямдия"], log)

Output:
"Тили"
"Мили"
"Трямдия"
```

Следует обратить внимание, что в параметр `action` записывается значение второго аргумента, с которым вызывается `forEach` (а именно `log`), а аргумент `log` в свою очередь ссылается на функцию `log`. Значит - в параметр `action` записывается ссылка на функцию `log`.

Теперь вызовем функцию `log` используя анонимную функцию (function first-class objects)

```
forEach(["Тили", "Мили", "Трямдия"], function (val) {
    console.log(val, fff);
});
```

То есть мы объявляем и передаем функцию на месте. **Следует запомнить, что литеральная запись массива никак не отличается от литеральной записи функции.**

Также функция может храниться в объекте:



```
var ppp = {  
  log: function (val) {  
    console.log('THIS FUNC');  
    console.log(val);  
  }  
};
```

И мы ссылаемся на нее следующим образом:

```
forEach(["Тили", "Мили", "Трямдия"], ppp.log);
```

# Задачи

---

## 1. Каков будет результат кода?

```
if ("a" in window) {  
    var a = 1;  
}  
  
alert(a);
```

### Ответ:

Одному. Посмотрим, почему.

На стадии подготовки к выполнению, из `var a` создается `window.a` :

```
// window = {a:undefined}  
  
if ("a" in window) { // в if видно что window.a уже есть  
    var a = 1; // поэтому эта строка сработает  
}  
  
alert(a);
```

В результате `a` становится 1.

## 2. Каков будет результат (перед a нет var)?

```
if ("a" in window) {  
    a = 1;  
}  
  
alert(a);
```

### Ответ:

Ошибка.

Переменной `a` нет, так что условие `"a" in window` не выполнится. В результате на последней строчке - обращение к неопределенной переменной.

```
if ("a" in window) {  
    a = 1;  
}  
  
alert(a); // <-- error!
```

## 3. Каков будет результат (перед a нет var, а ниже есть)?

```
if ("a" in window) {  
    a = 1;  
}  
var a;  
  
alert(a);
```

## Ответ:

Одному.

Переменная `a` создается до начала выполнения кода, так что условие `"a" in window` выполнится и сработает `a = 1`.

```
if ("a" in window) {  
  a = 1;  
}  
var a;  
  
alert(a); // 1
```

## 4. Каков будет результат кода? Почему?

```
var a = 5;  
function a() { }  
  
alert(a);
```

## Ответ:

5.

Чтобы понять, почему — разберём внимательно как работает этот код.

1. До начала выполнения создаётся переменная `a` и функция `a`. Стандарт написан так, что функция создаётся первой и переменная ее не перезаписывает. То есть, функция имеет приоритет.
2. После инициализации, когда код начинает выполняться — срабатывает присваивание `a = 5`, перезаписывая `a`, и уже не важно, что там лежало.
3. Объявление Function Declaration на стадии выполнения игнорируется (уже обработано).
4. В результате `alert(a)` выводит 5.

## 5. Каков будет результат выполнения этого кода?

```
var value = 0;  
  
function f() {  
  if (1) {  
    value = true;  
  } else {  
    var value = false;  
  }  
  alert(value);  
}  
  
f();
```

Изменится ли внешняя переменная `value` ?

P.S. Какими будут ответы, если из строки `var value = false` убрать `var` ?

## Ответ:

Результатом будет true, т.к. `var` обработается и переменная будет создана до выполнения кода.

Соответственно, присвоение `value = true` работает на локальной переменной, и `alert` выведет `true`.

Внешняя переменная не изменится.

P.S. Если `var` нет, то в функции переменная не будет найдена. Интерпретатор обратится за ней в `window` и изменит её там.

Так что без `var` результат будет также `true`, но внешняя переменная изменится.

## 6. Каков будет результат выполнения этого кода? Почему?

```
function test() {  
  alert(window);  
  var window = 5;  
  alert(window);  
}  
  
test();
```

### Ответ:

Результатом будет `undefined`, затем `5`.

Директива `var` обработается до начала выполнения кода функции. Будет создана локальная переменная, т.е. свойство `LexicalEnvironment`:

```
LexicalEnvironment = {  
  window: undefined  
}
```

Когда выполнение кода начнется и сработает `alert`, он выведет локальную переменную. Затем сработает присваивание, и второй `alert` выведет уже `5`.

## 7. Каков будет результат выполнения кода? Почему?

```
var a = 5  
  
(function() {  
  alert(a)  
})();
```

P.S. Подумайте хорошо! Здесь все ошибаются! P.P.S. Внимание, здесь подводный камень! Ок, вы предупреждены.

### Ответ:

Результат - ошибка.

Дело в том, что после `var a = 5` нет точки с запятой.

JavaScript воспринимает этот код как если бы перевода строки не было:

```
var a = 5(function() {  
  alert(a)  
})();
```

То есть, он пытается вызвать функцию `5`, что и приводит к ошибке.

Если точку с запятой поставить, все будет хорошо. Это один из наиболее частых и опасных подводных камней, приводящих к ошибкам тех, кто не ставит точки с запятой.

## 8. Еще задача на понятие замыканий

1. Будет ли работать доступ к переменной `name` через замыкание в примере ниже?
2. Удалится ли переменная `name` из памяти при выполнении `delete donkey.sayHi` ? Если нет — можно ли к `name` как-то обратиться после удаления `donkey.sayHi` ?
3. А если присвоить `donkey.sayHi = donkey.yell = null` — останется ли `name` в памяти?

```
var makeDonkey = function() {  
  var name = "Ослик Иа";  
  
  return {  
    sayHi: function() {  
      alert(name);  
    },  
    yell: function() {  
      alert('И-а, и-а!');  
    }  
  };  
}  
var donkey = makeDonkey();  
donkey.sayHi();
```

### Решение:

1. Да, будет работать, благодаря ссылке `[[Scope]]` на внешний объект переменных, которая будет присвоена функциям `sayHi` и `yell` при создании объекта.
2. Нет, `name` не удалится из памяти, поскольку несмотря на то, что `sayHi` больше нет, есть ещё функция `yell` , которая также ссылается на внешний объект переменных. Этот объект хранится целиком, вместе со всеми свойствами. При этом, так как функция `sayHi` удалена из объекта и ссылок на нее нет, то больше к переменной `name` обращаться некому. Получилось, что она «застряла» в памяти, хотя, по сути, никому не нужна.
3. Если и `sayHi` и `yell` удалить, тогда, так как больше внутренних функций не останется, удалится и объект переменных вместе с `name` .

# Напишите функцию `sum`, которая работает так: `sum(a)(b) = a + b`.

---

Да, именно так, через двойные скобки. Например:

```
sum(1)(2); // 3
sum(5)(-1); // 4
```

## Решение:

Чтобы вторые скобки в вызове работали - первые должны возвращать функцию.

Эта функция должна знать про `a` и уметь прибавлять `a` к `b`. Вот так:

```
function sum(a) {
  return function(b) {
    return a + b; // возьмет a из внешнего LexicalEnvironment
  };
}

alert(sum(1)(2));
alert(sum(5)(-1));
```

# Функция filter(arr, func)

1. Создайте функцию `filter(arr, func)`, которая получает массив `arr` и возвращает новый, в который входят только те элементы `arr`, для которых `func` возвращает `true`.
2. Создайте набор «готовых фильтров»: `inBetween(a,b)` – «между a, b», `inArray([...])` – «в массиве [...]». Использование должно быть таким:
  - Функция `filter(arr, inBetween(3,6))` – выберет только числа от 3 до 6,
  - Функция `filter(arr, inArray([1,2,3]))` – выберет только элементы, совпадающие с одним из значений массива.

Пример, как это должно работать:

```
/* .. ваш код для filter, inBetween, inArray */
var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, function(a) { return a % 2 == 0 }) ); // 2,4,6
alert( filter(arr, inBetween(3,6)) ); // 3,4,5,6
alert( filter(arr, inArray([1,2,10])) ); // 1,2
```

## Решение:

Функция фильтрации:

```
function filter(arr, func) {
  var result = [];
  for(var i = 0; i < arr.length; i++) {
    var val = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }
  return result;
}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, function(a) { return a % 2 == 0; }) ); // 2, 4, 6
```

Фильтр inBetween:

```
function filter(arr, func) {
  var result = [];
  for(var i = 0; i < arr.length; i++) {
    var val = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }
  return result;
}

function inBetween(a, b) {
  return function(x) {
    return x >= a && x <= b;
  };
}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert(filter(arr, inBetween(3,6))); // 3,4,5,6
```

Фильтр inArray:

```
function filter(arr, func) {
  var result = [];
  for(var i = 0; i < arr.length; i++) {
    var val = arr[i];
    if (func(val)) {
      result.push(val);
    }
  }
  return result;
}

function inArray(arr) {
  return function(x) {
    return arr.indexOf(x) != -1;
  };
}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, inArray([1,2,10])) ); // 1,2
```



# Функции. Продолжение

---

Продолжаем говорить о функциях, решать задачи.

**Почитать:**

Про строгий режим: <http://dmitrypodgorniy.com/blog/2012/07/09/1/>

Еще про замыкания:

(рассматривалось в главе 4, пункт 4.1 MDN Функции)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Function\\_scope](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Function_scope)

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Closures>

И еще про замыкания: <http://dmitrysoshnikov.com/ecmascript/ru-chapter-6-closures/#primeneniye-zamyikaniy>

Методы массивов, принимающие аргументом функцию (хорошо потренировать их на пройденных домашних заданиях).

[https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/Array/filter](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

[https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

[https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/Array/reduce](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)

# Строгий режим "use strict"

Источник: <http://dmitrypodgorniy.com/blog/2012/07/09/1/>

Автор: Дмитрий Подгорный

## С кодом будь строг. Интерпретатор

Строгий контроль написания кода уменьшает количество потенциальных ошибок. javascript допускает многие вольности, при том такие, которые могут породить трудно обнаруживаемые ошибки. Тут на помощь приходят редакторы и интерпретаторы. На уровне редакторов есть несколько инструментов, статического анализа кода, а со стороны браузеров доступен строгий режим выполнения javascript кода.

### Use strict

Ниже — примеры того, где строгий режим генерирует ошибки (полное описание отличий строгого режима). Надо отметить, что генерация ошибок происходит на этапе чтения исходного файла. Это значит, что функции, объявленные в нестрогой области видимости будут без проблем вызваны из «строгих функций». Так же не рекомендуется использовать строгий режим глобально, потому что вероятнее всего он сломает 3-d party код.

Нельзя присвоить значение необъявленной переменной. Код ниже в строгом режиме вместо создания глобальной переменной выдаст ошибку.

```
// Нестрогое поведение
function f () {
  a = 10;
}
f();
// Неожиданная глобальная переменная
console.log(a); // 10

// Строгое поведение
function f () {
  'use strict';
  a = 10;
}
f(); // ошибка
```

Запрещено дублирование имен свойств объекта.

```
(function () {
  var obj = {
    prop : true,
    prop : false
  }

  console.warn(obj); // {prop:false}
})();

(function () {
  'use strict';

  var obj = {
    prop : true,
    prop : false
  }

  console.warn(obj); // error
})();
```

Объект `arguments` — неизменяемый. Проблема заключается, в том, что изменяя `arguments`, изменение коснутся и

аргументов, связанных с переменными, и самое неприглядное то, что примитивные значения тоже изменятся.

```
function f (a, b) {  
  arguments[0] = 10;  
  console.log(a, b);  
}  
  
f(1, 99); // 10, 99
```

Запрещено использование `arguments.callee`, и `arguments.callee.caller`. Говорят, что из-за наличия такого кода, интерпретатор не может оптимизировать код. Вместо этих конструкций рекомендуется использовать Named Function Expression.

```
(function waiter () {  
  if (condition()) {  
    action();  
  } else {  
    setTimeout(waiter, 50); // используем имя вместо arguments.callee  
  }  
})();
```

# Конспект. Функции, области видимости, замыкания

Источник: <http://forum.jscourse.com/t/06-konspekt-funkczii-oblasti-vidimosti-zamykaniya/563>

Автор: @dmitry

## Области видимости

**Области видимости** это такая фича, которая ограничивает доступность тех или иных переменных. Область видимости бывает глобальная (в ней начинают выполняться все скрипты) или уровня функции. Каждая функция, когда вызывается, создаёт свою область видимости. Допустим, мы подключили к странице такой скрипт:

```
var a = 10
function f(x,y,z){
  // x,y,z
  // arguments
}
f();
```

Переменные `a`, `f` находятся в глобальной области видимости. Каждый раз когда интерпретатор заходит в тело функции, создаётся новая область видимости для функции. Например, когда интерпретатор заходит в функцию `f`, он создаёт область видимости, в которой есть переменные `x`, `y`, `z`, так же объект `arguments`. Когда мы заходим во вложенную область видимости, нам доступны переменные из родительских областей видимостей, соответственно можно получить переменную `a` из родительской (в данном случае она ещё и глобальная)

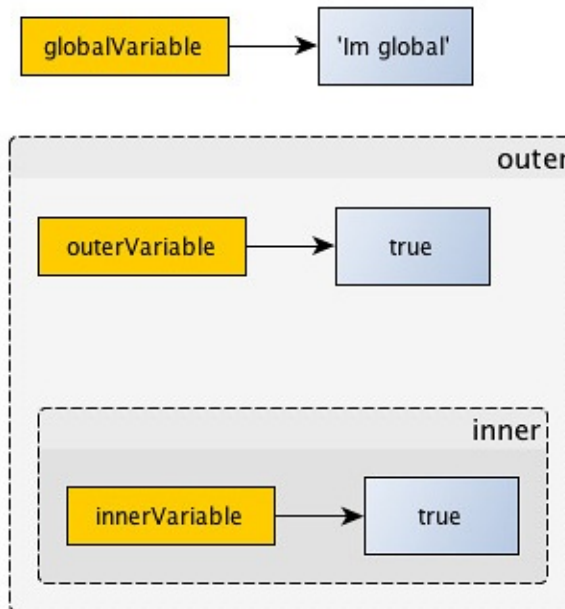
Рассмотрим этот код и картинку:

```
var globalVariable = 'Im global';
function outer() {
  var outerVariable = true;

  function inner() {
    var innerVariable = true;
    console.log(outerVariable, innerVariable, globalVariable);
    console.log(ppp); //
  }

  window.inner = inner;
  console.log(outerVariable); // true
  console.log(innerVariable); // referenceError
  inner();
}
```

У нас есть некий скрипт. В глобальной области видимости создаётся переменная `globalVariable` с текстовым значением и переменная `outer`, где хранится ссылка на функцию. Каждый раз когда интерпретатор выполняет функцию `outer` (большая часть площади картинки), он вынужден создать область видимости `inner`. **Не имеет значения, в какой области видимости функция вызывается, имеет значение только место где она объявлена.**



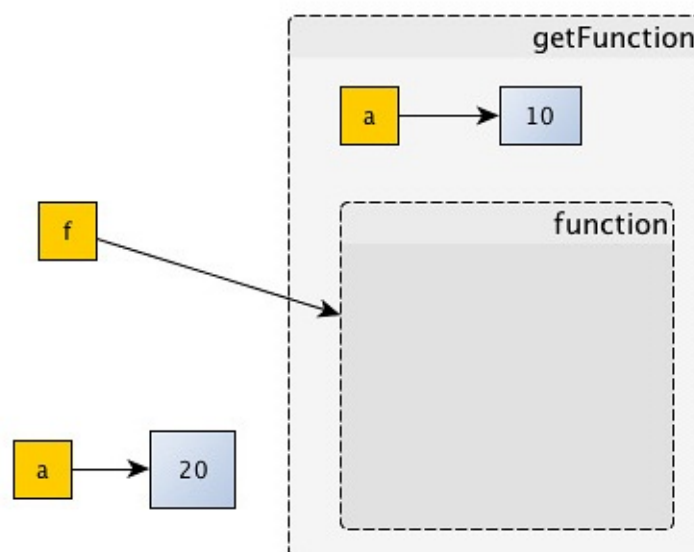
Рассмотрим ещё один случай.

```

var a = 20;
function getFunction() {
    var a = 10;
    return function () {
        console.log(a);
    };
}

var f = getFunction();
f(); // 10
  
```

В этом коде мы объявляем функцию внутри функции, причём внутренняя функция попадает в другую область видимости за пределами той, где она была объявлена. У нас есть код в глобальной области видимости, он создаёт переменную `a`, так же создаётся `getFunction`, результат работы которой объект функции, который создаётся внутри функции `getFunction`. Когда мы вызываем функцию `getFunction`, она вычисляется в новую функцию, при этом тело новой функции объявляется внутри области видимости функции `getFunction`.



Если смотреть на картинку, у нас есть глобальная область видимости (белый цвет), в которой есть `a = 20`, есть функция `getFunction`, которая каждый раз при вызове создаёт область видимости и внутри неё создаёт `a = 10`. Так же, внутри `getFunction` объявляется новая функция, при этом для вложенной анонимной функции `f` это переменная, которая объявлена в глобальной области видимости с ссылкой на функцию внутри `getFunction`.

Scope chain определяется тем местом, где функция была объявлена. Так, как внутренняя функция была объявлена внутри `getFunction`, то при попытке найти а первая переменная с именем `a` находится в родительском scope - `getFunction`. Можно сказать что функция замыкает на себя эту переменную.

**Замыкание** это функция или ссылка на функцию вместе с окружением. Замыкание, в отличии от обычной функции, умеет ссылаться на переменные, находящиеся за пределами области видимости, ограниченной функцией.

Следствие лексической области видимости заключается в том, что все переменные которые объявлены внутри функции de-facto объявляются в начале функции (в отличии от блочной области видимости, которая характерна для таких языков как Java, C++, и где область видимости переменной определена блоком, например `{ }`).

Любая функция является демонстрацией замыкания. На самом деле нету никакого смысла отделять понятие замыкания от того факта объявления функции. Сами правила по которым работает область видимости (см. выше) определяют что такое замыкание.

Рассмотрим детально код ниже:

```
(function () {
  'use strict';

  var privateVariable;

  function doSomething() { }

  window.uberPlugin = function (userAction) {
    doSomething();
    userAction(privateVariable);
  };
})();
```

1. Создана и вызвана на месте анонимная функция, автоматически создана область видимости которая соответствует телу этой функции.
2. Ссылка на функцию `uberPlugin` передана в другой scope (другую область видимости, в данном случае - глобальную);
3. Функция `uberPlugin` имеет доступ ко всем переменным `privateVariable` внутри анонимной функции и к функциям `doSomething`. Соответственно она может их использовать.

Передаем в функцию, которая объявлена внутри анонимной (а именно - `uberPlugin`) функцию `(function (pluginInternal) {})` (см. ниже) Функция `function (pluginInternal) {}` вызывается в внутренней переменной, которую видит только функция `function (userAction){}`. И единственный способ к ней подступиться - это передать свою функцию которая будет вызвана `uberPlugin` вместе в аргументом

```
uberPlugin(function (pluginInternal) {
  console.warn('pluginInternal ', pluginInternal);
}); // pluginInternal undefined
```

## Shadowing

**Shadowing** ("перекрывтие" или "сокрытие" переменных) - это процесс, когда в одной из вложенных областей видимости создаются переменные с тем же названием, как и во внешних областях видимости.

```
function outer (a, b) {
  var ars = arguments;
  function inner () {
    var a = 10; // shadow parent scope variable
    console.log('inner', a, ars);
  }
  inner();
  console.log('outer', a);
}
```

```
}  
outer(100500); // inner 10 , outer 100500
```

Такой подход может привести к ошибкам и поэтому рекомендуется использовать разные названия переменных.

## Паттерны анонимной функции

Есть два часто используемых паттерна анонимной функции.

1 ) **Использование модуля.** Заключается в записывании в переменную объекта с набором функций, которые возвращаются из большой анонимной функции. Такой модуль имеет как публичный интерфейс (доступный снаружи), так и внутренний (доступный только внутри).

```
var module = (function () {  
    function privateCall() {  
        console.warn('Private');  
    }  
  
    return {  
        pub: function () {  
            return privateCall();  
        }  
    };  
})();
```

2 ) **Использование анонимной функции, внутренней переменной и возврата других вспомогательных функций внутри.** В итоге мы получаем в глобальной области видимости только 1 функцию, и не видим других, поддерживающих ее функций (использование абстракции).

```
var f = (function () {  
    var uniqueString = (new Date()).now().toString();  
  
    return function () {  
    };  
})();
```

Пример с учебника Кантора - счётчик:

```
function makeCounter() {  
    var currentCount = 0;  
    return function() {  
        currentCount++;  
        return currentCount;  
    };  
}  
//результат выполнения ф-ии makeCounter записывается в counter  
var counter = makeCounter();  
  
var c1 = makeCounter();  
var c2 = makeCounter();  
  
console.log(c1()); // 1  
console.log(c1()); // 2  
console.log(c1()); // 3  
console.log(c2()); // 1
```

Возвращаемая функция при вызове видит переменные из той области видимости, в которой была объявлена (в данном случае - переменную `currentCount`). Внутренняя функция при вызове будет увеличивать значение переменной `currentCount` и возвращать его. Если вызвать ф-ю `makeCounter` несколько раз, то при каждом вызове внутренняя функция будет видеть новое значение переменной `currentCount`.

Когда вызывается `c1`, вызывается функция, которая возвращается из функции `makeCounter`. Переменная

`currentCount` создается только раз, и меняется каждый раз при вызове `c1`. Таким образом функция сохраняет свое состояние.

Разберем еще один пример кода:

```
var a = 20;
function getFunction() {
  var a = 10;
  function fff () {
    console.log(a);
  }
  return fff;
}

var f = getFunction();
f(); // 10
f(); // 10
```

Присвоение `var f = getFunction()` работает следующим образом - сначала вычисляется expression справа и результат присваивается в переменную слева. Для того чтобы вычислить expression `(getFunction())` интерпретатор заходит в функцию `getFunction` и выполняет ее. Результатом вычисления `getFunction` является функция `fff`. При вызове этой функции (с помощью ссылки - `f()`) интерпретатор выполняет ее.

Функция `fff` берет свою переменную `a` из родительской области видимости. Если между вызовами `f()` значение переменной `a` будет изменено, то это изменение состояния будет сохраняться (см. пример выше со счётчиком, ф-я `makeCount`):

```
function getFunction() {
  var a = 0;
  return function() {
    a++;
    return a;
  };
}

var f = getFunction();
console.log(f()); // 1
console.log(f()); // 2
```



# Методы массивов

---

1. Метод `indexOf()` ;
2. Метод `filter()` ;
3. Метод `map()` ;
4. Метод `forEach()` ;
5. Метод `every()` ;
6. Метод `some()` ;
7. Метод `reduce()` .

# Метод indexOf( )

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/indexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf)

Метод `indexOf()` возвращает первый индекс, по которому данный элемент может быть найден в массиве или `-1`, если такого индекса нет.

## Синтаксис

```
arr.indexOf(searchElement[, fromIndex = 0])
```

## Параметры

- **searchElement** Искомый элемент в массиве.
- **fromIndex** Индекс, с которого начинать поиск. Если индекс больше или равен длине массива, возвращается `-1`, что означает, что массив даже не просматривается. Если индекс является отрицательным числом, он трактуется как смещение с конца массива. Обратите внимание: если индекс отрицателен, массив всё равно просматривается от начала к концу. Если рассчитанный индекс оказывается меньше 0, поиск ведётся по всему массиву. Значение по умолчанию равно 0, что означает, что просматривается весь массив.

## Описание

Метод `indexOf` сравнивает искомый элемент `searchElement` с элементами в массиве, используя строгое сравнение (тот же метод используется оператором `===`, тройное равно).

## Примеры

### Пример: использование indexOf

В следующем примере `indexOf` используется для поиска значений в массиве.

```
var array = [2, 5, 9];
var index = array.indexOf(2);
// index равен 0
index = array.indexOf(7);
// index равен -1
index = array.indexOf(9, 2);
// index равен 2
index = array.indexOf(2, -1);
// index равен -1
index = array.indexOf(2, -3);
// index равен 0
```

### Пример: нахождение всех вхождений элемента

В следующем примере `indexOf` используется для поиска всех индексов элемента в указанном массиве, которые с помощью `push` добавляются в другой массив.

```
var indices = [];
var array = ['a', 'b', 'a', 'c', 'a', 'd'];
var element = 'a';
var idx = array.indexOf(element);
while (idx !== -1) {
    indices.push(idx);
    idx = array.indexOf(element, idx + 1);
}
```

```
}  
  
console.log(indices);  
// [0, 2, 4]
```

# Метод filter( )

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/filter](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

Метод `filter()` создаёт новый массив со всеми элементами, прошедшими проверку, задаваемую в передаваемой функции.

## Синтаксис:

```
arr.filter(callback[, thisArg])
```

## Параметры

- **callback** Функция проверки каждого элемента. Возвращает `true` для сохранения элемента и `false` для его пропуска.
- **thisArg** Необязательный параметр. Значение, используемое в качестве `this` при выполнении функции `callback`.

Функция `callback` вызывается с тремя аргументами:

1. Значение элемента;
2. Индекс элемента;
3. Массив, по которому осуществляется проход.

## Описание

Метод `filter` вызывает переданную функцию `callback` один раз для каждого элемента, присутствующего в массиве, и конструирует новый массив со всеми значениями, для которых функция `callback` вернула `true`. Функция `callback` вызывается только для индексов массива, имеющих присвоенные значения; она не вызывается для индексов, которые были удалены или которым значения никогда не присваивались. Элементы массива, не прошедшие проверку функцией `callback`, просто пропускаются и не включаются в новый массив.

Если в метод `filter` был передан параметр `thisArg`, при вызове `callback` он будет использоваться в качестве значения `this`. В противном случае, в качестве значения `this` будет использоваться значение `undefined`. В конечном итоге, значение `this`, наблюдаемое из функции `callback`, определяется согласно обычным правилам определения `this`, видимого из функции.

Метод `filter` не изменяет массив, для которого он был вызван.

## Пример: отфильтровывание всех маленьких значений

Следующий пример использует `filter` для создания отфильтрованного массива, все элементы которого больше 10.

```
function isBigEnough(element) {  
    return element >= 10;  
}  
var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);  
// массив filtered равен [12, 130, 44]
```

# Метод `map()`

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

Метод `map()` создаёт новый массив с результатом вызова указанной функции для каждого элемента массива.

## Синтаксис

```
arr.map(callback[, thisArg])
```

## Параметры

**callback** Функция, создающая элемент в новом массиве, принимает три аргумента:

- **currentValue** Текущий элемент, создаваемый в массиве.
- **index** Индекс текущего обрабатываемого элемента в массиве.
- **array** Массив, по которому осуществляется проход.

**thisArg** Необязательный параметр. Значение, используемое в качестве `this` при вызове функции `callback`.

## Описание

Метод `map` вызывает переданную функцию `callback` один раз для каждого элемента, в порядке их появления и конструирует новый массив из результатов её вызова. Функция `callback` вызывается только для индексов массива, имеющих присвоенные значения; она не вызывается для индексов, значения по которым равны `undefined`, то есть, которые были удалены или которым значения никогда не присваивались.

Функция `callback` вызывается с тремя аргументами: значением элемента, индексом элемента и массивом, по которому осуществляется проход.

Метод `map` не изменяет массив, для которого он был вызван (хотя функция `callback` может это делать).

## Примеры

### Пример: отображение массива чисел на массив квадратных корней

Следующий код берёт массив чисел и создаёт новый массив, содержащий квадратные корни чисел из первого массива.

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// теперь roots равен [1, 2, 3], а numbers всё ещё равен [1, 4, 9]
```

### Пример: отображение массива чисел с использованием функции, содержащей аргумент

Следующий код показывает, как работает отображение, когда функция требует один аргумент. Аргумент будет автоматически присваиваться каждому элементу массива, когда `map` проходит по оригинальному массиву.

```
var numbers = [1, 4, 9];
var doubles = numbers.map(function(num) {
    return num * 2;
});
```

```
});  
// теперь doubles равен [2, 8, 18], а numbers всё ещё равен [1, 4, 9]
```

## Пример: обобщённое использование map

Этот пример показывает, как использовать `map` на объекте строки `String` для получения массива байт в кодировке ASCII, представляющего значения символов:

```
var map = Array.prototype.map;  
var a = map.call('Hello World', function(x) { return x.charCodeAt(0); });  
// теперь a равен [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
```

# Метод `forEach()`

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

Метод `forEach()` выполняет указанную функцию один раз для каждого элемента в массиве.

## Синтаксис

```
arr.forEach(callback[, thisArg])
```

## Параметры

**callback** Функция, выполняемая для каждого элемента, принимает три аргумента:

- **currentValue** Текущий обрабатываемый элемент в массиве.
- **index** Индекс текущего обрабатываемого элемента в массиве.
- **array** Массив, по которому осуществляется проход.

**thisArg** Значение, используемое в качестве `this` при вызове функции `callback`. Описание

Метод `forEach` выполняет функцию `callback` один раз для каждого элемента, находящегося в массиве в порядке возрастания. Она не будет вызвана для удалённых или пропущенных элементов массива. Однако, она будет вызвана для элементов, которые присутствуют в массиве и имеют значение `undefined`.

Метод `forEach` выполняет функцию `callback` один раз для каждого элемента массива; в отличие от методов `every` и `some`, он **всегда возвращает значение `undefined`**.

## Пример: Печать содержимого массива

Следующий код выводит каждый элемент массива в новой строке лога:

```
function logArrayElements(element, index, array) {
  console.log('a[' + index + '] = ' + element);
}

// Обратите внимание на пропуск по индексу 2, там нет элемента, поэтому он не посещается
[2, 5, , 9].forEach(logArrayElements);
// логи:
// a[0] = 2
// a[1] = 5
// a[3] = 9
```

# Метод every( )

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/every](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/every)

Метод `every()` проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции.

## Синтаксис

```
arr.every(callback[, thisArg])
```

## Параметры

**callback** Функция проверки каждого элемента, принимает три аргумента:

- **currentValue** Текущий обрабатываемый элемент в массиве.
- **index** Индекс текущего обрабатываемого элемента в массиве.
- **array** Массив, по которому осуществляется проход.

**thisArg** Необязательный параметр. Значение, используемое в качестве `this` при выполнении функции `callback`.

## Описание

Метод `every` вызывает переданную функцию `callback` один раз для каждого элемента, присутствующего в массиве до тех пор, пока не найдет такой, для которого `callback` вернет ложное значение (значение, становящееся равным `false` при приведении его к типу `Boolean`). **Если такой элемент найден, метод `every` немедленно вернёт `false`. В противном случае, если `callback` вернёт `true` для всех элементов массива, метод `every` вернёт `true`.**

Функция `callback` вызывается только для индексов массива, имеющих присвоенные значения; она не вызывается для индексов, которые были удалены или которым значения никогда не присваивались.

Метод `every` не изменяет массив, для которого он был вызван.

## Примеры

### Пример: проверка размера всех элементов массива

Следующий пример проверяет, являются ли все элементы массива числами, большими 10.

```
function isBigEnough(element, index, array) {  
    return element >= 10;  
}  
var passed = [12, 5, 8, 130, 44].every(isBigEnough);  
// passed равен false  
passed = [12, 54, 18, 130, 44].every(isBigEnough);  
// passed равен true
```

### Пример: Прерывание цикла

Следующий код использует `Array.prototype.every` для логирования содержимого массива и останавливается при превышении значением заданного порогового значения `THRESHOLD`.

```
var THRESHOLD = 12;  
var v = [5, 2, 16, 4, 3, 18, 20];  
var res;
```



```
res = v.every(function(element, index, array) {
  console.log('element:', element);
  if (element >= THRESHOLD) {
    return false;
  }

  return true;
});
console.log('res:', res);
// логи:
// element: 5
// element: 2
// element: 16
// res: false

res = v.some(function(element, index, array) {
  console.log('element:', element);
  if (element >= THRESHOLD) {
    return true;
  }

  return false;
});
console.log('res:', res);
// логи:
// element: 5
// element: 2
// element: 16
// res: true
```

# Метод `some()`

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/some](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/some)

Метод `some()` проверяет, удовлетворяет ли хоть какой-нибудь элемент массива условию, заданному в передаваемой функции.

## Синтаксис

```
arr.some(callback[, thisArg])
```

## Параметры

- **callback** Функция проверки каждого элемента.
- **thisArg** Необязательный параметр. Значение, используемое в качестве `this` при выполнении функции `callback`.

Метод `some` вызывает переданную функцию `callback` один раз для каждого элемента, присутствующего в массиве до тех пор, пока не найдет такой, для которого `callback` вернет истинное значение (значение, становящееся равным `true` при приведении его к типу `Boolean`). Если такой элемент найден, метод `some` немедленно вернёт `true`. В противном случае, если `callback` вернёт `false` для всех элементов массива, метод `some` вернёт `false`. Функция `callback` вызывается только для индексов массива, имеющих присвоенные значения; она не вызывается для индексов, которые были удалены или которым значения никогда не присваивались.

Функция `callback` вызывается с тремя аргументами: значением элемента, индексом элемента и массивом, по которому осуществляется проход.

Метод `some` не изменяет массив, для которого он был вызван.

## Пример: проверка значений элементов массива

Следующий пример проверяет, есть ли в массиве какой-нибудь элемент, больший 10.

```
function isBigEnough(element, index, array) {  
    return element >= 10;  
}  
var passed = [2, 5, 8, 1, 4].some(isBigEnough);  
// passed равен false  
passed = [12, 5, 8, 1, 4].some(isBigEnough);  
// passed равен true
```

# Метод reduce()

Источники: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Array/Reduce](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce)

[http://msdn.microsoft.com/ru-ru/library/ie/ff679975\(v=vs.94\).aspx](http://msdn.microsoft.com/ru-ru/library/ie/ff679975(v=vs.94).aspx)

Метод `reduce()` вызывает заданную функцию обратного вызова для всех элементов в массиве. Возвращаемое значение функции обратного вызова представляет собой накопленный результат и предоставляется как аргумент в следующем вызове функции обратного вызова.

## Синтаксис

```
arr.reduce(callback[, initialValue])
```

## Параметры

**callback** Функция, выполняющаяся для каждого элемента массива, принимает четыре аргумента:

- **previousValue** Значение, возвращённое предыдущим выполнением функции `callback`, либо значение `initialValue`, если оно предоставлено (смотрите пояснения ниже).
- **currentValue** Текущий обрабатываемый элемент массива.
- **index** Индекс текущего обрабатываемого элемента массива.
- **array** Массив, для которого была вызвана функция `reduce`.

**initialValue** Необязательный параметр. Объект, используемый в качестве первого аргумента при первом вызове функции `callback`.

## Описание

Метод `reduce` выполняет функцию `callback` один раз для каждого элемента, присутствующего в массиве, за исключением пустот, принимая четыре аргумента: начальное значение (или значение от предыдущего вызова `callback`), значение текущего элемента, текущий индекс и массив, по которому происходит итерация.

При первом вызове функции, параметры `previousValue` и `currentValue` могут принимать одно из двух значений. Если при вызове `reduce` передан аргумент `initialValue`, то значение `previousValue` будет равным значению `initialValue`, а значение `currentValue` будет равным первому значению в массиве. Если аргумент `initialValue` не задан, то значение `previousValue` будет равным первому значению в массиве, а значение `currentValue` будет равным второму значению в массиве.

Если массив пустой и аргумент `initialValue` не указан, будет брошено исключение `TypeError`. Если массив состоит только из одного элемента (независимо от его положения в массиве) и аргумент `initialValue` не указан, или если аргумент `initialValue` указан, но массив пустой, то будет возвращено одно это значение, без вызова функции `callback`.

Предположим, что `reduce` используется следующим образом:

```
[0, 1, 2, 3, 4].reduce(function(previousValue, currentValue, index, array) {  
    return previousValue + currentValue;  
});
```

Функция обратного вызова будет вызвана четыре раза, аргументы и возвращаемое значение при каждом вызове

будут следующими:

0:0	previousValue	currentValue	index	array	возвращаемое значение
1-й вызов	0	1	1	[0, 1, 2, 3, 4]	1
2-й вызов	1	2	2	[0, 1, 2, 3, 4]	3
3-й вызов	3	3	3	[0, 1, 2, 3, 4]	6
4-й вызов	6	4	4	[0, 1, 2, 3, 4]	10

Значение, возвращённое `reduce` будет равным последнему результату выполнения функции обратного вызова (10). Если же вы зададите начальное значение `initialValue`, результат будет выглядеть так:

```
[0, 1, 2, 3, 4].reduce(function(previousValue, currentValue, index, array) {  
  return previousValue + currentValue;  
}, 10);
```

0:0	previousValue	currentValue	index	array	возвращаемое значение
1-й вызов	10	0	0	[0, 1, 2, 3, 4]	10
2-й вызов	10	1	1	[0, 1, 2, 3, 4]	11
3-й вызов	11	2	2	[0, 1, 2, 3, 4]	13
4-й вызов	13	3	3	[0, 1, 2, 3, 4]	16
5-й вызов	16	4	4	[0, 1, 2, 3, 4]	20

Значение, возвращённое `reduce` в этот раз, конечно же, будет равным 20.

# Примеры

## Пример: суммирование всех значений в массиве

```
var total = [0, 1, 2, 3].reduce(function(a, b) {  
  return a + b;  
});  
// total == 6
```

## Пример: разворачивание массива массивов

```
var flattened = [[0, 1], [2, 3], [4, 5]].reduce(function(a, b) {  
  return a.concat(b);  
});  
// flattened равен [0, 1, 2, 3, 4, 5]
```

# Функции, методы, наследование

---

## Почитать:

<http://learn.javascript.ru/object-methods>

<http://learn.javascript.ru/this>

<http://dmitrysoshnikov.com/ecmascript/ru-chapter-7-1-oop-general-theory/>

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)

# Свои объекты: конструкторы и методы

Источник: <http://learn.javascript.ru/object-methods>

## Свои методы объектов

При объявлении в объект можно записать функцию. Она становится его методом, например:

```
var user = {
  name: 'Василий',

  // метод
  sayHi: function() {
    alert('Привет!');
  }
};

// Вызов метода
user.sayHi();
```

## Доступ к объекту через this

Для полноценной работы метод должен иметь доступ к данным объекта. В частности, вызов `user.sayHi()` может захотеть вывести имя пользователя.

Для доступа к объекту из метода используется ключевое слово `this`. Значением `this` является объект, в контексте которого вызван метод, например:

```
var user = {
  name: 'Василий',

  sayHi: function() {
    alert(this.name);
  }
};

user.sayHi();
```

Здесь при выполнении функции `user.sayHi()` в `this` будет храниться ссылка на текущий объект `user`. Использование `this` гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.

Через `this` можно обратиться к любому свойству объекта, а при желании и передать его куда-то:

```
var user = {
  name: 'Василий',
  sayHi: function() {
    showName(this); // передать текущий объект в showName
  }
};

function showName(obj) {
  alert( obj.name );
}

user.sayHi(); // Василий
```

## Функция - конструктор, «new»

Обычный синтаксис `{...}` позволяет создать один объект. Но зачастую нужно создать много однотипных объектов.

Для этого используют функции, запуская их при помощи специального оператора `new`.

**Конструктором становится любая функция, вызванная через new.**

Например:

```
function Animal(name) {  
  this.name = name;  
  this.canWalk = true;  
}  
  
var animal = new Animal("ёжик");
```

Любую функцию можно вызвать при помощи new. При этом она работает несколько иным образом, чем обычно:

1. Автоматически создается новый, пустой объект.
2. Специальное ключевое слово this получает ссылку на этот объект.
3. Функция выполняется. Как правило, она модифицирует this, добавляет методы, свойства.
4. Возвращается this.

Так что результат выполнения примера выше — это объект:

```
animal = {  
  name: "ёжик",  
  canWalk: true  
}
```

О создаваемом объекте говорят, что это «объект класса (или типа) Animal».

Если функция явно возвращает объект, то будет возвращён он, а не this.

**Названия функций, которые предназначены создавать объекты, как правило, начинают с большой буквы.**

## Создание методов в конструкторе

Использование функций для создания объекта дает большую гибкость. Можно передавать функции свойства создаваемого объекта и параметры, определяющие как его создавать.

Например, функция `User(name)` создает объект с заданным значением свойства `name` и методом `sayHi`:

```
function User(name) {  
  this.name = name;  
  
  this.sayHi = function() {  
    alert("Моё имя: " + this.name);  
  };  
}
```

## Приватные свойства

Локальные переменные функции-конструктора, с одной стороны, доступны вложенным функциям, с другой — недоступны снаружи.

В объектно-ориентированном программировании это называется «приватный (private) доступ».

Например, в коде ниже к `name` имеет доступ только метод `say`. Со стороны объекта, после его создания, больше никто не может получить `name`.

```
function User(name) {  
  this.say = function(phrase) {  
    alert(name + ' сказал: ' + phrase);  
  };  
}
```

```
    };  
  }  
  
  var user = new User('Вася');
```

Если бы name было свойством this.name — можно было бы получить его как user.name, а тут — локальная переменная. Приватный доступ.

**Замыкания никак не связаны с this.**

Доступ через замыкание осуществляется к локальной переменной, находящейся «выше» по области видимости.

А this содержит ссылку на «текущий» объект — контекст вызова, и позволяет обращаться к его свойствам. С локальными переменными это никак не связано.

Приватные свойства можно менять, например ниже метод `this.upperCaseName()` меняет приватное свойство `name` :

```
function User(name) {  
  this.upperCaseName = function() {  
    name = name.toUpperCase(); // <-- изменяет name из User  
  };  
  
  this.say = function(phrase) {  
    alert(name + ' сказал: ' + phrase); // <-- получает name из User  
  };  
}  
  
var user = new User('Вася');
```

```
user.upperCaseName();  
user.say("Да здравствует ООП!") // ВАСЯ сказал: Да здравствует ООП!
```

Вы помните, в главе Замыкания, функции изнутри мы говорили о скрытых ссылках `[[Scope]]` на внешний объект переменных? В этом примере `user.upperCaseName. [[Scope]]` и `user.say. [[Scope]]` как раз ссылаются на один и тот же объект `LexicalEnvironment`, в контексте которого они были созданы. За счёт этого обе функции имеют доступ к `name` и другим локальным переменным.

Все переменные конструктора `User` становятся приватными, так как доступны только через замыкание, из внутренних функций.

## Итого

У объекта могут быть методы:

- **Свойство, значение которого - функция, называется методом объекта и может быть вызвано как `obj.method()` . При этом объект доступен как `this` .**

Объекты могут быть созданы при помощи функций-конструкторов:

- Любая функция может быть вызвана с `new` , при этом она получает новый пустой объект в качестве `this`, в который она добавляет свойства. Если функция не решит вернуть свой объект, то её результатом будет `this` .
- Функции, которые предназначены для создания объектов, называются **конструкторами**. Их названия пишут с большой буквы, чтобы отличать от обычных.



# Контекст `this` в деталях

Источник: <http://learn.javascript.ru/this>

Значение `this` в JavaScript не зависит от объекта, в котором создана функция. Оно определяется во время вызова.

**Любая функция может иметь в себе `this`.**

Совершенно неважно, объявлена она в объекте или вне него.

**Значение `this` называется контекстом вызова и будет определено в момент вызова функции.**

Например: такая функция вполне допустима:

```
function sayHi() {  
    alert(this.firstName);  
}
```

Эта функция ещё не знает, каким будет `this`. Это выяснится при выполнении программы.

Есть несколько правил, по которым JavaScript устанавливает `this`.

## Вызов в контексте объекта

Самый распространенный случай — когда функция объявлена в объекте или присваивается ему, как в примере ниже:

```
var user = {  
    firstName: "Вася"  
};  
  
function func() {  
    alert(this.firstName);  
}  
  
user.sayHi = func;  
user.sayHi(); // this = user
```

При вызове функции как метода объекта, через точку или квадратные скобки — функция получает в `this` этот объект. В данном случае `user.sayHi()` присвоит `this = user`.

Если одну и ту же функцию запускать в контексте разных объектов, она будет получать разный `this`:

```
var user = { firstName: "Вася" };  
  
var admin = { firstName: "Админ" };  
  
function func() {  
    alert(this.firstName);  
}  
  
user.a = func; // присвоим одну функцию в свойства  
admin.b = func; // двух разных объектов user и admin  
  
user.a(); // Вася  
admin['b'](); // Админ (не важно, доступ через точку или квадратные скобки)
```

Значение `this` не зависит от того, как функция была создана, оно определяется исключительно в момент вызова.

## Вызов в режиме обычной функции

Если функция использует `this` - это подразумевает работу с объектом. Но и прямой вызов `func()` технически возможен.

Как правило, такая ситуация возникает при ошибке в разработке.

При этом `this` получает значение `window`, глобального объекта.

```
function func() {  
  alert(this); // выведет [object Window] или [object global]  
}  
  
func();
```

В современном стандарте языка это поведение изменено, вместо глобального объекта `this` будет `undefined`.

```
function func() {  
  "use strict";  
  alert(this); // выведет undefined (кроме IE<10)  
}  
  
func();
```

...Но по умолчанию браузеры ведут себя по-старому.

## Явное указание this: apply и call

Функцию можно вызвать, явно указав значение `this`.

Для этого у неё есть два метода: `call` и `apply`.

### Метод call

Синтаксис метода `call`:

```
func.call(context, arg1, arg2,...)
```

При этом вызывается функция `func`, первый аргумент `call` становится её `this`, а остальные передаются «как есть».

Вызов `func.call(context, a, b...)` — то же, что обычный вызов `func(a, b...)`, но с явно указанным контекстом `context`.

Например, функция `showName` в примере ниже вызывается через `call` в контексте объекта `user`:

```
var user = {  
  firstName: "Василий",  
  lastName: "Петров"  
};  
  
function showName() {  
  alert(this.firstName + ' ' + this.lastName);  
}  
  
showName.call(user) // "Василий Петров"
```

Можно сделать её более универсальной, добавив аргументы:

```
var user = {
  firstName: "Василий",
  surname: "Петров"
};

function getName(a, b) {
  alert( this[a] + ' ' + this[b] );
}

getName.call(user, 'firstName', 'surname') // "Василий Петров"
```

Здесь функция `getName` ВЫЗВАНА С КОНТЕКСТОМ `this = user` И ВЫВОДИТ `user['firstName']` И `user['surname']` .

## Метод `apply`

Метод `call` жёстко фиксирует количество аргументов, через запятую:

```
f.call(context, 1, 2, 3);
```

..А что, если мы захотим вызвать функцию с четырьмя аргументами? А что, если количество аргументов заранее неизвестно, и определяется во время выполнения? Для решения этой задачи существует метод `apply` .

Вызов функции при помощи `func.apply` работает аналогично `func.call` , но принимает массив аргументов вместо списка:

```
func.call(context, arg1, arg2...)

// то же что и:

func.apply(context, [arg1, arg2 ... ]);
```

Эти две строчки сработают одинаково:

```
getName.call(user, 'firstName', 'surname');

getName.apply(user, ['firstName', 'surname']);
```

Метод `apply` гораздо мощнее, чем `call` , так как можно сформировать массив аргументов динамически:

```
var args = [];
args.push('firstName');
args.push('surname');

func.apply(user, args); // вызовет func('firstName', 'surname') с this=user
```

## «Одалживание метода»

При помощи `call/apply` можно легко взять метод одного объекта, в том числе встроенного, и вызвать в контексте другого.

В JavaScript методы объекта, даже встроенные — это функции. Поэтому можно скопировать функцию, даже встроенную, из одного объекта в другой.

Это называется «одалживание метода» (на англ. `method borrowing`).

Используем эту технику для упрощения манипуляций с `arguments` . Как мы знаем, это не массив, а обычный объект.. Но как бы хотелось вызывать на нём методы массива.

```
function sayHi() {
  arguments.join = [].join; // одолжили метод (1)

  var argStr = arguments.join(':'); // (2)

  alert(argStr); // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

В строке (1) создали массив. У него есть метод  `[].join(..)` , но мы не вызываем его, а копируем, как и любое другое свойство в объект `arguments`. В строке (2) запустили его, как будто он всегда там был.

Однако, прямое копирование метода не всегда приемлемо.

**Для безопасного вызова используем `apply/call`:**

```
function sayHi() {

  var join = [].join; // ссылка на функцию теперь в переменной

  // вызовем join с this=arguments,
  // этот вызов эквивалентен arguments.join(':') из примера выше
  var argStr = join.call(arguments, ':');

  alert(argStr); // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

## Делаем из `arguments` настоящий `Array`

В JavaScript есть очень простой способ сделать из `arguments` настоящий массив. Вызовем метод массива `arr.slice(start, end)` В КОНТЕКСТЕ `arguments` :

```
function sayHi() {
  // вызов arr.slice() скопирует все элементы из this в новый массив
  var args = [].slice.call(arguments);
  alert(args.join(':')); // args -- массив аргументов
}

sayHi(1,2);
```

## «Переадресация» вызова через `apply`

При помощи `apply` мы можем сделать универсальную «переадресацию» вызова из одной функции в другую.

Например, функция `f` вызывает `g` в том же контексте, с теми же аргументами:

```
function f(a, b) {
  g.apply(this, arguments);
}
```

Плюс этого подхода — в том, что он полностью универсален:

- Его не понадобится менять, если в `f` добавятся новые аргументы.
- Если `f` является методом объекта, то текущий контекст также будет передан. Если не является — то `this` здесь вроде как не при чём, но и вреда от него не будет.

## Итого

Значение `this` устанавливается в зависимости от того, как вызвана функция:

### При вызове функции как метода

```
obj.func(...)    // this = obj  
obj["func"](...)
```

### При обычном вызове

```
func(...)        // this = window
```

### В new

```
new func()       // this = {} (новый объект)
```

### Явное указание

```
func.apply(ctx, args) // this = ctx (новый объект)  
func.call(ctx, arg1, arg2, ...)
```

# Тонкости ECMA-262-3. Часть 7.1. ООП: Общая теория

Источник: <http://dmitrysoshnikov.com/ecmascript/ru-chapter-7-1-oop-general-theory/>

**ECMAScript** – это объектно-ориентированный язык программирования с прототипной организацией.

## Особенности классовой и прототипной организаций

Рассмотрим общую теорию и ключевые моменты этих парадигм.

### Статическая классовая организация

В классовой организации присутствует понятие класса (class) и сущности (instance), принадлежащей данной классификации. Сущности класса также часто называют объектами (object) или экземплярами.

Класс представляет собой формальное абстрактное множество обобщённых характеристик сущности (знаний об объектах).

Понятие множество в этом отношении более близко к математике, однако, можно говорить о типе или классификации.

Пример (здесь и ниже – примеры будут псевдокодом):

```
C = Class {a, b, c} // класс C, с характеристиками a, b, c
```

К характеристикам сущностей относятся свойства (описание объекта) и методы (активность объекта).

Сами характеристики, также могут быть представлены объектами.

При этом, объекты хранят состояние (т.е. конкретные значения всех свойств, описанных в классе), а классы определяют жёсткую структуру (т.е. наличие тех или иных свойств) и жёсткое поведение (наличие тех или иных методов) своих экземпляров.

```
C = Class {a, b, c, method1, method2}

c1 = {a: 10, b: 20, c: 30} // объект c1 класса C
c2 = {a: 50, b: 60, c: 70} // объект c2 со своим состоянием, того же класса C
```

### Ключевые моменты классовой модели

Итак, имеем следующие ключевые моменты:

- чтобы породить объект, нужно перед этим обязательно описать его класс;
- при этом, объект будет создан по “образу и подобию” (структуре и поведению) своей классификации;
- разрешение методов осуществляется в жёсткой цепи наследования;
- классы-потомки (и соответственно, порождаемые от них объекты) содержат все свойства цепи наследования (даже, если какие-то из этих свойств не нужны конкретному унаследованному классу);
- будучи порождённым, класс не может (в виду статической организации) изменить набор характеристик (ни свойств, ни методов) своих экземпляров;
- экземпляры (вновь, в виду жёсткой статической организации) не могут обладать ни дополнительным (своим

уникальным) поведением, ни дополнительными свойствами, отличными от структуры и поведения своего класса.

Посмотрим, что предлагает альтернативная ООП организация, на базе прототипов.

## Прототипная организация

Здесь основным понятием являются динамические изменяемые (мутируемые) объекты (dynamic mutable objects).

Мутации (полная изменяемость: не только значений, но и всех характеристик) непосредственно связаны с динамикой языка.

Такие объекты могут самостоятельно хранить все свои характеристики (свойства, методы) и в классе не нуждаются.

```
object = {a: 10, b: 20, c: 30, method: fn};
object.a; // 10
object.c; // 30
object.method();
```

Более того, в виду динамики, они могут свободно изменять (добавлять, удалять, модифицировать) свои характеристики:

```
object.method5 = function () {...}; // добавили новый метод
object.d = 40; // добавили новое свойство "d"
delete object.c; // удалили свойство "c"
object.a = 100; // модифицировали свойство "a"

// в итоге: object: {a: 100, b: 20, d: 40, method: fn, method5: fn};
```

То есть, при присвоении, если определённая характеристика не существует в объекте, она создаётся и инициализируется переданным значением; если существует, — производится её модификация.

Повторное использование кода в данном случае достигается не за счёт расширения классов (обратите внимание, ни о каких классах, как о множествах жёстких характеристик, речи не идёт; здесь их вообще нет), а посредством обращения к, так называемому, прототипу.

**Прототип (Prototype)** — это объект, служащий либо прообразом для других объектов, либо вспомогательным объектом (делегатом), к характеристикам которого может обратиться оригинальный объект, в случае, если сам оригинальный объект не обладает нужной характеристикой.

Прототипом для объекта может служить абсолютно любой объект, и, опять же, в виду мутаций, объект свободно может менять свой прототип — динамически, по ходу программы.

Я напомню, мы сейчас ведём разговор об общей теории, мало касаясь реализаций; когда будем разбирать конкретные реализации (и, в частности, ECMAScript), увидим ряд своих особенностей.

Пример (псевдокод):

```
x = {a: 10, b: 20};
y = {a: 40, c: 50};
y.[[Prototype]] = x; // x - прототип y

y.a; // 40, собственная характеристика
y.c; // 50, тоже собственная
y.b; // 20 - полученная из прототипа: y.b (нет) -> y.[[Prototype]].b (да): 20

delete y.a; // удалили собственную "a"
y.a; // 10 - получена из прототипа

z = {a: 100, e: 50}
y.[[Prototype]] = z; // изменили прототип y на z
```

```
y.a; // 100 – получена из прототипа
y.e // 50, тоже – получена из прототипа

z.q = 200 // добавили новое свойство в прототип
y.q // изменения отобразились и на y
```

Данный пример показывает важную особенность и механизм, связанный с прототипом, когда прототип выступает в качестве вспомогательного объекта, к характеристикам которого, в случае отсутствия собственных подобных характеристик, обращаются другие объекты.

Этот механизм называется **делегацией (delegation)**, а связанная с ним прототипная модель, — делегирующим прототипированием.

Обращение к характеристикам в данном случае называется посылкой сообщения объекту. Т.е., когда объект не может ответить на сообщение самостоятельно, он обращается к своему прототипу (делегирует ему полномочия за ответ).

Повторное использование кода в данном случае называется **делегирующим наследованием (delegation based inheritance)** или наследованием, основанным на прототипах (prototype based inheritance).

Поскольку прототипом может быть любой объект, соответственно, и у прототипов, могут быть свои прототипы. Данная комбинация связанных между собой прототипных объектов образует, так называемую, **цепь прототипов (prototype chain)**. Она, так же, как и в статичных классах, иерархична, однако, в виду мутаций может свободно перегруппировываться, изменяя иерархию и состав.

```
x = {a: 10}

y = {b: 20}
y.[[Prototype]] = x

z = {c: 30}
z.[[Prototype]] = y

z.a // 10

// z.a найдено по цепи прототипов:
// z.a (нет) ->
// z.[[Prototype]].a (нет) ->
// z.[[Prototype]].[[Prototype]].a (да): 10
```

Касательно ECMAScript, здесь используется именно эта реализация — делегирующее прототипирование. Однако, как мы увидим, на уровне стандарта и реализаций есть и свои особенности.

## Ключевые особенности прототипной модели

Итак, выделим ключевые моменты данной организации:

- основным понятием является объект;
- объекты полностью динамичны и изменяемы (и в теории, могут полностью мутировать из одного вида в другой);
- у объектов нет жёстких классов, задающих их структуру и поведение; объекты не нуждаются в классах;
- однако, не имея классов, объекты могут иметь прототипы, к которым можно делегировать, если сами объекты не в состоянии ответить на посланное им сообщение;
- прототип объекта может быть изменён в любое время программы;
- в делегирующей прототипной модели, изменение характеристик прототипа отображается на всех объектах, связанных с этим прототипом;



- каскадная же прототипная модель, служит прообразом, с которого порождаемые объекты снимают точную копию и дальше становятся полностью самостоятельными; изменение прототипа в данной модели уже не влияет на клонируемые от него объекты;
- если сообщение обработать не удаётся, можно сигнализировать об этом вызывающей стороне, которая может предпринять дополнительные меры (например, изменить диспетчеризацию);
- идентификация объектов может производиться не по их иерархии и принадлежности к конкретному типу, а по текущему набору характеристик.

## Полиморфизм

Объекты ECMAScript – полиморфны во многих отношениях.

К примеру, одна функция может быть применена к разным объектам, как, если бы, она являлась родной характеристикой объекта (в виду определения `this` на этапе вызова):

```
function test() {  
  alert([this.a, this.b]);  
}  
  
test.call({a: 10, b: 20}); // 10, 20  
test.call({a: 100, b: 200}); // 100, 200  
  
var a = 1;  
var b = 2;  
  
test(); // 1, 2
```

# Наследование и цепочка прототипов

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)

Модель наследования в JavaScript может сбить с толку опытных разработчиков на высокоуровневых объектно-ориентированных языках (таких, например, как Java или C++), так как она динамическая и не включает в себя реализацию понятия class (хотя ключевое слово class и является зарезервированным, т.е., не может быть использовано в качестве имени переменной).

**В плане наследования JavaScript работает лишь с одной сущностью: объектами.** Каждый объект имеет внутреннюю ссылку на другой объект, называемый его **прототипом**. У объекта-прототипа также есть свой собственный прототип и так далее до тех пор, пока цепочка не завершится объектом, у которого свойство prototype равно null. null, по определению, не имеет прототипа и служит в качестве завершающего звена в цепочке прототипов.

## Наследование с цепочкой прототипов

### Наследование свойств

Объекты в JavaScript - это как бы динамические "контейнеры", наполненные свойствами (называемыми собственными свойствами) и у каждого объекта есть при этом ссылка на свой объект-прототип. При попытке получить доступ к какому-либо свойству объекта происходит следующее:

```
// Допустим, у нас есть объект 'o' с цепочкой прототипов выглядящей как:
// {a:1, b:2} ---> {b:3, c:4} ---> null
// где 'a' и 'b' - собственные свойства объекта 'o'.

// В этом примере someObject.[[Prototype]] означает прототип someObject.
// Это упрощённая нотация (описанная в стандарте ECMAScript). Она не может быть использована в скриптах.

console.log(o.a); // 1
// Есть ли у объекта 'o' собственное свойство 'a'? Да, и его значение равно 1

console.log(o.b); // 2
// Есть ли у объекта 'o' собственное свойство 'b'? Да, и его значение равно 2
// У прототипа тоже есть свойство 'b', но обращения к нему в данном случае не происходит. Это и называется "property shadowing"

console.log(o.c); // 4
// Есть ли у объекта 'o' собственное свойство 'c'? Нет, тогда поищем его в прототипе.
// Есть ли у объекта o.[[Prototype]] собственное свойство 'c'? Да, оно равно 4

console.log(o.d); // undefined
// Есть ли у объекта 'o' собственное свойство 'd'? Нет, тогда поищем его в прототипе.
// Есть ли у объекта o.[[Prototype]] собственное свойство 'd'? Нет, продолжим поиск по цепочке прототипов.
// o.[[Prototype]].[[Prototype]] равно null, прекращаем поиск, свойство не найдено, возвращаем undefined
```

При добавлении к объекту нового свойства создаётся новое собственное свойство (own property). Единственным исключением из этого правила являются наследуемые свойства, имеющие getter или setter.

### Наследование "методов"

JavaScript не имеет "методов" в смысле, принятом в классической модели ООП. В JavaScript любая функция может быть добавлена к объекту в виде его свойства. Унаследованная функция ведёт себя точно так же, как любое другое свойство объекта, в том числе и в плане "затенения свойств" (property shadowing), как показано в примере выше (в данном конкретном случае это форма переопределения метода - method overriding).

В области видимости унаследованной функции ссылка this указывает на наследуемый объект, а не на прототип, в котором данная функция является собственным свойством.

```

var o = {
  a: 2,
  m: function(b){
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// в этом случае при вызове 'o.m' this указывает на 'o'

var p = Object.create(o);
// 'p' - наследник 'o'

p.a = 12; // создаст собственное свойство 'a' объекта 'p'
console.log(p.m()); // 13
// при вызове 'p.m' this указывает на 'p'.
// т.е. когда 'p' наследует функцию 'm' объекта 'o', this.a означает 'p.a', собственное свойство 'a' объекта 'p'

```

## Различные способы создания объектов и получаемые в итоге цепочки прототипов

### Создание объектов с помощью литералов

```

var o = {a: 1};

// Созданный объект 'o' имеет Object.prototype в качестве своего [[Prototype]]
// 'o' имеет собственное свойство 'hasOwnProperty'
// hasOwnProperty - это собственное свойство Object.prototype. Таким образом 'o' наследует hasOwnProperty от Object.prototype
// Object.prototype в качестве прототипа имеет null.
// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];

// Массивы наследуются от Array.prototype (у которого есть такие методы, как indexOf, forEach и т.п.).
// Цепочка прототипов при этом выглядит так:
// a ---> Array.prototype ---> Object.prototype ---> null

function f(){
  return 2;
}

// Функции наследуются от Function.prototype (у которого есть такие методы, как call, bind и т.п.):
// f ---> Function.prototype ---> Object.prototype ---> null

```

### Создание объектов с помощью конструктора

Очень упрощённо говоря, "конструктор" в JavaScript - это "обычная" функция, вызываемая с оператором `new`.

```

function Graph() {
  this.vertexes = [];
  this.edges = [];
}

Graph.prototype = {
  addVertex: function(v){
    this.vertexes.push(v);
  }
};

var g = new Graph();
// объект 'g' имеет собственные свойства 'vertexes' и 'edges'.
// g.[[Prototype]] принимает значение Graph.prototype при выполнении new Graph().

```

### Object.create

В ECMAScript 5 представлен новый метод создания объектов: `Object.create`. Прототип создаваемого объекта указывается в первом аргументе этого метода:

```
var a = {a: 1};  
// a ---> Object.prototype ---> null  
  
var b = Object.create(a);  
// b ---> a ---> Object.prototype ---> null  
console.log(b.a); // 1 (унаследовано)  
  
var c = Object.create(b);  
// c ---> b ---> a ---> Object.prototype ---> null  
  
var d = Object.create(null);  
// d ---> null  
console.log(d.hasOwnProperty); // undefined, т.к. 'd' не наследуется от Object.prototype
```

## Производительность

Поиск свойств, располагающихся относительно высоко по цепочке прототипов, может негативно сказаться на производительности, особенно в критических в этом плане местах кода. Если же мы попытаемся найти несуществующее свойство, то поиск будет осуществлён вообще по всей цепочке, со всеми вытекающими последствиями.

Вдобавок, при циклическом переборе свойств объекта будет обработано каждое свойство, присутствующее в цепочке прототипов.

Если вам необходимо проверить, определено ли свойство у самого объекта, а не где-то в его цепочке прототипов, вы можете использовать метод `hasOwnProperty`, который все объекты наследуют от `Object.prototype`.

**hasOwnProperty** — единственная функция в JavaScript, которая помогает получать свойства объекта без обращения к цепочке его прототипов.

**Примечание:** Для проверки существования свойства недостаточно проверять, эквивалентно ли оно `undefined`. Свойство может вполне себе существовать, но при этом ему может быть присвоено значение `undefined`.

## Плохое применение: расширение базовых прототипов

Часто встречается неверное применение модели прототипного наследования — расширение прототипа `Object.prototype` или прототипов нативных (т.е., базовых) объектов JavaScript.

Подобная практика нарушает принцип инкапсуляции и снискала себе соответствующее название — **monkey patching**. К сожалению, в основу многих широко распространенных фреймворков, например "Prototype.js", положен принцип изменения базовых прототипов. На самом деле до сих пор не известно разумных причин примешивать в нативные прототипы нестандартную функциональность.

Единственным оправданием для расширения базовых прототипов может быть только эмуляция возможностей более новых движков JavaScript для более старых, например, эмуляция метода `Array.forEach`, который появился в версии языка 1.6.

# Методы функций

---

1. Метод `call()` ;
2. Метод `apply()` .

# Метод call( )

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Function/call](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call)

Метод `call()` вызывает функцию с указанным значением `this` и индивидуально предоставленными аргументами.

**Примечание:** хотя синтаксис этой функции практически полностью идентичен функции `apply()`, фундаментальное различие между ними заключается в том, что функция `call()` принимает список аргументов, в то время, как функция `apply()` принимает простой массив с аргументами.

## Синтаксис

```
fun.call(thisArg[, arg1[, arg2[, ...]]])
```

## Параметры

- **thisArg** Значение `this`, предоставляемое для вызова функции `fun`. Обратите внимание, что `this` может не быть реальным значением, видимым этим методом: если метод является функцией в нестрогом режиме, значения `null` и `undefined` будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.
- **arg1, arg2, ...** Аргументы для объекта.

## Описание

Вы можете присваивать различные объекты `this` при вызове существующей функции. `this` ссылается на текущий объект, вызвавший объект. С помощью `call` вы можете написать метод один раз, а затем наследовать его в других объектах, без необходимости переписывать метод для каждого нового объекта.

## Примеры

### Пример: использование call для связи конструкторов объекта в цепочку

Вы можете использовать метод `call` для объединения в цепочку конструкторов объекта, как в Java. В следующем примере для объекта продукта `Product` объявлен конструктор с двумя параметрами, названием `name` и ценой `price`. Продукт инициализирует свойства `name` и `price`, а специализированные функции определяют ещё категорию `category`.

```
function Product(name, price) {
  this.name = name;
  this.price = price;

  if (price < 0) {
    throw RangeError('Нельзя создать продукт ' +
      this.name + ' с отрицательной ценой');
  }

  return this;
}

function Food(name, price) {
  Product.call(this, name, price);
  this.category = 'еда';
}

Food.prototype = Object.create(Product.prototype);

function Toy(name, price) {
  Product.call(this, name, price);
  this.category = 'игрушка';
}
```

```
}

Toy.prototype = Object.create(Product.prototype);

var cheese = new Food('фета', 5);
var fun = new Toy('робот', 40);
```

## Пример: использование call для вызова анонимной функции

В этом чисто искусственном примере, мы создаём анонимную функцию и используем `call` для вызова её на каждом элементе массива. Главная задача анонимной функции здесь — добавить функцию печати в каждый объект, способную напечатать правильный индекс объекта в массиве. Передача объекта в качестве значения `this` не является острой необходимостью, но мы делаем это в целях объяснения.

```
var animals = [
  { species: 'лев', name: 'Король' },
  { species: 'кит', name: 'Фэйл' }
];

for (var i = 0; i < animals.length; i++) {
  (function(i) {
    this.print = function() {
      console.log('#' + i + ' ' + this.species
        + ': ' + this.name);
    }
    this.print();
  }).call(animals[i], i);
}
```

# Метод `apply()`

Источник: [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/Function/apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply)

Метод `apply()` вызывает функцию с указанным значением `this` и аргументами, предоставленными в виде массива (либо массивоподобного объекта).

**Примечание:** хотя синтаксис этой функции практически полностью идентичен функции `call()`, фундаментальное различие между ними заключается в том, что функция `call()` принимает список аргументов, в то время, как функция `apply()` принимает простой массив с аргументами.

## Синтаксис

```
fun.apply(thisArg[, argsArray])
```

### Параметры

- **thisArg** Значение `this`, предоставляемое для вызова функции `fun`. Обратите внимание, что `this` может не быть реальным значением, видимым этим методом: если метод является функцией в нестрогом режиме, значения `null` и `undefined` будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.
- **argsArray** Массивоподобный объект, определяющий аргументы, с которыми функция `fun` должна быть вызвана, либо `null` или `undefined`, если в функцию не надо передавать аргументы. Начиная с ECMAScript 5 эти аргументы могут быть обобщёнными массивоподобными объектами, а не только массивом. Смотрите ниже информацию по совместимости с браузерами.

## Описание

Вы можете присваивать различные объекты `this` при вызове существующей функции. `this` ссылается на текущий объект, вызвавший объект. С помощью `apply` вы можете написать метод один раз, а затем наследовать его в других объектах, без необходимости переписывать метод для каждого нового объекта.

Метод `apply` очень похож на метод `call()`, за исключением поддерживаемого типа аргументов. Вы можете использовать массив аргументов вместо набора именованных параметров. Вместе с `apply` вы можете использовать литерал массива, например, `fun.apply(this, ['есть', 'бананы'])`, либо объект `Array`, например, `fun.apply(this, new Array('есть', 'бананы'))`.

Также вы можете использовать в качестве параметра `argsArray` псевдомассив `arguments`. `arguments` является локальной переменной функции. Он может использоваться для всех неопределённых аргументов вызываемого объекта. Таким образом, вы не обязаны знать, сколько и какие аргументы требует вызываемый объект при использовании метода `apply`. Вы можете использовать псевдомассив `arguments` для передачи всех аргументов в вызываемый объект. Вызываемый объект самостоятельно разберётся с обработкой аргументов.



# Конспект. Функции, debugger, this

Источник: <http://forum.jscourse.com/t/07-konspekt-funkczii-debugger-this/629>

Автор конспекта: @Nikolay

## this

Все объекты, которые создаются в JS, являются представлением данных из предметной области. Объект объединяет набор свойств из той области, в которой задача решается.

Работая с объектами/структурами, мы выполняем действия, которые являются проекцией на предметную область (реальную или абстрактную). Объекты - это отражение реальности.

Обычно, в JS действия, которые связаны с объектом, привязываются к самому объекту. И действия и данные хранятся в одном месте (в других ЯП это называется class). То есть, действия, которые можно связать с объектом, вкладываются в сам объект. Использование данной методики позволяет быстрее понять какие методы к каким типам данным предопределены.

То есть, вместо того чтобы не создавать функцию отдельно, (как в этом примере):

```
var car2 = {
  durability: 0.38,
  gas: 50,
  speeds: {
    slow: {
      durability: 0.002,
      gas: 10
    },
    average: {
      durability: 0.002,
      gas: 12
    },
    fast: {
      durability: 0.006,
      gas: 11
    }
  }
};

car.drive.call(car2, 100, 'slow')

function drive(car, distance, speed) {
  var speedCharacteristics = car.speeds[speed];
  var drivingDistance = Math.min(distance, car.gas / speedCharacteristics.gas * 100);

  var durabilitySpent = drivingDistance * speedCharacteristics.durability;
  var durabilityLeft = car.durability - durabilitySpent;

  var realDistance;
  if (durabilityLeft < 0) {
    realDistance = car.durability / speedCharacteristics.durability;
  } else {
    realDistance = drivingDistance;
  }
  var realDurabilitySpent = realDistance * speedCharacteristics.durability;
  var realGasSpent = realDistance * speedCharacteristics.gas / 100;
  car.durability -= realDurabilitySpent;
  car.gas -= realGasSpent;
  return car;
}

drive(car2, 100, 'average');
```

и не обращаться к ней способом `drive(car, 100, 'average')`, передавая ей аргументом объект `car`, можно поместить эту функцию в сам объект `this`

```

var car = {
  durability: 0.78,
  gas: 100,
  speeds: { ... }, // часть кода упущена
  drive: function(distance, speed) {
    var speedCharacteristics = this.speeds[speed];
    var drivingDistance = Math.min(distance, this.gas / speedCharacteristics.gas * 100);
    var durabilitySpent = drivingDistance * speedCharacteristics.durability;
    var durabilityLeft = this.durability - durabilitySpent;
    // часть кода упущена
    return car;
  }
};
// вместо car. теперь используется this.
car.drive();

```

Если записать функцию в объект и вызвать ее в форме `car.drive()`, то при ее вызове `this` будет ссылаться на тот объект, который находится до точки, т.е. `car`. Такой функции нету разницы с какой машиной работать: она берет значение с абстрактных мест, при этом не зная какие это значения конкретно и какая машина.

`this` позволяет абстрагироваться от объекта, с которым работает функция.

## Call & Apply

Рассмотрим следующий способ вызвать функцию и передать в нее явный `this` с использованием `call/apply`:

```

function logThis() {
  console.log('this', this);
}
logThis.call({foo: 'bar'});

//Output:
this {foo: "bar"}

```

В функцию передается тот `this`, который должен быть в ней.

Также `call/apply` позволяют передавать в функцию и другие аргументы:

```

function logThis() {
  console.log('this', this);
  console.log('arguments', arguments);
}

logThis.call({foo: 'bar'}, 'uno', 'tuo', 'tre');

//Output:
this {foo: "bar"} // this - это первый аргумент
arguments ["uno", "tuo", "tre"] // и arguments - все последующие аргументы

```

Разница между `call` и `apply` в том, что они по-разному принимают аргументы. `apply` принимает масивоподобные объекты или массивы в качестве второго объекта, позволяет передать аргументы пачкой. Его удобно использовать, когда аргументы какой-то функции сначала необходимо собрать в массив и потом явно их передать из массива.

Вернемся к примеру с машинами и вызовем функцию `drive` с одного объекта (`car`), для другого (`car2`).

```

car.drive.call(car2, 100, 'slow')

```

```

// ссылка на функцию с объекта car вызывает эту функцию в контексте car2, при этом задаются и другие аргументы - 100, '

```

При вызове этой функции `this` будет ссылаться на `car2` и функция, которая находится в одном объекте (`car`),

будет выполняться для другого объекта (`car2`) . Таким образом можно использовать одну функцию для работы с множеством схожих по структуре объектов. Это называется наследованием.

## Вызов функции как метода объекта.

Следует обратить внимание на то, что более важно то, как функция была вызвана, а не объявлена. Способ вызова функции определяет, чему будет равно `this` .

```
function logThis() {  
    console.log('this', this);  
}  
  
var obj = {  
    test: true,  
    action: logThis  
};  
  
obj.action();  
  
//this -> объект obj
```

## Можно сделать следующие выводы:

- В способе вызова функции `obj.action()`; `this`-ом будет объект до точки - `obj` .
- Если используется `call/apply` , то `this` внутри функции будет явный объект, который стоит на первом месте.
- Если функция объявлена и вызвана без переменной, то `this` ссылается на глобальный объект либо на `undefined`.

# Задачи

---

## Задача 1

Напишите функцию `f`, которая будет обёрткой вокруг другой функции `g`. Функция `f` обрабатывает первый аргумент сама, а остальные аргументы передаёт в функцию `g`, сколько бы их ни было.

Например:

```
function f() { /* ваш код */ }

function g(a, b, c) {
  alert( a + b + (c || 0) );
}

f("тест", 1, 2); // f выведет "тест", дальше g посчитает сумму "3"
f("тест2", 1, 2, 3); // f выведет "тест2", дальше g посчитает сумму "6"
```

Код функции `f` не должен зависеть от количества аргументов.

## Решение

```
function f(a) {
  alert(a);
  var args = [].slice.call(arguments, 1);
  g.apply(this, args);
}

function g(a, b, c) {
  alert( a + b + (c || 0) );
}

f("тест", 1, 2);
f("тест2", 1, 2, 3);
```

# Создайте объект calculator с тремя методами

- метод `readValues()` запрашивает `prompt` два значения и сохраняет их как свойства объекта;
- метод `sum()` возвращает сумму двух значений;
- метод `mul()` возвращает произведение двух значений.

```
var calculator = {  
  ... ваш код ...  
}  
  
calculator.readValues();  
alert(calculator.sum());  
alert(calculator.mul());
```

## Решение:

```
var calculator = {  
  sum: function() {  
    return this.a + this.b;  
  },  
  
  mul: function() {  
    return this.a * this.b;  
  },  
  
  readValues: function() {  
    this.a = +prompt('a?', 0);  
    this.b = +prompt('b?', 0);  
  }  
}  
  
calculator.readValues();  
alert(calculator.sum());  
alert(calculator.mul());
```

# Напишите функцию-конструктор Summator, которая создает объект с двумя методами

---

1. Метод `sum(a, b)` возвращает сумму двух значений;
2. Метод `run()` запрашивает два значения при помощи `prompt` и выводит их сумму, используя метод `sum`.

В итоге вызов `new Summator().run()` должен спрашивать два значения и выводить их сумму.

## Решение:

```
function Summator() {  
  
    this.sum = function(a, b) {  
        return a + b;  
    };  
  
    this.run = function() {  
        var a = +prompt('a?', 0); // преобразовать в число при вводе данных  
        var b = +prompt('b?', 0);  
        alert("sum=" + this.sum(a, b));  
    };  
}  
  
new Summator().run();
```

# Напишите функцию-конструктор Adder(startingValue).

Объекты, которые она создает, должны хранить текущую сумму и прибавлять к ней то, что вводит посетитель.

Более формально, объект должен:

Хранить текущее значение в своём свойстве `value`. Начальное значение свойства `value` ставится конструктором равным `startingValue`. Метод `addInput()` вызывает `prompt`, принимает число и прибавляет его к свойству `value`. Метод `showValue()` выводит текущее значение `value`. Таким образом, свойство `value` является текущей суммой всего, что ввел посетитель при вызовах метода `addInput()`, с учетом начального значения `startingValue`.

По ссылке ниже вы можете посмотреть работу кода:

```
var adder = new Adder(1); // начальное значение 1
adder.addInput(); // прибавит ввод prompt к текущему значению
adder.addInput(); // прибавит ввод prompt к текущему значению
adder.showValue(); // выведет текущее значение
```

## Решение

```
function Adder(startingValue) {
  this.value = startingValue;

  this.addInput = function() {
    this.value += +prompt('Сколько добавлять будем?', 0);
  };

  this.showValue = function() {
    alert(this.value);
  };
}

var adder = new Adder(1);
adder.addInput();
adder.addInput();
adder.showValue();
```

# Классы, наследование

---

Почитать, выполнить прилагающиеся задачи:

<http://learn.javascript.ru/prototype>

<http://learn.javascript.ru/classes>

<http://dmitrypodgorniy.com/blog/2012/07/22/1/>



# Прототип: наследование и методы

Источник: <http://learn.javascript.ru/prototype>

**Смысл прототипного наследования** в том, что один объект можно сделать прототипом другого. При этом если свойство не найдено в объекте — оно берётся из прототипа.

## Наследование через ссылку `__proto__`

Наследование в JavaScript реализуется при помощи специального свойства `__proto__`.

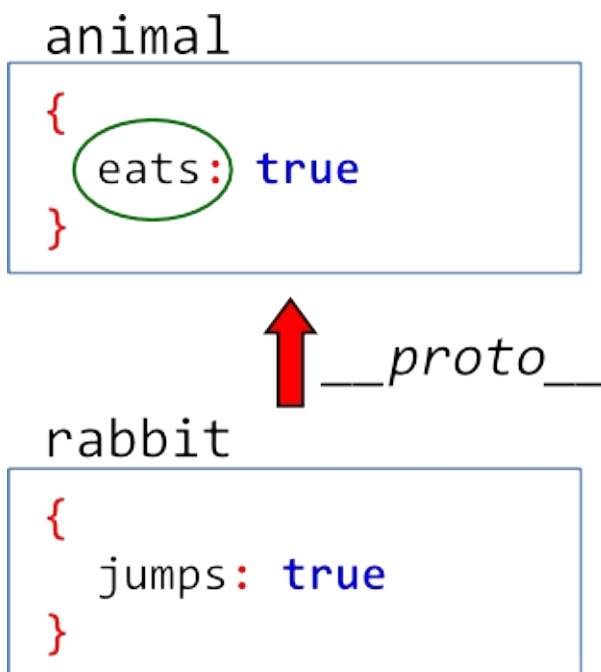
Если один объект, например, `rabbit`, имеет специальную ссылку `__proto__` на другой объект `animal`, то все свойства, которые ищутся в `rabbit`, будут затем искаться в `animal`.

```
var animal = { eats: true };
var rabbit = { jumps: true };

rabbit.__proto__ = animal; // унаследовать

alert(rabbit.eats); // true
alert(rabbit.jumps); // true
```

Когда запрашивается свойство `rabbit`, интерпретатор ищет его сначала в самом объекте `rabbit`, а если не находит — в объекте `rabbit.__proto__`, то есть, в данном случае, в `animal`.



Объект, на который указывает `__proto__`, называется его «прототипом».

Нет никаких ограничений на объект, который записывается в прототип. Например:

```
var rabbit = { };

rabbit.__proto__ = window;

rabbit.open('http://google.com'); // вызовет метод open из window
```

Если вы будете читать спецификацию EcmaScript — свойство `__proto__` обозначено в ней как `[[Prototype]]`. Не

путать со свойством `prototype` . Прототип используется только если свойство не найдено.

```
var animal = { eats: true };
var fedUpRabbit = { eats: false };

fedUpRabbit.__proto__ = animal;

alert(fedUpRabbit.eats); // false, свойство взято из fedUpRabbit
```

Другими словами, прототип — это «резервное хранилище свойств и методов» объекта, автоматически используемое при поиске.

## Цепочка прототипов

У объекта, который является `__proto__` , может быть свой `__proto__` , у того — свой, и так далее.

Например, цепочка наследования из трех объектов `donkey -> winnie -> owl` :

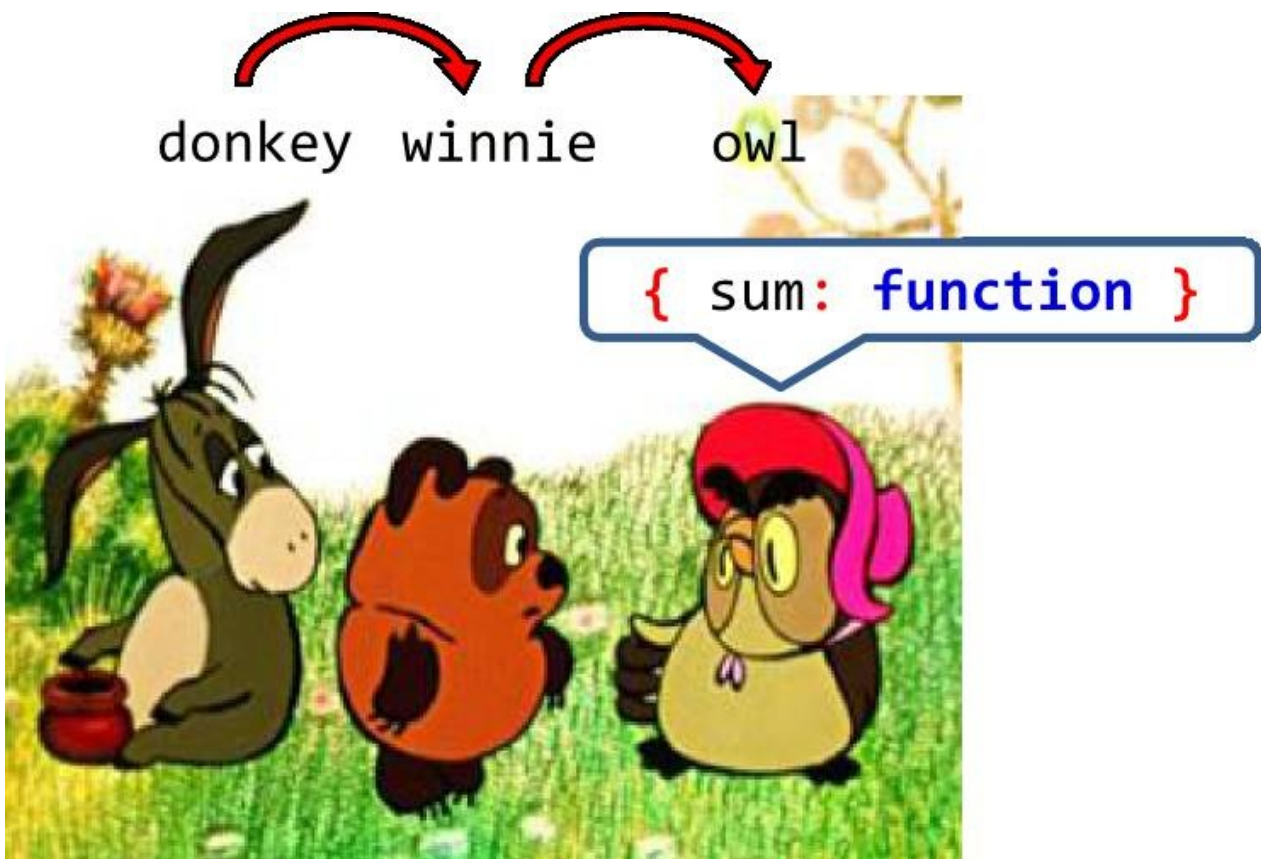
```
var owl = {
  sum: function(a, b) {
    return a + b;
  }
}

var winnie = { /* ... */ }
winnie.__proto__ = owl;

var donkey = { /* ... */ }
donkey.__proto__ = winnie;

alert( donkey.sum(2,2) ); // "4" ответит owl
```

Картина происходящего:



Создание объекта с данным прототипом

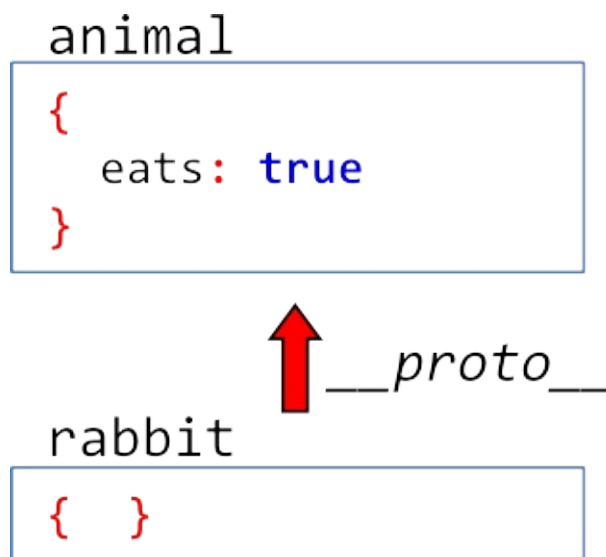
Свойство `__proto__` доступно в явном виде не во всех браузерах. Поэтому используются другие способы для создания объектов с данным прототипом.

## Object.create(proto) (кроме IE8-)

Вызов `Object.create(proto)` создаёт пустой объект с данным прототипом `proto`. Например:

```
var animal = { eats: true };  
var rabbit = Object.create(animal);  
alert(rabbit.eats); // true
```

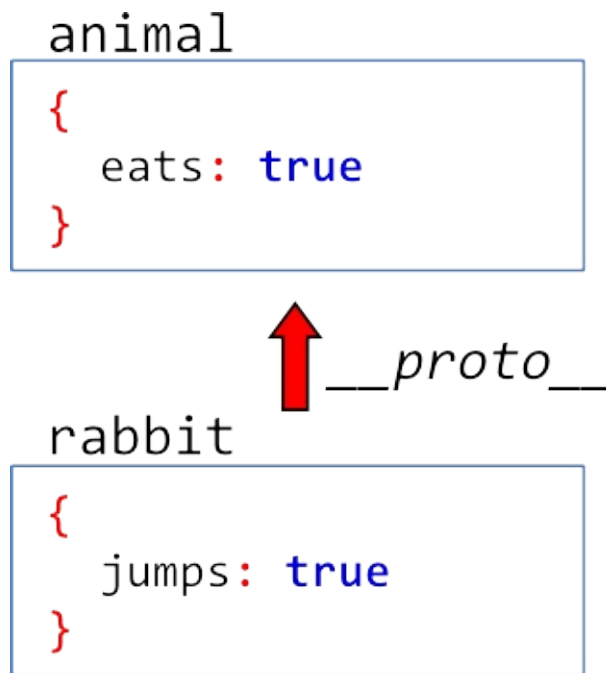
Этот код создал пустой объект `rabbit` с прототипом `animal`:



Мы можем добавить свойства в новый объект `rabbit`:

```
var animal = { eats: true };  
var rabbit = Object.create(animal);  
rabbit.jumps = true;
```

Станет:



Этот метод позволяет только создать новый пустой объект. Он не может изменить прототип существующего объекта.

## Свойство `F.prototype`

Созданием объектов часто занимается функция-конструктор. Чтобы таким объектам автоматически ставить прототип, существует свойство `prototype`.

Свойство `prototype` можно указать на любом объекте, но особый смысл оно имеет, если назначено функции.

При создании объекта через `new`, в его прототип `__proto__` записывается ссылка из `prototype` функции-конструктора.

Например:

```
var animal = { eats: true };

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

var rabbit = new Rabbit('John');

alert( rabbit.eats ); // true, т.к. rabbit.__proto__ == animal
```

Установка `Rabbit.prototype = animal` говорит интерпретатору следующее: «запиши `__proto__ = animal` при создании объекта через `new Rabbit`».

Значением `prototype` может быть только объект.

## Эмуляция `Object.create` для IE8-

Вызов `object.create(proto)`, который создаёт пустой объект с данным прототипом, можно эмулировать, так что он будет работать во всех браузерах, включая IE6+.

Кросс-браузерный аналог — `inherit` состоит буквально из нескольких строк:

```
function inherit(proto) {
  function F() {}
  F.prototype = proto;
  var object = new F;
  return object;
}
```

1. Создана новая функция `F`. Она ничего не делает с `this`, так что вызов `new F` вернёт пустой объект.
2. Свойство `F.prototype` устанавливается в будущий прототип `proto`;
3. Результатом вызова `new F` будет пустой объект с `__proto__` равным значению `F.prototype`.
4. Готово! Мы получили пустой объект с заданным прототипом.

Результат вызова `inherit(animal)` идентичен `Object.create(animal)`. Это будет новый пустой объект с прототипом `animal`.

Например:

```
var animal = { eats: true };

var rabbit = inherit(animal);

alert(rabbit.eats); // true
```

## Метод `Object.getPrototypeOf(obj)`

Вызов `Object.getPrototypeOf(obj)` возвращает прототип `obj`.

Не поддерживается в IE8-.

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);

alert( Object.getPrototypeOf(rabbit) === animal ); // true
```

Этот метод даёт возможность получить `__proto__`, но не изменить его.

## Метод `obj.hasOwnProperty`

Метод `obj.hasOwnProperty(prop)` есть у всех объектов. Он позволяет проверить, принадлежит ли свойство самому объекту, без учета его прототипа.

Например:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

alert( rabbit.hasOwnProperty('jumps') ); // true: jumps принадлежит rabbit
alert( rabbit.hasOwnProperty('eats') ); // false: eats не принадлежит
```

## Перебор свойств объекта без прототипа

Цикл `for...in` перебирает все свойства в объекте и его прототипе.

Например:

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

for (var key in rabbit) {
  alert (key + " = " + rabbit[key]); // выводит и "eats" и "jumps"
}
```

Иногда хочется посмотреть, что находится именно в самом объекте, а не в прототипе.

Это можно сделать, если профильтровать `key` через `hasOwnProperty` :

```
var animal = {
  eats: true
};

rabbit = Object.create(animal);
rabbit.jumps = true;

for (var key in rabbit) {
  if ( !rabbit.hasOwnProperty(key) ) continue; // пропустить "не свои" свойства
  alert (key + " = " + rabbit[key]); // выводит только "jumps"
}
```

## Итого

Мы рассмотрели основы наследования в JavaScript. Упорядочим эту информацию.

- Наследование реализуется через специальное свойство `__proto__` (в спецификации обозначено `[[Prototype]]` ). Оно работает так: при попытке получить свойство из объекта, если его там нет, оно ищется в `__proto__` объекта.
- Замыкания и `this` с прототипами никак не связаны, они работают по своим правилам.

Установка прототипа `__proto__` :

- Firefox, Chrome и Safari дают полный доступ к `obj.__proto__` . Эта нестандартная возможность бывает полезна в целях отладки.
- Функция-конструктор при создании объекта устанавливает его `__proto__` равным своему `prototype` .
- Вызов `Object.create(proto)` создаёт пустой объект с прототипом `proto` .

В IE8- этого метода нет, но его можно эмулировать при помощи следующей функции `inherit` :

```
function inherit(proto) {
  function F() {}
  F.prototype = proto;
  return new F;
}
```

Можно даже присвоить `Object.create = inherit` , чтобы иметь унифицированный вызов, но при этом стоит иметь в виду, что современные браузеры поддерживают также дополнительный второй аргумент `Object.create` , позволяющий задать свойства объекта, а `inherit` — нет.

Кроме того, есть следующие методы для работы с прототипом:

- Метод `Object.getPrototypeOf(obj)` — получить прототип объекта `obj`, кроме IE8-
- Метод `obj.hasOwnProperty(prop)` — возвращает `true`, если `prop` является свойством объекта `obj`, без учёта прототипа.

Проверка `obj.hasOwnProperty` используется, в частности, в `for...in` для перебора свойств именно самого объекта, без прототипа.

# "Классы" в JavaScript

Источник: <http://learn.javascript.ru/classes>

В JavaScript есть встроенные объекты: `Date`, `Array`, `Object` и другие. Они используют прототипы и демонстрируют концепцию «псевдоклассов», которую мы вполне можем применить и для себя.

## Откуда методы у `{ }` ?

Начнём мы с того, что создадим пустой объект и выведем его.

```
var obj = { };  
alert( obj ); // "[object Object]" ?
```

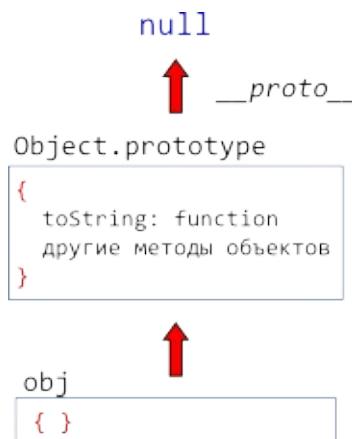
В объекте, очевидно, ничего нет... Но кто же тогда генерирует строковое представление для `alert(obj)` ?

## Object.prototype

Встроенный прототип `Object.prototype` ставится всем объектам `Object`, и поэтому его методы доступны с момента создания.

В деталях, работает это так:

1. Запись `obj = { }` является краткой формой `obj = new Object`, где `Object` — встроенная функция-конструктор для объектов.
2. При выполнении `new Object`, создаваемому объекту ставится `__proto__` по `prototype` конструктора, в данном случае это будет `Object.prototype` — встроенный объект, хранящий свойства и методы, общие для объектов, в частности, есть `Object.prototype.toString`.
3. В дальнейшем при обращении к `obj.toString()` — функция будет взята из прототипа.



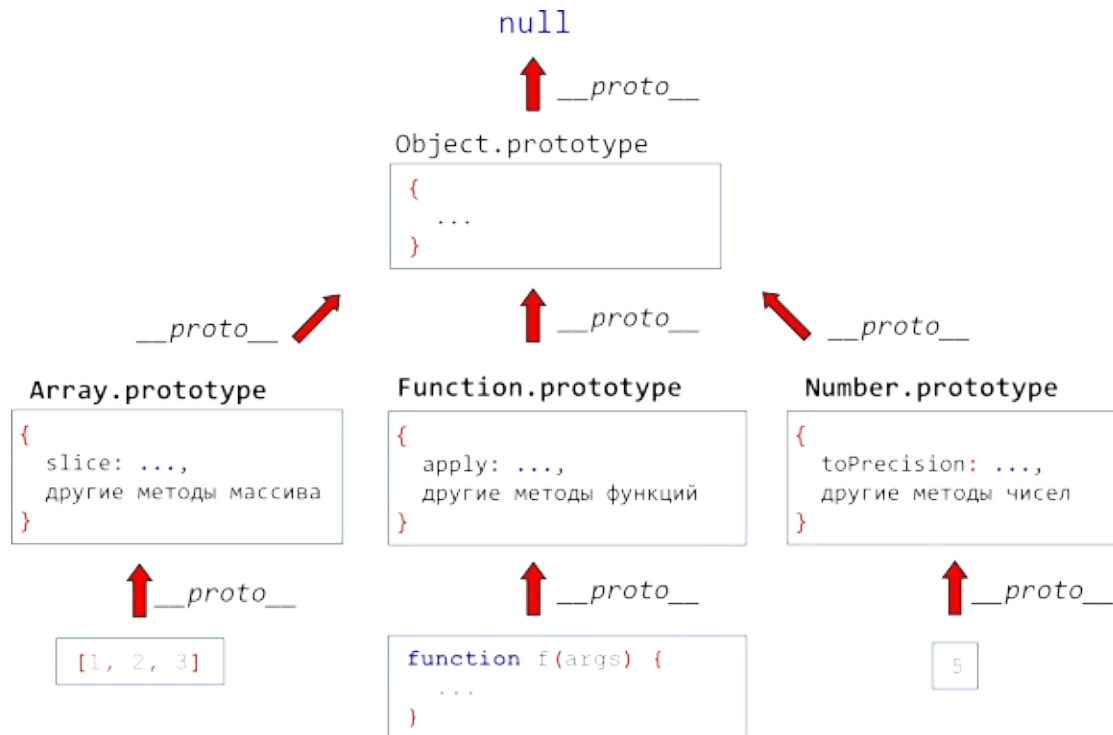
Это можно легко проверить:

```
var obj = { };  
  
// метод берётся из прототипа?  
alert(obj.toString == Object.prototype.toString); // true, да  
  
// проверим прототип в Firefox, Chrome (где есть __proto__)  
alert(obj.__proto__ == Object.prototype); // true
```



## Встроенные «классы» в JavaScript

Точно такой же подход используется в массивах `Array`, функциях `Function` и других объектах. Встроенные методы для них находятся в `Array.prototype`, `Function.prototype` и т.п.



Как видно, получается иерархия наследования, которая всегда заканчивается на `Object.prototype`. Объект `Object.prototype` — единственный, у которого `__proto__`.

Поэтому говорят, что «все объекты наследуют от `Object`». На самом деле ничего подобного. Это все прототипы наследуют от `Object.prototype`, равно `null`.

Некоторые методы при этом переопределяются. Например, у массива `Array` есть свой `toString`, который находится в `Array.prototype.toString`:

```
var arr = [1, 2, 3]

alert( arr ); // 1,2,3 <-- результат работы Array.prototype.toString
```

JavaScript ищет `toString`, сначала в `arr`, затем в `arr.__proto__ == Array.prototype`. Если бы там не нашёл — пошёл бы выше в `Array.prototype.__proto__`, который по стандарту (см. диаграмму выше) равен `Object.prototype`.

Методы `apply/call` у функций тоже берутся из встроенного прототипа `Function.prototype`.

```
function f() { }

alert( f.call == Function.prototype.call ); // true
```

## Объявляем свои «классы»

Термины «псевдокласс», «класс» отсутствуют в спецификации ES5. Но их используют, потому что подход, о котором мы будем говорить, похож на «классы», используемые в других языках программирования, таких как C++, Java, PHP и т.п.

Классом называют функцию-конструктор вместе с её `prototype`.

Например: «класс Object», «класс Date» — это примеры встроенных классов. Мы можем использовать тот же подход для объявления своих.

Чтобы задать класс, нужно:

1. Объявить функцию-конструктор.
2. Записать методы и свойства, нужные всем объектам класса, в `prototype`.

Опишем класс `Animal`:

```
// конструктор
function Animal(name) {
  this.name = name;
}

// методы в прототипе
Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert(this.name + ' бежит, скорость ' + this.speed);
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert(this.name + ' стоит');
};

// свойство speed со значением "по умолчанию"
Animal.prototype.speed = 0;

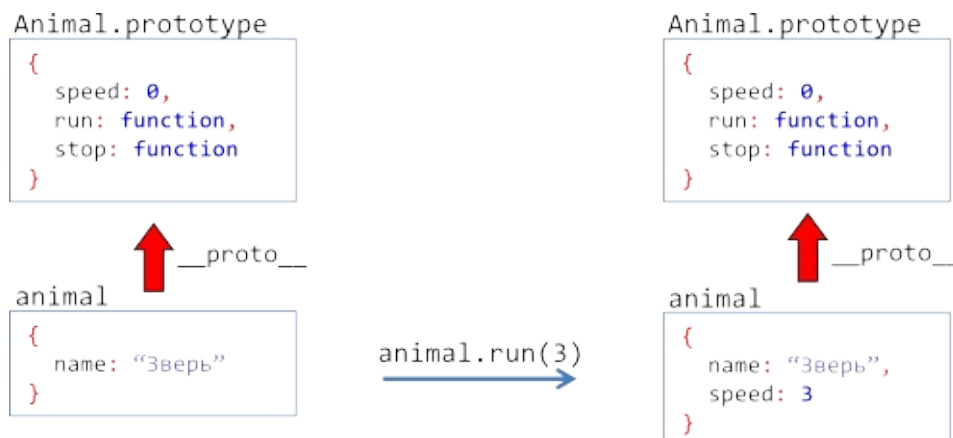
var animal = new Animal('Зверь');

alert(animal.speed);           // 0, свойство взято из прототипа

animal.run(5);                 // Зверь бежит, скорость 5
animal.run(5);                 // Зверь бежит, скорость 10
animal.stop();                 // Зверь стоит
```

Здесь объекту `animal` принадлежит лишь свойство `name`, а остальное находится в прототипе.

Вызовы `animal.run()`, `animal.stop()` в примере выше изменяют `this.speed`.



При этом начальное значение `speed` берётся из прототипа, а новое — пишется уже в сам объект. И в дальнейшем используется. Это вполне нормально, но здесь есть важная тонкость.

**Значения по умолчанию не следует хранить в прототипе в том случае, если это объекты.**

# Прототипное наследование

Источник: <http://dmitrypodgorniy.com/blog/2012/07/22/1/>

Автор: @dmitry (Дмитрий Подгорный)

В форме таких заключений прототипное наследование лежит у меня в голове. Примеры, подкрепляющие положения, смотреть в фаерфоксе или хrome, ибо `__proto__` не является частью стандарта. Разработчики хрома и фаерфокса позволили напрямую обращение к этому свойству.

1) Каждый объект имеет скрытое свойство `__proto__`, которое ссылается на некий другой объект (исключение — объект `Object.prototype`, у которого `__proto__` ссылается на `null`). Это и называется цепочкой прототипов.

```
var arr = [],
    obj = {};
typeof arr.__proto__ === 'object'; // true
typeof obj.__proto__ === 'object'; // true
arr.__proto__ === Array.prototype // true. Экземпляр класса "Массив" наследует от Array.prototype
arr.__proto__.__proto__ === Object.prototype // true он-же наследует от Object.prototype
arr.__proto__.__proto__.__proto__ // null Последнее звено цепи прототипов
```

2) При чтении свойства в объекте, оно ищется непосредственно в объекте. Если в объекте не находится, то интерпретатор ищет свойство в `__proto__`. Поиск останавливается при первом нахождении или когда ссылка `__proto__` ведет на `null`;

```
function F() {}
F.prototype.state = false; // состояние по умолчанию
F.prototype.state_on = function () {
    this.state = true; // запись свойства в экземпляр класса
}
F.prototype.reset_state = function () {
    delete this.state; // удаление свойства из экземпляра класса
}

var f = new F;
f.state; // false
f.state_on(); // добавили свойство в экземпляр класса
f.state; // true
f.reset_state(); // удаление свойства из объекта
f.state; // false (значение взялось из прототипа)
f.test; // undefined (свойства нет в объекте и цепочке прототипов)
```

3) `__proto__` устанавливает в момент создания объекта, и ссылается туда-же, куда и `Конструктор_объекта.prototype`;

4) Каждая функция при объявлении получает свойство `prototype`.

```
function F () {}
typeof F.prototype === 'object'; // true
var f = new F;
f.__proto__ === F.prototype; // true
```

# Задачи

## Задача 1

При выполнении этого кода вызов `rabbit.eat()` запишет в объект свойство `full`.

Вопрос — в какой объект: в `rabbit` или `animal`?

```
var animal = { };
var rabbit = { };

rabbit.__proto__ = animal;

animal.eat = function() {
  this.full = true;
};

rabbit.eat();
```

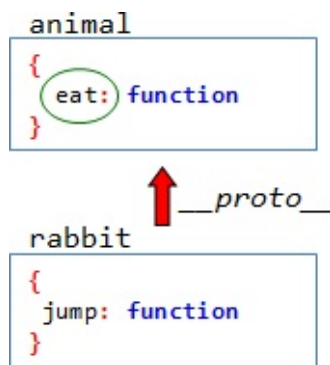
## Решение:

Ответ: свойство будет записано в `rabbit`.

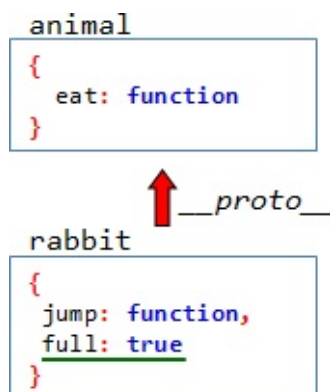
Если коротко — то потому что `this` будет указывать на `rabbit`, а прототип при записи не используется.

Если в деталях — посмотрим как выполняется `rabbit.eat()`:

1. Интерпретатор ищет `rabbit.eat`, чтобы его вызвать. Но свойство `eat` отсутствует в объекте `rabbit`, поэтому он идет по ссылке `rabbit.__proto__` и находит это свойство там.



2. Функция `eat` запускается. Контекст ставится равным объекту перед точкой, т.е. `this = rabbit`. Итак — получается, что команда `this.full = true` устанавливает свойство `full` в самом объекте `rabbit`. Итог:



## Задача 2

Какие значения будут выводиться в коде ниже?

```
var animal = { jumps: null };
var rabbit = { jumps: true };

rabbit.__proto__ = animal;

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;
alert( rabbit.jumps ); // ? (2)

delete animal.jumps;
alert( rabbit.jumps ); // ? (3)
```

Итого три вопроса.

## Решение

1. true, свойство взято из rabbit.
2. null, свойство взято из animal.
3. undefined, свойства больше нет.

## Задача 3

Есть объекты:

```
var head = {
  glasses: 1
};

var table = {
  pen: 3
};

var bed = {
  sheet: 1,
  pillow: 2
};

var pockets = {
  money: 2000
};
```

Задание состоит из двух частей:

1. Присвойте объектам ссылки `__proto__` так, чтобы любой поиск чего-либо шёл по алгоритму `pockets -> bed -> table -> head`. То есть `pockets.pen == 3`, `bed.glasses == 1`, но `table.money == undefined`.
2. После этого ответьте на вопрос, как быстрее искать `glasses`: обращением к `pockets.glasses` или `head.glasses`? Попробуйте протестировать.

## Решение:

1) Расставим `__proto__`:

```
var head = {
  glasses: 1
};

var table = {
  pen: 3
};
table.__proto__ = head;
```

```

var bed = {
  sheet: 1,
  pillow: 2
};
bed.__proto__ = table;

var pockets = {
  money: 2000
};
pockets.__proto__ = bed;

alert( pockets.pen ); // 3
alert( bed.glasses ); // 1
alert( table.money ); // undefined

```

2) В современных браузерах, с точки зрения производительности, нет разницы, брать свойство из объекта или прототипа. Они запоминают, где было найдено свойство и в следующий раз при запросе, к примеру, `pockets.glasses` начнут искать сразу в прототипе `head`.

## Задача 4

В примерах ниже производятся различные действия с `prototype`.

Каковы будут результаты выполнения? Почему?

```

function Rabbit() { }
Rabbit.prototype = { eats: true };

var rabbit = new Rabbit();

Rabbit.prototype = {};

alert(rabbit.eats);

```

А если код будет такой? (заменена одна строка):

```

function Rabbit(name) { }
Rabbit.prototype = { eats: true };

var rabbit = new Rabbit();

Rabbit.prototype.eats = false; // (*)

alert(rabbit.eats);

```

А такой? (заменена одна строка)

```

function Rabbit(name) { }
Rabbit.prototype = { eats: true };

var rabbit = new Rabbit();

delete Rabbit.prototype.eats; // (*)

alert(rabbit.eats);

```

А если бы в последнем коде вместо строки (\*) было `delete rabbit.eats` ?

Итого 4 вопроса.

## Решение

1. Результат: `true`. Свойство `prototype` всего лишь задаёт `__proto__` у новых объектов. Так что его изменение не

повлияет на `rabbit.__proto__`. Свойство `eats` будет получено из прототипа.

2. Результат: `false`. Свойство `Rabbit.prototype` и `rabbit.__proto__` указывают на один и тот же объект. В данном случае изменения вносятся в сам объект.
3. Результат: `undefined`. Удаление осуществляется из самого прототипа, поэтому свойство `rabbit.eats` больше взять неоткуда.
4. Результат был бы `true`, так как `delete rabbit.eats` попыталось бы удалить `eats` из `rabbit`, где его и так нет. А чтение в `alert` прошло бы из прототипа.

## Задача 5

Создадим новый объект, вот такой:

```
function Rabbit() { }
Rabbit.prototype.test = function() { alert(this); }

var rabbit = new Rabbit();
```

Есть ли разница между вызовами:

```
rabbit.test();
rabbit.__proto__.test();
Rabbit.prototype.test();
Object.getPrototypeOf(rabbit).test();
```

Какие из этих вызовов идентичны в браузере IE9+? А в Chrome?

## Решение

1. Первый вызов ставит `this == rabbit`, остальные ставят `this` равным прототипу, следуя правилу «`this` — объект перед точкой». При этом второй вызов не поддерживается в IE, т.к. свойство `__proto__` — нестандартное. А третий и четвёртый — идентичны. В Chrome идентичны три последних вызова.

## Задача 6

Вы — руководитель команды, которая разрабатывает игру, хомяковую ферму. Один из программистов получил задание создать класс «хомяк» (англ - "Hamster").

Объекты-хомяки должны иметь массив `food` для хранения еды и метод `found`, который добавляет к нему.

Ниже — его решение. При создании двух хомяков, если поел один — почему-то сытым становится и второй тоже.

В чём дело? Как поправить?

```
function Hamster() { }

Hamster.prototype.food = [ ]; // пустой "живот"

Hamster.prototype.found = function(something) {
    this.food.push(something);
};

// Создаём двух хомяков и кормим первого
speedy = new Hamster();
lazy = new Hamster();

speedy.found("яблоко");
speedy.found("орех");
```

```
alert(speedy.food.length); // 2
alert(lazy.food.length);   // 2 (!??)
```

## Решение

Давайте подробнее разберем происходящее при вызове `speedy.found("яблоко")` :

Интерпретатор ищет свойство `found` в `speedy` . Но `speedy` — пустой объект, т.к. `new Hamster` ничего не делает с `this` . Интерпретатор идёт по ссылке `speedy.__proto__` ( `==Hamster.prototype` ) и находят там метод `found` , запускает его. Значение `this` устанавливается в объект перед точкой, т.е. в `speedy` . Для выполнения `this.food.push()` нужно найти свойство `this.food` . Оно отсутствует в `speedy` , но есть в `speedy.__proto__` . Значение "яблоко" добавляется в `speedy.__proto__.food` . У всех хомяков общий живот! Или, в терминах JavaScript, свойство `food` изменяется в прототипе, который является общим для всех объектов-хомяков.

Заметим, что этой проблемы не было бы при простом присваивании:

```
this.food = something;
```

В этом случае значение записалось бы в сам объект, без поиска `found` в прототипе.

Проблема возникает только со свойствами-объектами в прототипе.

Для исправления проблемы нужно дать каждому хомяку свой живот. Это можно сделать, присвоив его в конструкторе.

```
function Hamster() {
  this.food = [];
}

Hamster.prototype.found = function(something) {
  this.food.push(something);
};

speedy = new Hamster();
lazy = new Hamster();

speedy.found("яблоко");
speedy.found("опех");

alert(speedy.food.length) // 2
alert(lazy.food.length)  // 0(!)
```

Теперь всё в порядке. У каждого хомяка — свой живот.



# Конспект. Функции, конструкторы, прототипы

Источник: <http://forum.jscourse.com/t/08-konspekt-funkczii-konstruktory-prototypy/601>

Автор конспекта: @eimrine

## this

Ключевое слово `this` возвращает контекст выполнения функции (также словом "контекст" называют замыкание, но `this` не имеет ничего общего с замыканиями). Значение `this` определяется только тем, как функция вызвана.

Есть несколько способов вызвать функцию:

- как функцию:

```
function f(){}  
f();
```

- как метод объекта:

```
var obj = {  
  f: function(){}  
}  
  
obj.f();
```

Когда функция вызывается как функция, `this` не представляет интереса. Функция интересна как те действия, которые спрятаны у неё внутри - результат или побочный эффект. Когда функция вызывается как метод объекта, тогда `this` это способ изнутри функции понять, у какого объекта этот метод вызывается.

**Контекст выполнения функции** `this` будет разным при каждом вызове. Однако, на протяжении хода контекста, значение `this` является неизменным, т.е. нельзя присвоить ему новое значение динамически в процессе исполнения, т.к. `this` не является переменной.

`this` удобно использовать в следующих целях:

1. Получения и использования метода из массива для последующего применения к массивоподобному объекту;
2. Создания и описания функции которая будет использована внутри класса для каждого из экземпляра класса;
3. в обработке событий

## call( ), apply( )

У каждой функции есть методы `call()` и `apply()`, который они наследуют от `Function.prototype`:

```
function f(){}  
f.call // function call(){native code}  
f.apply // function call(){native code}
```

Мы можем добавлять в функцию свойства так же, как в объект:

```
f.test = true  
f.test // true
```

Записать с круглыми скобками это значит её вызвать.

```
f.call()
```

или `f.apply()` тоже предназначены для вызова функции. Отличием `call()` и `apply()` от вызова функции напрямую является то, что этим функциям мы явно передаём `this` и аргументы в вызове:

```
fun.call(thisArg[, arg1[, arg2[, ...]])  
fun.apply(thisArg[, argsArray])
```

`thisArg` - контекст функции, другими словами `this`, который мы передаём в `call` или `apply`. `arg1, arg2, ...` - аргументы объекта; `call` позволяет передать аргументы через запятую. `argsArray` - массивоподобный объект; `apply` позволяет передать аргументы в виде массива. Метод объекта это такое свойство объекта, в значение которого записана функция. Рассмотрим метод `f` объекта `obj`.

```
obj.f(1,2,3,4)
```

Изнутри функции `f` это эквивалентно :

```
f.call(obj, 1,2,3,4); //(f.call в контексте obj)  
f.apply(obj, [1,2,3,4]); //(f.call в контексте obj с аргументами 1,2,3,4)
```

## new

Ещё один способ вызова функции это использование ключевого слова `new`.

Смысловая нагрузка `new` в том, что функция возвращает новый объект (еще говорят экземпляр класса). Не бывает функций, которые можно вызывать как с `new`, так и без него. Будет ли функция использоваться как конструктор решается в момент создания функции. Конструкторы объектов принято именовать с прописной буквы.

В ES5 на уровне языка нет понятия классов (зарезервированное слово `class` будет активно в ES6, но без `private/public`). Однако даже сейчас можно утверждать, что функция, которая использует прототип как конструктор, по которому задаются методы экземпляров класса, является классом. Поскольку в текущем стандарте EcmaScript де-факто нет классов, нам следует использовать термин **конструктор объектов**.

Попробуем создать пустой конструктор:

```
function F(){}  
new F(); // F {}
```

А теперь вызвать:

```
function F(){  
  // this ссылается на вновь создаваемый объект.  
  this.test = true; // запишет свойство test в создаваемый объект.  
}  
  
new F(); // F {test: true}
```

Вновь создаваемый объект это то, во что вычисляется `new F();`. На этапе создания объекта можно начинить его новыми свойствами, так же свойства можно написать новой функцией.

## prototype

Есть 2 способа определить функцию, которая будет конструировать объекты и есть 2 способа определить, как методы будут вызываться для этих объектов. Либо мы определим этот метод в самом объекте, либо в прототипе конструктора.

```
function F(){
  this.test = true;
}

F.prototype.isTest = function(){
  return this.test;
}

var f = new F();
f.isTest();
```

Помимо того что у этой функции конструируется новый объект, у этого объекта конструируется хитрая ссылка:

```
this.__proto__ === F.prototype
```

Каждая функция имеет свойство `prototype`. Потенциально каждая функция может стать конструктором новых объектов. Если вызвать функцию с ключевым словом `new`, то внутри создастся новый объект, на который будет ссылаться `this`. Свойство `__proto__` указывает туда же, куда и `F.prototype`.

Прототипное наследование можно описать в 4 пункта:

1. Каждый объект функции имеет свойство `__proto__`, которое ссылается на некий другой объект функции либо на `null`. Это называется цепочкой прототипов.
2. При чтении свойства в объекте, оно ищется непосредственно в объекте. Если в объекте не находится, то интерпретатор ищет свойство в `__proto__`. Поиск останавливается при первом нахождении.
3. `__proto__` устанавливается в момент создания объекта, и ссылается туда же, куда и `Конструктор_объекта.prototype`;
4. Каждая функция при объявлении получает свойство `prototype`.

### Цепочка прототипов:

```
var arr = []
arr.__proto__ // []
arr.__proto__.__proto__ // Object {}
arr.__proto__.__proto__.__proto__ // null

arr.__proto__ === Array.prototype // true
```

Если мы хотим создать новый тип данных, данные, что характерны в отдельности для данного instance этих данных, записываем в `this`, а те, которые должны быть характерны для каждого экземпляра этого типа пишем в `prototype`.

```
function F(){
  this.isTest = true;
}
F.prototype.isTest = function(){
  return this.test;
};
var f = new F();
console.dir(f) // способ посмотреть, что есть в f
```

В консоли получаем:

```
console.dir(f) // способ посмотреть, что есть в f
▼ F ⓘ
  isTest: true
  ▼ proto : F
    ► constructor: function F(){
    ► isTest: function (){
    ► __proto__: Object
  }
  < undefined
```

Важно отметить, что в ссылке на `isTest` (строка `isTest: function (){}`  хранится функция, которая записана в `F.prototype.isTest` :

```
F.isTest == F.prototype.isTest
```

## WalkyTalky

Уход от концепции "отдельно данные, отдельно функции" к концепции "класс, у экземпляров класса общие методы, но разные данные".

Есть код, который характеризует агента (точку) с определенной позицией. Изменяя значение координат `x,y` позицию можно менять (используя функцию `walk` ), а также сообщать текущую позицию в консоль( функция `talk` ).

```
var position = {
  x: 0,
  y: 0,
};
function walk(obj, x, y) {
  obj.x = x;
  obj.y = y;
}
function talk(obj) {
  console.log('I am at x:' + obj.x + ', y:' + obj.y);
}
walk(position, 10, 20);
talk(position); // I am at position x:10, y:20
```

В данном случае мы храним отдельно набор данных (объект `position` ), и функции (`walk, talk` ). Это не очень удобно, и поэтому их необходимо связать. Это можно сделать следующим способом: объявить метод (функцию) непосредственно в объекте с данными:

```
// object props
var position = {
  x: 0,
  y: 0,
  walk: function (x, y) {
    this.x += x;
    this.y += y;
  },
  talk: function () {
    console.log('I am at x:' + this.x + ', y:' + this.y);
  }
};
position.walk(20, 30);
position.talk(); // I am at position x:20, y:30
```

Таким образом изнутри функции понятно с какими данными функция работает. `this` ссылается на объект `position` . Технически получаем тот же результат, но код уже организован совершенно по-другому.

Но и этот вариант организации кода не исчерпывает все возможности javascript-а. Если, к примеру, необходимо создать много точек `position` , то будет необходимо создавать много одинаковых (по структуре) объектов и

вписывать в каждую из них одну и ту же функцию (пример кода не прилагается). Это неудобно, и для этого есть другие способы организации кода (см. ниже)

На заметку: группу одинаковых объектов, которые отличаются каким-то состоянием\данными, называют классом. И две одинаковых точки с одной и той же функцией называются элементами одного класса.

Для более удобного создания элементов одного класса используют отдельную функцию.

```
function createWalkyTalky(x, y) {
  return {
    x: x || 0,
    y: y || 0,
    walk: function (x, y) {
      this.x += x;
      this.y += y;
    },
    talk: function () {
      console.log('I am at x:' + this.x + ', y:' + this.y);
    }
  };
}
var position1 = createWalkyTalky(90, 49);
var position2 = createWalkyTalky(30);
position1.walk(20, 30);
position2.walk(-20, 15);
position1.talk();
position2.talk();
```

В этом коде прямо из функции возвращается новый `position`. Таким образом также избавляемся от дублирования кода. Взглянув на код можно заметить, что создание множества объектов, у которых похожи методы и свойства, выглядит как объявление новой функции которая каждый раз этот объект конструирует.

Чаще всего для того чтобы описать элементы одного класса (типа) необходимо создать функцию-конструктор (в нашем случае - `WalkyTalky`).

```
// class instance
function WalkyTalky(x, y) {
  this.x = x || 0;
  this.y = y || 0;
  this.walk = function (x, y) {
    this.x += x;
    this.y += y;
  };
  this.talk = function () {
    console.log('I am at x:' + this.x + ', y:' + this.y);
  };
}
var wt1 = new WalkyTalky(90, 49);
var wt2 = new WalkyTalky(30);
wt1.walk(20, 30);
wt2.walk(-20, 15);
wt1.talk(); //I am at x:110, y:79
wt2.talk(); // I am at x:10, y:15
//следует обратить внимание что позиция изменяется относительно предыдущей позиции, которая не всегда x:0, y:0;
```

Также можно использовать следующую конструкцию: не создавать каждый раз функции `walk` и `talk` непосредственно в объекте, а вынести их `proto` функции, а данные записывать непосредственно в сам объект.

```
// Prototype-base
function WalkyTalky(x, y) {
  this.x = x || 0;
  this.y = y || 0;
}
WalkyTalky.prototype.walk = function (x, y) {
  this.x += x;
  this.y += y;
};
WalkyTalky.prototype.talk = function () {
```

```
    console.log('I am at x:' + this.x + ', y:' + this.y);
};
var wt1 = new WalkyTalky(90, 49);
var wt2 = new WalkyTalky(30);
wt1.walk(20, 30);
wt2.walk(-20, 15);
wt1.talk(); //I am at x:110, y:79
wt2.talk(); //I am at x:10, y:15
```

Функция `walkyTalky` используется как конструктор, вызывается через `New`. В `this` записывается вновь создаваемый объект (по умолчанию пустой), в `proto` `this` записывается то, куда ссылается `walkyTalky.prototype`.

Использование прототипов позволяет использовать наследование. Например - создать класс, который делает тоже самое что и `walkyTalky` + еще что-то.

Таким образом осуществляется уход от концепции "отдельно данные, отдельно функции" к концепции "класс, у экземпляров класса общие методы, но разные данные".

Так выглядит цепочка прототипов `walkyTalky` :

```
> console.dir(wt1);
▼ WalkyTalky ⓘ
  x: 110
  y: 79
  proto : WalkyTalky
    ► constructor: function WalkyTalky(x, y) {
    ► talk: function () {
    ► walk: function (x, y) {
    ► __proto__: Object
< undefined
```

# Массивы, чтение

---

## Почитать:

<http://learn.javascript.ru/native-prototypes>

<http://learn.javascript.ru/inheritance-intro>

<http://habrahabr.ru/post/241587/>

# Расширение встроенных прототипов

Источник: <http://learn.javascript.ru/native-prototypes>

Встроенные в JavaScript объекты можно расширять и изменять. Что интересно, изменение некоторых из них повлияет и на примитивы. Можно добавить методы стандартным числам, строкам, и не только.

## Конструкторы String, Number, Boolean

Строки, числа, булевы значения в JavaScript являются примитивами. Но есть также и встроенные функции-конструкторы `String`, `Number`, `Boolean`. У единственного допустимого использования этих конструкторов — запуск в режиме обычной функции для преобразования типа. Например, `Number("12")` преобразует в число, так же как `+"12"`.

## Автопреобразование примитивов

Несмотря на то, что в явном виде объекты `String`, `Number`, `Boolean` не создаются, их прототипы всё же используются. Они хранят методы строк, чисел, булевых значений. Например, метод `slice` для строк хранится как `String.prototype.slice`.

При вызове метода на примитиве, например, `"строка".slice(1)`, происходит следующее:

1. Примитивное значение неявно преобразуется во временный объект `String`.
2. Затем ищется и вызывается метод прототипа: `String.prototype.slice`.
3. Результатом вызова `slice` является снова примитив, а временный объект уничтожается.

Посмотрим на интересное следствие такого поведения. Попытаемся добавить свойство строке:

```
var hello = "Привет мир!";

hello.test = 5; // запись свойства сработала, ошибки нет...

alert(hello.test); // ...читаем свойство -- выдаёт undefined!
```

Будет выведено `undefined`, так как присвоение произошло во временный объект, созданный для обработки обращения к свойству примитива. Этот временный объект тут же уничтожился вместе со свойством.

Конечно же, браузеры при таком преобразовании применяют оптимизации и, возможно, дополнительные объекты не создаются, но логика поведения — именно такая как описана.

А значит, **методы для строк, чисел, булевых значений можно изменять и добавлять свои, в прототип...**

## Изменение встроенных прототипов

**Встроенные прототипы можно изменять. В том числе — добавлять свои методы.**

Есть объекты, которые не участвуют в циклах `for...in`, например строки, функции... С ними уж точно нет такой проблемы, и в их прототипы, пожалуй, можно добавлять свои методы.

Но здесь есть свои «за» и «против»:

- Методы в прототипе позволяют писать более короткий и ясный код.
- Новые свойства, добавленные в прототип из разных мест, могут конфликтовать между собой. Представьте, что вы подключили две библиотеки, которые добавили одно и то же свойство в прототип, но определили его по-



разному. Конфликт неизбежен.

- Изменение встроенного прототипа влияет глобально, на весь код, и менять их не очень хорошо с архитектурной точки зрения.

Допустимо изменение прототипа встроенных объектов, которое добавляет поддержку метода из современных стандартов в те браузеры, где её пока нет.

Например, добавим `Object.create(proto)` в старые браузеры:

показать чистый исходник в новом окнеСкрыть/показать номера строкпечатать код с сохранением подсветки

```
if (!Object.create) {  
  
    Object.create = function(proto) {  
        function F() {}  
        F.prototype = proto;  
        return new F;  
    };  
  
}
```

## Итого

- Методы встроенных объектов хранятся в их прототипах.
- Встроенные прототипы можно расширить или поменять.
- Добавление методов в `Object.prototype` ломает циклы `for..in`. Другие прототипы изменять не настолько опасно, но все же не рекомендуется во избежание конфликтов.

# Область применения наследования

---

Источник: <http://learn.javascript.ru/inheritance-intro>

## Наследование для расширения

Причина, по которой возникают такие вопросы, очень проста. На начальном этапе разработки наследование действительно не нужно. Ведь наследование — это способ расширения существующего функционала. А если интерфейс состоит из трёх кнопок и одного меню — там нечего расширять.

С другой стороны, пусть у нас есть меню `Menu`, с простыми методами `close()`, `open()` и событием `select`.

Проект растёт. Через некоторое время может понадобиться создать расширенный вариант меню, например с анимацией при открытии. А в другом месте проекта будет нужно, чтобы меню загрузило своё содержимое с сервера при открытии... В третьем месте — ещё что-то.

Конечно, можно учитывать возрастающие требования и добавлять возможности в функцию-конструктор `Menu`, но чем дальше, тем более громоздким объект будет становиться, тем сложнее будет продолжать работу с ним, просто потому что возможностей масса, и все они в одном месте.

Если такое меню является частью библиотеки, то разработчик много раз подумает, прежде чем тащить в проект супер-универсальный комбайн с 50 методами, из которых ему нужны только 3. К тому же, работу с универсальным меню очень сложно осваивать, ведь много возможностей — это много документации. Поэтому и было придумано наследование.

При помощи наследования, описываются объекты, наследующие от меню и модифицирующие его поведение. Там, где нужна модификация — используется наследник, там где нет — обычное меню.

## Наследование для выноса общего функционала

Второй наиболее частый способ применения наследования — вынос базового функционала «за скобки», в общий родительский класс. Он применяется, в частности, для создания новых компонент.

Например, мы сделали в проекте меню `Menu`, дерево `Tree`, вкладки `Tabs` ... И замечаем, что во всех них используются общие методы для работы с документом, элементами.

Наследование позволяет нам вынести эти методы в «базовый компонент» `Widget`, так что `Menu`, `Tree` и другие компоненты расширяют его и получают их сразу.

# Тайная жизнь объектов

Источник: <http://habrahabr.ru/post/241587/>

Проблема объектно-ориентированных языков в том, что они тащат с собой всё своё неявное окружение. Вам нужен был банан – а вы получаете гориллу с бананом, и целые джунгли впридачу.

Джо Армстронг, в интервью *Coders at Work*

## Методы

Методы – свойства, содержащие функции. Простой метод:

```
var rabbit = {};  
rabbit.speak = function(line) {  
  console.log("Кролик говорит '" + line + "'");  
};  
  
rabbit.speak("Я живой.");  
// → Кролик говорит 'Я живой.'
```

Обычно метод должен что-то сделать с объектом, через который он был вызван. Когда функцию вызывают в виде метода – как свойство объекта, например `object.method()` – специальная переменная в её теле будет указывать на вызвавший её объект.

```
function speak(line) {  
  console.log("А " + this.type + " кролик говорит '" + line + "'");  
}  
var whiteRabbit = {type: "белый", speak: speak};  
var fatRabbit = {type: "толстый", speak: speak};  
  
whiteRabbit.speak("Ушки мои и усики, " + "я же наверняка опаздываю!");  
// → А белый кролик говорит ' Ушки мои и усики, я же наверняка опаздываю!'  
fatRabbit.speak("Мне бы сейчас морковочки.");  
// → А толстый кролик говорит ' Мне бы сейчас морковочки.'
```

Код использует ключевое слово `this` для вывода типа говорящего кролика.

Вспомните, что методы `apply` и `bind` принимают первый аргумент, который можно использовать для эмуляции вызова методов. Этот первый аргумент как раз даёт значение переменной `this`.

Есть метод, похожий на `apply`, под названием `call`. Он тоже вызывает функцию, методом которой является, только принимает аргументы как обычно, а не в виде массива. Как `apply` и `bind`, в `call` можно передать значение `this`.

```
speak.apply(fatRabbit, ["Отрыжка!"]);  
// → А толстый кролик говорит ' Отрыжка!'  
speak.call({type: "старый"}, "О, господи.");  
// → А старый кролик говорит 'О, господи.'
```

## Прототипы

В дополнение к набору свойств, почти у всех также есть прототип. Прототип – это ещё один объект, который используется как запасной источник свойств. Когда объект получает запрос на свойство, которого у него нет, это свойство ищется у его прототипа, затем у прототипа прототипа, и т.д.

Ну а кто же прототип пустого объекта? Это великий предок всех объектов, `Object.prototype`.

```
console.log(Object.getPrototypeOf({}) == Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Как и следовало ожидать, функция `Object.getPrototypeOf` возвращает прототип объекта.

Прототипические отношения в JavaScript выглядят как дерево, а в его корне находится `Object.prototype`. Он предоставляет несколько методов, которые появляются у всех объектов, типа `toString`, который преобразует объект в строковый вид.

Прототипом многих объектов служит не непосредственно `Object.prototype`, а какой-то другой объект, который предоставляет свои свойства по умолчанию. Функции происходят от `Function.prototype`, массивы — от `Array.prototype`.

```
console.log(Object.getPrototypeOf(isNaN) == Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) == Array.prototype);
// → true
```

У таких прототипов будет свой прототип — часто `Object.prototype`, поэтому он всё равно, хоть и не напрямую, предоставляет им методы типа `toString`.

Функция `Object.getPrototypeOf` возвращает прототип объекта. Можно использовать `Object.create` для создания объектов с заданным прототипом.

```
var protoRabbit = {
  speak: function(line) {
    console.log("A " + this.type + " кролик говорит '" + line + "'");
  }
};
var killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "убийственный";
killerRabbit.speak("ХРЯЯЯСЬ!");
// → А убийственный кролик говорит ' ХРЯЯЯСЬ!'
```

Прото-кролик работает в качестве контейнера свойств, которые есть у всех кроликов. Конкретный объект-кролик, например убийственный, содержит свойства, применимые только к нему, например, свой тип, и наследует разделяемые с другими свойства от прототипа.

## Конструкторы

Более удобный способ создания объектов, наследуемых от некоего прототипа — конструктор. В JavaScript вызов функции с предшествующим ключевым словом `new` приводит к тому, что функция работает как конструктор. У конструктора будет в распоряжении переменная `this`, привязанная к свеже созданному объекту, и если она не вернёт непосредственно другое значение, содержащее объект, этот новый объект будет возвращён вместо него.

Говорят, что объект, созданный при помощи `new`, является экземпляром конструктора.

Вот простой конструктор кроликов. Имена конструкторов принято начинать с заглавной буквы, чтобы отличать их от других функций.

```
function Rabbit(type) {
  this.type = type;
}

var killerRabbit = new Rabbit("убийственный");
var blackRabbit = new Rabbit("чёрный");
console.log(blackRabbit.type);
// → чёрный
```

Конструкторы (а вообще-то, и все функции) автоматически получают свойство под именем `prototype`, которое по умолчанию содержит простой и пустой объект, происходящий от `Object.prototype`. Каждый экземпляр, созданный этим конструктором, будет иметь этот объект в качестве прототипа. Поэтому, чтобы добавить кроликам, созданным конструктором `Rabbit`, метод `speak`, мы просто можем сделать так:

```
Rabbit.prototype.speak = function(line) {
  console.log("А " + this.type + " кролик говорит '" + line + "'");
};
blackRabbit.speak("Всем капец...");
// → А чёрный кролик говорит ' Всем капец...'
```

Важно отметить разницу между тем, как прототип связан с конструктором (через свойство `prototype`) и тем, как у объектов есть прототип (который можно получить через `Object.getPrototypeOf`). На самом деле прототип конструктора — `Function.prototype`, поскольку конструкторы — это функции. Его свойство `prototype` будет прототипом экземпляров, созданных им, но не его прототипом.

### Перезагрузка унаследованных свойств

Когда вы добавляете свойство объекту, есть оно в прототипе или нет, оно добавляется непосредственно к самому объекту. Теперь это его свойство. Если в прототипе есть одноимённое свойство, оно больше не влияет на объект. Сам прототип не меняется.

```
Rabbit.prototype.teeth = "мелкие";
console.log(killerRabbit.teeth);
// → мелкие
killerRabbit.teeth = "длинные, острые и окровавленные ";
console.log(killerRabbit.teeth);
// → длинные, острые и окровавленные
console.log(blackRabbit.teeth);
// → мелкие
console.log(Rabbit.prototype.teeth);
// → мелкие
```

На диаграмме нарисована ситуация после прогона кода. Прототипы `Rabbit` и `Object` находятся за `killerRabbit` на манер фона, и у них можно запрашивать свойства, которых нет у самого объекта.

Перезагрузка свойств, существующих в прототипе, часто приносит пользу. Как в примере с зубами кролика её можно использовать для выражения каких-то исключительных характеристик у более общих свойств, в то время как обычные объекты просто используют стандартные значения, взятые у прототипов.

Также она используется для назначения функциям и массивам разных методов `toString`.

```
console.log(Array.prototype.toString == Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Вызов `toString` массива выводит результат, похожий на `.join(",")` — получается список, разделённый запятыми. Вызов `Object.prototype.toString` напрямую для массива приводит к другому результату. Эта функция не знает ничего о массивах:

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

## Нежелательное взаимодействие прототипов

Прототип помогает в любое время добавлять новые свойства и методы всем объектам, которые основаны на нём. К примеру, нашим кроликам может понадобиться танец.

```
Rabbit.prototype.dance = function() {
  console.log("А " + this.type + " кролик танцует джигу.");
};
killerRabbit.dance();
// → А убийственный кролик танцует джигу.
```

Это удобно. Но в некоторых случаях это приводит к проблемам. В предыдущих главах мы использовали объект как способ связать значения с именами — мы создавали свойства для этих имён, и давали им соответствующие значения. Вот пример из 4-й главы:

```
var map = {};
function storePhi(event, phi) {
  map[event] = phi;
}

storePhi("пицца", 0.069);
storePhi("тронул дерево", -0.081);
```

Мы можем перебрать все значения фи в объекте через цикл `for/in`, и проверить наличие в нём имени через оператор `in`. К сожалению, нам мешается прототип объекта.

```
Object.prototype.nonsense = "ку";
for (var name in map)
  console.log(name);
// → пицца
// → тронул дерево
// → nonsense
console.log("nonsense" in map);
// → true
console.log("toString" in map);
// → true

// Удалить проблемное свойство
delete Object.prototype.nonsense;
```

Это же неправильно. Нет события под названием “nonsense”. И тем более нет события под названием “toString”.

Занятно, что `toString` не вылезло в цикле `for/in`, хотя оператор `in` возвращает `true` на его счёт. Это потому, что JavaScript различает счётные и несчётные свойства.

Все свойства, которые мы создаём, назначая им значение — счётные. Все стандартные свойства в `Object.prototype` — несчётные, поэтому они не вылезают в циклах `for/in`.

Мы можем объявить свои несчётные свойства через функцию `Object.defineProperty`, которая позволяет указывать тип создаваемого свойства.

```
Object.defineProperty(Object.prototype, "hiddenNonsense", {enumerable: false, value: "ку"});
for (var name in map)
  console.log(name);
// → пицца
// → тронул дерево
console.log(map.hiddenNonsense);
// → ку
```

Теперь свойство есть, а в цикле оно не вылезает. Хорошо. Но нам всё ещё мешает проблема с оператором `in`, который утверждает, что свойства `Object.prototype` присутствуют в нашем объекте. Для этого нам понадобится метод `hasOwnProperty`.

```
console.log(map.hasOwnProperty("toString"));
// → false
```

Он говорит, является ли свойство свойством объекта, без оглядки на прототипы. Часто это более полезная информация, чем выдаёт оператор `in`.

Если вы волнуетесь, что кто-то другой, чей код вы загрузили в свою программу, испортил основной прототип объектов, я рекомендую писать циклы `for/in` так:

```
for (var name in map) {
  if (map.hasOwnProperty(name)) {
    // ... это наше личное свойство
  }
}
```

## Объекты без прототипов

Но кроличья нора на этом не заканчивается. А если кто-то зарегистрировал имя `hasOwnProperty` в объекте `map` и назначил ему значение 42? Теперь вызов `map.hasOwnProperty` обращается к локальному свойству, в котором содержится номер, а не функция.

В таком случае прототипы только мешаются, и нам бы хотелось иметь объекты вообще без прототипов. Мы видели функцию `Object.create`, что позволяет создавать объект с заданным прототипом. Мы можем передать `null` для прототипа, чтобы создать свеженький объект без прототипа. Это то, что нам нужно для объектов типа `map`, где могут быть любые свойства.

```
var map = Object.create(null);
map["пицца"] = 0.069;
console.log("toString" in map);
// → false
console.log("пицца" in map);
// → true
```

Так-то лучше! Нам уже не нужна приבלуда `hasOwnProperty`, потому что все свойства объекта заданы лично нами. Мы спокойно используем циклы `for/in` без оглядки на то, что люди творили с `Object.prototype`.

## Итог

Получается, что объекты чуть более сложны, чем я их подавал сначала. У них есть прототипы – это другие объекты, и они ведут себя так, как будто у них есть свойство, которого на самом деле нет, если это свойство есть у прототипа. Прототипом простых объектов является `Object.prototype`.

**Конструкторы** – функции, имена которых обычно начинаются с заглавной буквы, - можно использовать с оператором `new` для создания объектов. Прототипом нового объекта будет объект, содержащийся в свойстве `prototype` конструктора. Это можно использовать, помещая свойства, которые делят между собой все величины данного типа, в их прототип. Оператор `instanceof`, если ему дать объект и конструктор, может сказать, является ли объект экземпляром этого конструктора.

Для объектов можно сделать интерфейс и сказать всем, чтобы они общались с объектом только через этот интерфейс. Остальные детали реализации объекта теперь инкапсулированы, скрыты за интерфейсом.

А после этого никто не запрещал использовать разные объекты при помощи одинаковых интерфейсов. Если разные объекты имеют одинаковые интерфейсы, то и код, работающий с ними, может работать с разными объектами одинаково. Это называется полиморфизмом, и это очень полезная штука.

Определяя несколько типов, различающихся только в мелких деталях, бывает удобно просто наследовать прототип нового типа от прототипа старого типа, чтобы новый конструктор вызывал старый. Это даёт вам тип объекта,

сходный со старым, но при этом к нему можно добавлять свойства или переопределять старые.



# Задачи

---

## Задача 1

Добавьте всем функциям в прототип метод `defer(ms)`, который откладывает вызов функции на `ms` миллисекунд.

После этого должен работать такой код:

показать чистый исходник в новом окнеСкрыть/показать номера строкпечатать код с сохранением подсветки

```
function f() {  
  alert("привет");  
}  
  
f.defer(1000); // выведет "привет" через 1 секунду
```

## Решение

```
Function.prototype.defer = function(ms) {  
  setTimeout(this, ms);  
}  
  
function f() {  
  alert("привет");  
}  
  
f.defer(1000); // выведет "привет" через 1 секунду
```

# DOM (document object model)

---

Почитать:

<http://habrahabr.ru/post/243311/>

<http://learn.javascript.ru/browser-environment>

<http://learn.javascript.ru/browser-objects>

<http://learn.javascript.ru/dom-console>

# JavaScript и браузер

ИСТОЧНИК: <http://habrahabr.ru/post/243311/>

## Be6

Каждый документ имеет имя в виде универсального локатора ресурсов, Universal Resource Locator (URL), который выглядит примерно так:

http://eloquentjavascript.net/12\_browser.html

протокол	сервер	путь

Первая часть говорит нам, что URL использует протокол HTTP (в отличие от, скажем, зашифрованного HTTP, который записывается как `https://`). Затем идёт часть, определяющая, с какого сервера мы запрашиваем документ. Последняя – строка пути, определяющая конкретный документ или ресурс.

# HTML и JavaScript

HTML, или язык разметки гипертекста, Hypertext Markup Language – формат документа, использующийся для веб-страниц. HTML содержит текст и теги, придающие тексту структуру, описывающие такие вещи, как ссылки, параграфы и заголовки. контексте нашей книги самый главный тег HTML script . Он позволяет включать в документ программу на JavaScript.

Тег HTML script позволяет включать в документ программу на JavaScript.

```
<h1>Внимание, тест.</h1>  
<script>alert("Привет!");</script>
```

Такой скрипт запустится сразу, как только браузер встретит тег `script` при разборе HTML. На странице появится диалог-предупреждение.

У тега script есть атрибут src, чтобы запрашивать файл со скриптом (текст, содержащий программу на JavaScript) с адреса URL.

```
<h1>Внимание, тест.</h1>  
<script src="code/hello.js"></script>
```

Некоторые атрибуты тоже могут содержать программу JavaScript. У тега (на странице он выглядит как кнопка) есть атрибут `onClick`, и его содержимое будет запущено, когда по кнопке щёлкнут мышкой.

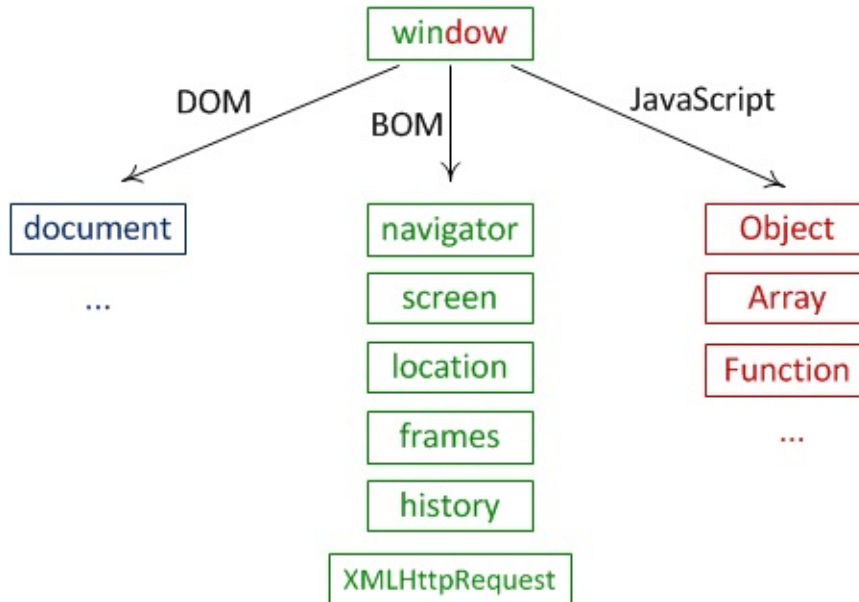
```
<button onclick="alert('Бабах!');">НЕ ЖМИ</button>
```

Заметьте, что я использовал одинарные кавычки для строки в атрибуте onclick, поскольку двойные кавычки уже используются в самом атрибуте. Можно было бы использовать `&quot;` , но это бы затруднило чтение.

# Окружение: DOM, BOM и JS

Источник: <http://learn.javascript.ru/browser-environment>

Браузер дает доступ к иерархии объектов, которые мы можем использовать для разработки. На рисунке схематически отображена структура основных браузерных объектов.



На вершине стоит `window`, который еще называют **глобальным объектом**.

Все остальные объекты делятся на 3 группы.

1. **Объектная модель документа (DOM).** Доступна через `document`. Дает доступ к содержимому страницы. На странице [W3C DOM](#) вы можете найти стандарты DOM, разработанные самим W3C. На данный момент существует 3 уровня DOM. Современные браузеры также поддерживают некоторые возможности, которые называются DOM 0 и которые остались еще с той эпохи, когда не было W3C.
2. **Объектная модель браузера (BOM).** BOM — это объекты для работы с чем угодно, кроме документа. Доступ к фреймам, запросы к серверу, функции `alert/confirm/prompt` — все это BOM. Большинство возможностей BOM стандартизированы в HTML5, но браузеры любят изобрести что-нибудь своё, особенное.
3. **Объекты и функции JavaScript.** Javascript — связующий все это язык. Встроенные в него объекты и сам язык в идеале должны соответствовать стандарту ECMA-262, но пока что браузеры к этому не пришли. Хотя положительная тенденция есть.

Глобальный объект `window` имеет две роли:

1. Это окно браузера. У него есть методы `window.focus()`, `window.open()` и другие.
2. Это глобальный объект JavaScript.

Вот почему он на рисунке представлен зеленым и красным цветом.

# BOM-объекты: navigator, screen, location, frames

Источник: <http://learn.javascript.ru/browser-objects>

## navigator: платформа и браузер

Объект `navigator` содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства:

- `navigator.userAgent` — содержит информацию о браузере.
- `navigator.platform` — содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п..

```
alert(navigator.userAgent);
alert(navigator.platform);
```

## screen

Объект `screen` содержит общую информацию об экране, включая его разрешение, цветность и т.п. Оно может быть полезно для определения, что код выполняется на мобильном устройстве с маленьким разрешением.

Текущее разрешение экрана посетителя по горизонтали/вертикали находится в `screen.width / screen.height`.

Это свойство можно использовать для сбора статистической информации о посетителях.

JavaScript-код счетчиков считывает эту информацию и отправляет на сервер. Именно поэтому можно просматривать в статистике, сколько посетителей приходило с каким экраном.

## location

Объект `location` предоставляет информацию о текущем URL и позволяет JavaScript перенаправить посетителя на другой URL. Значением этого свойства является объект типа `Location`.

## Методы и свойства Location

Самый главный метод — это, конечно же, `toString`. Он возвращает полный URL.

Код, которому нужно провести строковую операцию над `location`, должен сначала привести объект к строке. Вот так будет ошибка:

```
// будет ошибка, т.к. location - не строка
alert( window.location.indexOf('/://') );
```

А так - правильно:

```
// привели к строке перед indexOf
alert( (window.location + '').indexOf('/://') );
```

Все следующие свойства являются строками. Колонка «Пример» содержит их значения для тестового URL:

- <http://www.google.com:80/search?q=javascript#test>

Свойство	Описание	Пример
----------	----------	--------

hash	часть URL, которая идет после символа решетки '#', включая символ '#'	#test
host	хост и порт	www.google.com:80
href	весь URL	<a href="http://www.google.com:80/search?q=javascript#test">http://www.google.com:80/search?q=javascript#test</a>
hostname	хост (без порта)	www.google.com
pathname	строка пути (относительно хоста)	/search
port	номер порта (если порт не указан, то пустая строка)	80
protocol	протокол	http: (двоеточие на конце)
search	часть адреса после символа "?", включая символ "?"	?q=javascript

## Методы объекта Location

1. **assign(url)** загрузить документ по данному url. Можно и просто приравнять `window.location.href = url`.
2. **reload([forceget])** перезагрузить документ по текущему URL. Аргумент `forceget` - булево значение, если оно true, то документ перезагружается всегда с сервера, если false или не указано, то браузер может взять страницу из своего кэша.
3. **replace(url)** заменить текущий документ на документ по указанному url.
4. **toString()** Возвращает строковое представление URL.

При изменении любых свойств `window.location`, кроме `hash`, документ будет перезагружен, как если бы для модифицированного url был вызван метод `window.location.assign()`.

Можно перенаправить и явным присвоением `location`, например:

```
// браузер загрузит страницу http://javascript.ru
window.location = "http://javascript.ru";
```

## frames

Коллекция, содержащая фреймы и ифреймы. Можно обращаться к ним как по номеру, так и по имени.

**В frames содержатся window-объекты дочерних фреймов.** Следующий код переводит фрейм на новый URL:

```
<iframe name="example" src="http://example.com" width="200" height="100"></iframe>

<script>
  window.frames.example.location = 'http://example.com';
</script>
```

## history

Объект `history` позволяет менять URL без перезагрузки страницы (в пределах того же домена) при помощи [History API](#), а также перенаправлять посетителя назад-вперед по истории.

Объект `history` не предоставляет возможности читать историю посещений. Можно отправить посетителя назад вызовом `history.back()` или вперед вызовом `history.forward()`, но сами адреса браузер не дает из соображений безопасности.

## Итого

Браузерные объекты:

- **navigator, screen** Содержат информацию о браузере и экране.
- **location** Содержит информацию о текущем URL и позволяет её менять. Любое изменение, кроме `hash` , перегружает страницу. Также можно перегрузить страницу с сервера вызовом `location.reload(true)` .
- **frames** Содержит коллекцию window-объектов для каждого из дочерних фреймов. Каждый фрейм доступен по номеру (с нуля) или по имени, что обычно удобнее.
- **history** Позволяет отправить посетителя на предыдущую/последующую страницу по истории, а также изменить URL без перезагрузки страницы с использованием History API.

# Работа с DOM из консоли

Источник: <http://learn.javascript.ru/dom-console>

## Доступ к элементу

Чтобы проанализировать любой элемент:

- Выберите его во вкладке Elements.
- Либо внизу вкладки Elements есть лупа, при нажатии на которую можно выбрать элемент кликом.
- Либо, что обычно удобнее всего, просто кликните на нужном месте на странице правой кнопкой и выберите в меню «Проверить Элемент».

## Elements - Консоль

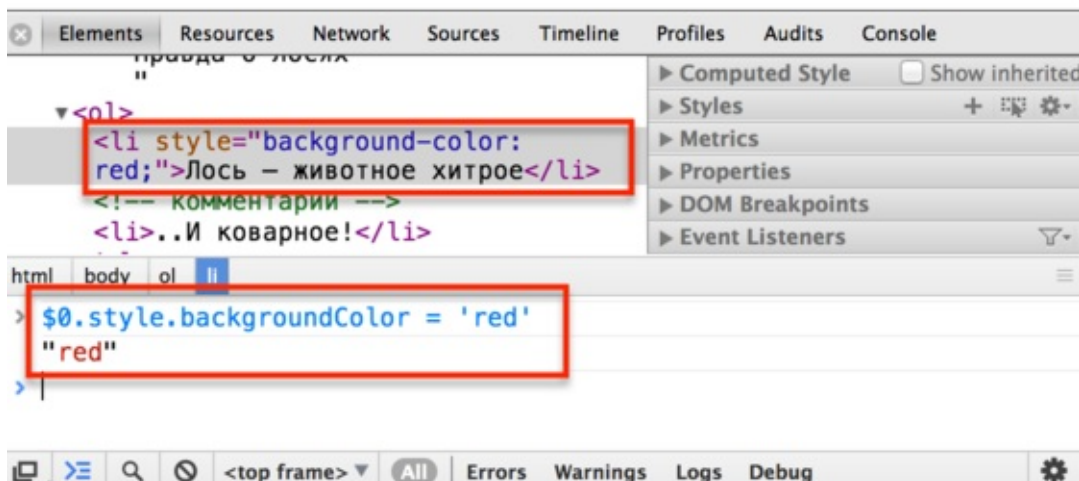
Зачастую бывает нужно выбрать элемент DOM и сделать с ним что-то на JavaScript.

Консоль можно либо открыть тут же, нажатием Esc, либо выбрать на соответствующую вкладку.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как `$0`, предыдущий — `$1` и так далее.

Правда о лосях

1. Лось — животное хитрое
2. ..И коварное!



## Ещё методы консоли

Для поиска элементов в консоли есть два специальных метода:

- `$("#div.my")` — возвращает элемент по CSS-селектору, только один (первый).
- `$$("div.my")` — возвращает массив элементов по CSS-селектору.



# Читаем про html, css

---

<http://htmlbook.ru/samhtml/struktura-html-koda>

<http://htmlbook.ru/samhtml/znacheniya-atributov-tegov>

<http://htmlbook.ru/samhtml/ssylki>

<http://htmlbook.ru/samhtml/izobrazheniya>

<http://htmlbook.ru/samhtml/spiski/markirovannyy-spisok>

<http://htmlbook.ru/samhtml/spiski/spisok-opredeleniy>

<http://htmlbook.ru/samcss/preimushchestva-stiley>

<http://htmlbook.ru/samcss/bazovyy-sintaksis-css>

<http://htmlbook.ru/samcss/znacheniya-stilevykh-svoystv>

Не смотрите, что статья старая. Умозаключения актуальны по сей день:

<http://softwaremaniacs.org/blog/2005/06/08/juice-and-flies/>

# Верстка

---

## Почитать:

Про работу со свойствами:

- display: inline, block, inline-block
- position
- top, left
- margin,
- padding,
- float, clear

Объяснение флоатов:

<http://softwaremaniacs.org/blog/2005/12/01/css-layout-float/>

Про поток:

<http://softwaremaniacs.org/blog/2005/08/27/css-layout-flow/>

Хороший tutorial про верстку:

<http://habrahabr.ru/post/202408/>

# Верстка

---

На следующем занятии продолжаем разговаривать про верстку, продолжим верстать страничку.



# DOM

---

## Почитать:

<http://www.howtocreate.co.uk/tutorials/javascript/domstructure>

<http://learn.javascript.ru/dom-nodes> READ

<http://learn.javascript.ru/traversing-dom> READ

<http://learn.javascript.ru/basic-dom-node-properties> READ

<http://learn.javascript.ru/modifying-document> READ

# Дерево DOM

Источник: <http://learn.javascript.ru/dom-nodes>

Основным инструментом работы и динамических изменений на странице является **DOM (Document Object Model)** — объектная модель, используемая для XML/HTML-документов.

Согласно DOM-модели, документ является иерархией, деревом.

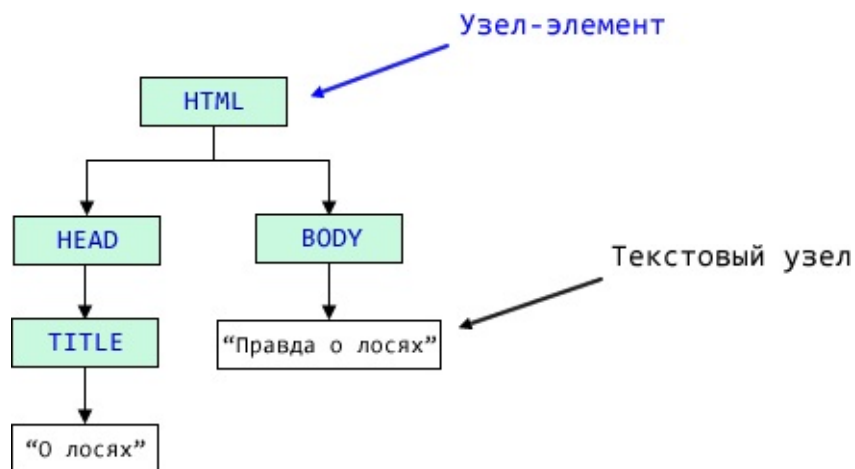
Каждый HTML-тег образует узел дерева с типом «элемент». Вложенные в него теги становятся дочерними узлами. Для представления текста создаются узлы с типом «текст».

Проще говоря, **DOM** — это представление документа в виде дерева тегов, доступное для изменения через **JavaScript**.

## Пример DOM

Построим, для начала, дерево DOM для следующего документа.

```
<html>
  <head>
    <title>О лосях</title>
  </head>
  <body>
    Правда о лосях
  </body>
</html>
```



В этом дереве выделено два типа узлов.

1. Теги образуют узлы-элементы (element node) DOM-дерева. Естественным образом одни узлы вложены в другие.
2. Текст внутри элементов образует текстовые узлы. Текстовый узел содержит исключительно строку текста и не может иметь потомков.

## Ещё узлы

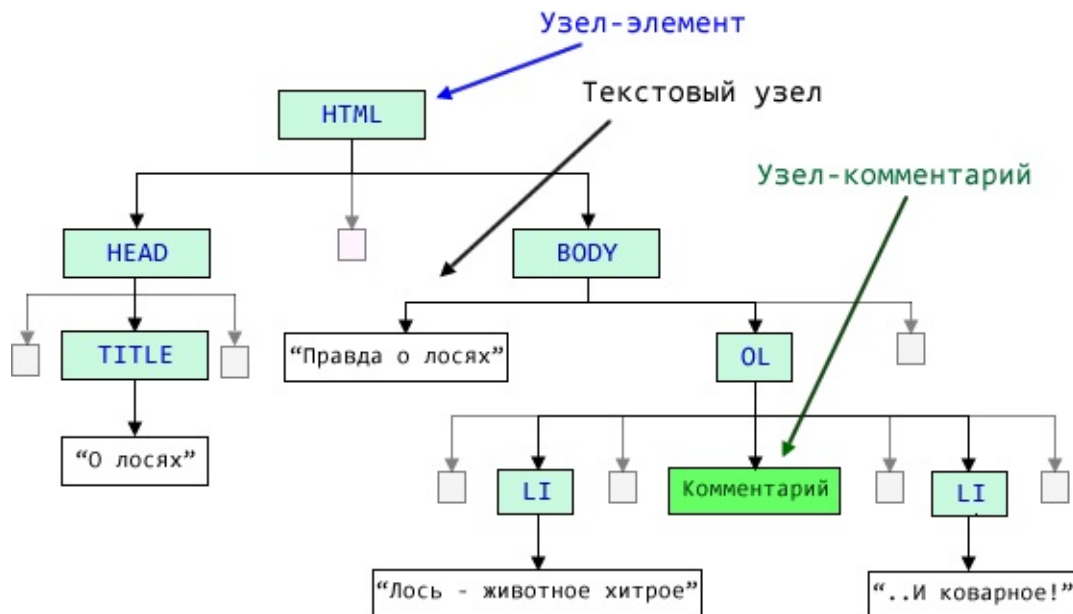
Дополним страницу новыми тегами и комментарием:

```
<!DOCTYPE HTML>
<html>
  <head>
```

```

<title>0 лосях</title>
</head>
<body>
  Правда о лосях
  <ol>
    <li>Лось — животное хитрое</li>
    <!-- комментарий -->
    <li>..и коварное!</li>
  </ol>
</body>
</html>

```



В этом примере тегов уже больше, и даже появился узел нового типа — **узел-комментарий**. Казалось бы, зачем комментарий в DOM? На отображение-то он всё равно не влияет. Но так как он есть в HTML — обязан присутствовать в DOM-дереве.

Всё, что есть в HTML, находится и в DOM.

**Поскольку DOM-модель в точности соответствует документу, пробельные символы так же важны, как и любой другой текст.**

На картинке выше текстовые узлы, содержащие пробелы, выделены серым. Единственное исключение — пробелы перед и между `HEAD / BODY` на самом верхнем уровне документа. В DOM их никогда нет, согласно требованиям стандарта. Все остальные пробелы сохраняются в точности.

Пробелы есть в DOM, только если они есть в документе. Если не будет пробелов между тегами — не будет и пробельных узлов.

Следующий документ вообще не содержит пробельных узлов:

```

<!DOCTYPE HTML><html><head><title>Title</title></head><body></body></html>

```

В IE до версии 9 пробельных узлов нет. DOCTYPE — тоже узел.

## Возможности, которые дает DOM

**DOM нужен для того, чтобы манипулировать страницей — читать информацию из HTML, создавать и изменять элементы.** Фактически, DOM предоставляет возможность делать со страницей всё, что угодно.

## Итого

- DOM-модель — это внутреннее представление HTML-страницы в виде дерева.
- Все элементы страницы, включая теги, текст, комментарии, являются узлами DOM.
- У элементов DOM есть свойства и методы, которые позволяют изменять их.
- Кстати, DOM-модель используется не только в JavaScript, это известный способ представления XML-документов.



# Навигация в DOM, свойства-ссылки

Источник: <http://learn.javascript.ru/traversing-dom>

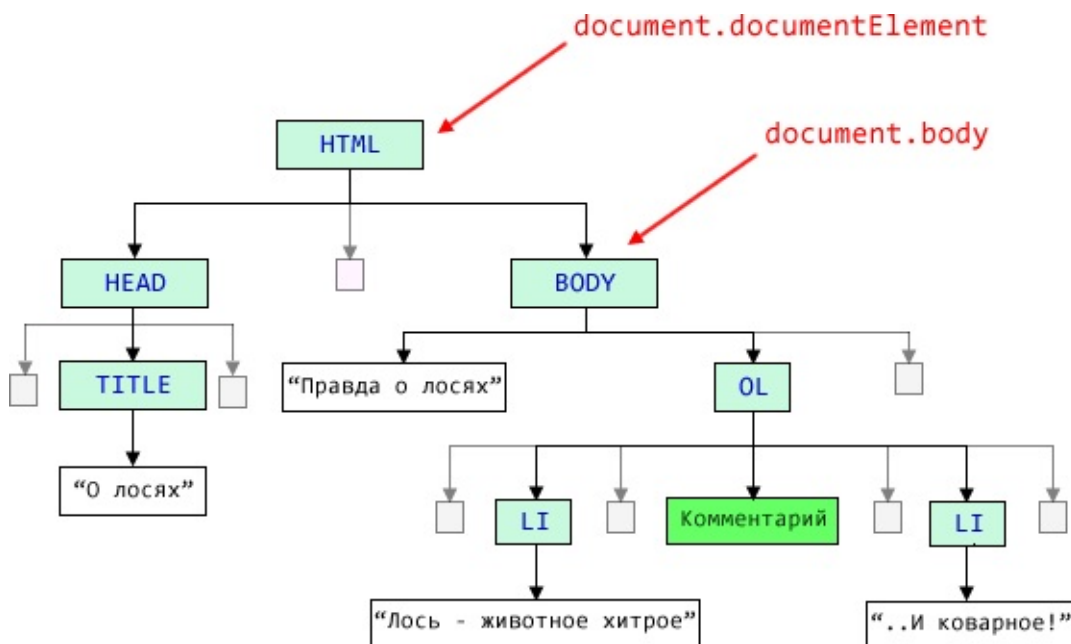
Для того, чтобы изменить узел DOM, например, раскрыть в нем меню, нужно сначала его получить.

Доступ к DOM начинается с `document`. Оттуда можно добраться до любых других узлов.

## Корень: `documentElement` и `body`

Войти в «корень» дерева можно двумя путями.

1. Первая точка входа — `document.documentElement`. Это свойство ссылается на DOM-объект для тега HTML.
2. Вторая точка входа — `document.body`, который соответствует тегу BODY.



Оба варианта отлично работают. Но есть одна тонкость: `document.body` может быть равен `null`.

Например, при доступе к `document.body` в момент обработки тега HEAD, то `document.body = null`. Это вполне логично, потому что BODY еще не существует.

Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.

В следующем примере, первый `alert` выведет `null`:

```
<!DOCTYPE HTML>
<html>
  <head>
    <script>
      alert("Из HEAD: " + document.body); // null
    </script>
  </head>
  <body>

    <script>
      alert("Из BODY: " + document.body);
    </script>

  </body>
</html>
```

В мире DOM для свойств-ссылок на узлы в качестве значения «нет такого элемента» или «узел не найден» используется не `undefined`, а `null`.

## Дочерние элементы

Из узла-родителя можно получить все дочерние элементы. Для этого есть несколько способов.

### childNodes

Псевдо-массив `childNodes` хранит все дочерние элементы, включая текстовые.

Пример ниже последовательно выведет дочерние элементы `document.body`:

```
<!DOCTYPE HTML>
<html>
  <head><meta charset="utf-8"></head>
  <body>
    <div>Пользователи:</div>
    <ul>
      <li>Маша</li>
      <li>Вовочка</li>
    </ul>

    <!-- комментарий -->

    <script>
      var childNodes = document.body.childNodes;
      console.log(childNodes);
    </script>

  </body>
</html>
```

### children

А что если текстовые узлы не нужны? Для этого существует свойство `children`, которое перечисляет только дочерние узлы, соответствующие тегам.

Посмотрим следующий пример. Он идентичен предыдущему, за исключением того, что в нем используется `children` вместо `childNodes`. Поэтому он будет выводить не все узлы, а только узлы-элементы.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8"></head>
  <body>
    <div>Пользователи:</div>
    <ul>
      <li>Маша</li>
      <li>Вовочка</li>
    </ul>

    <!-- комментарий -->

    <script>
      var children = document.body.children;
      console.log(children);
    </script>

  </body>
</html>
```

## Ссылки вверх и вниз

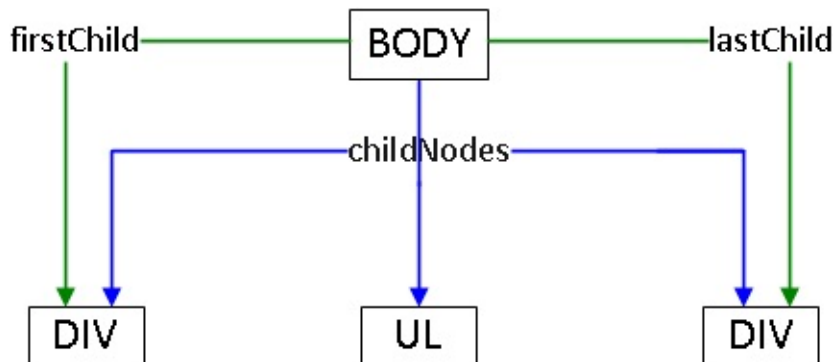
Для комфортного перемещения по узлам существуют дополнительные свойства, указывающие вверх, вниз, на соседей и т.п.

## firstChild и lastChild

Свойства `firstChild` и `lastChild` обеспечивают быстрый доступ к первому и последнему потомку.

Например, для документа:

```
<html><body><div>...</div><ul>...</ul><div>...</div></body></html>
```



Почему документ в одну строчку? Да потому что если добавить пробелов, то первым и последним узлами будут не элементы, а пробельные узлы.

В любом случае `firstChild` и `lastChild` — это более быстрый и короткий способ обратиться к первому и последнему элементам `childNodes`. Верны равенства:

```
body.firstChild === body.childNodes[0]
body.lastChild === body.childNodes[body.childNodes.length-1]
```

## parentNode, previousSibling и nextSibling

- Свойство `parentNode` ссылается на родительский узел.
- Свойства `previousSibling` и `nextSibling` дают доступ к левому и правому соседу.

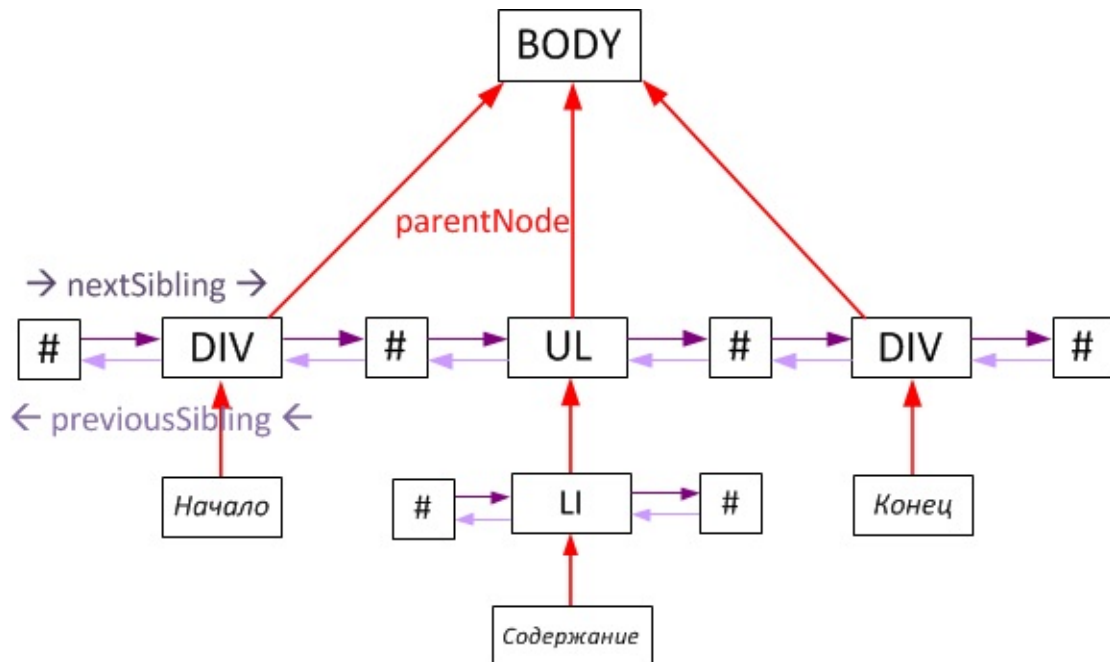
Ниже изображены ссылки между BODY и его потомками документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <div>Начало</div>

  <ul>
    <li>Содержание</li>
  </ul>

  <div>Конец</div>
</body>
</html>
```

Ссылки (пробельные узлы обозначены решеткой #):



Все навигационные ссылки — только для чтения. При изменениях DOM, добавлении или удалении элементов они обновляются автоматически.

## Таблицы

У таблиц есть дополнительные свойства для более удобной навигации по ним (выделены наиболее полезные):

### TABLE

- `table.rows` — список строк TR таблицы.
- `table.caption/tHead/tFoot` — ссылки на элементы таблицы CAPTION, THEAD, TFOOT.
- `table.tBodies` — список элементов таблицы TBODY, по спецификации их может быть несколько.

### THEAD/TFOOT/TBODY

- `tbody.rows` — список строк TR секции.

### TR

- `tr.cells` — список ячеек TD/TH
- `tr.sectionRowIndex` — номер строки в текущей секции THEAD/TBODY
- `tr.rowIndex` — номер строки в таблице

### TD/TH

- `td.cellIndex` — номер ячейки в строке

Пример использования:

```
<table>
  <tr> <td>один</td> <td>два</td> </tr>
  <tr> <td>три</td> <td>четыре</td> </tr>
</table>

<script>
  var table = document.body.children[0];
  alert( table.rows[0].cells[0].innerHTML ) // "один"
</script>
```

Спецификация: [HTMLTableElement](#) and [HTMLTableRowElement](#).

Эти свойства бывают удобны при работе с таблицами, т.к. делают код проще и короче.

В IE7 ряд «табличных» свойств не работает, если элемент вне документа.

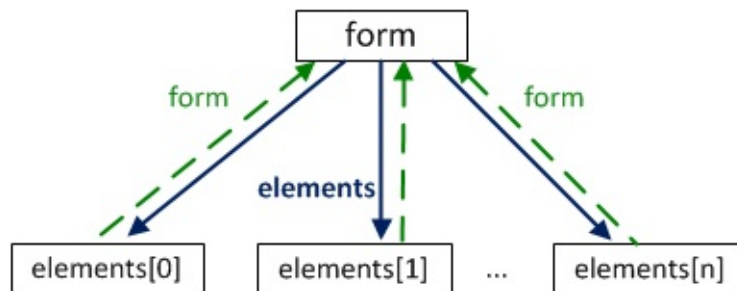
## Формы

Элементы FORM можно получить по имени или номеру, используя свойство `document.forms[name/index]`.

Например:

```
document.forms.my // форма с именем 'my'
document.forms[0] // первая форма в документе
```

Любой элемент формы `form` можно получить аналогичным образом, используя свойство `form.elements`.



Например:

```
<body>
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  var form = document.forms.my; // можно document.forms[0]
  var elem = form.elements.one; // можно form.elements[0]
  alert(elem.value); // 1
</script>

</body>
```

Может быть несколько элементов с одинаковым именем. В таком случае `form.elements[name]` вернет коллекцию элементов, например:

```
<body>
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
  var form = document.forms[0];
  var elems = form.elements.age;
  alert(elems[0].value); // 10, первый input
</script>

</body>
```

Эти ссылки не зависят от окружающих тегов. Элемент может быть «зарыт» где-то глубоко в форме, но он всё равно доступен через `form.elements`.

Спецификация: [HTMLFormElement](#).

`form.name` тоже работает, но (возможно) с ошибками.

## От элементов к форме

По элементу можно получить его форму, используя свойство `element.form`.

Пример:

```
<body>
<form>
  <input type="text" name="surname">
</form>

<script>
  var form = document.forms[0];
  var elem = form.elements.surname;
  alert(elem.form == form); // true
</script>

</body>
```

## Дополнительные ссылки для элементов (кроме IE<9)

Все современные браузеры, включая IE9+, поддерживают дополнительные ссылки:

- `childElementCount` — число детей-элементов (= `children.length` )
- `firstElementChild` — первый потомок-элемент (= `children[0]` )
- `lastElementChild` — последний потомок-элемент (= `children[children.length-1]` )
- `nextElementSibling` — правый брат-элемент
- `previousElementSibling` — левый брат-элемент

Любые другие узлы, кроме элементов, при этом просто игнорируются. Например:

## Итого

Сверху в DOM можно войти либо через `document.documentElement` (тег HTML), либо через `document.body` (тег BODY).

По элементу DOM можно получить всех соседей через ссылки:

- `childNodes` , `children` — список дочерних узлов.
- `firstChild` , `lastChild` — первый и последний потомки
- `parentNode` — родительский узел
- `previousSibling` , `nextSibling` — соседи влево-вправо

Все навигационные ссылки доступны только для чтения и поддерживаются автоматически.

Были рассмотрены ссылки для элементов, форм и таблиц.

# Свойства узлов: тип, тег, содержимое и другие

Источник: <http://learn.javascript.ru/basic-dom-node-properties>

В этой главе мы рассмотрим три основных свойства DOM-узлов: **тип**, **тег** и **содержимое**.

## Тип: `nodeType`

С тремя типами мы уже встречались. Это:

1. Элемент
2. Текстовый узел
3. Комментарий

На самом деле типов узлов гораздо больше. Строго говоря, их 12, и они описаны в спецификации [DOM Уровень 1](#):

```
interface Node {  
  // NodeType  
  const unsigned short ELEMENT_NODE      = 1;  
  const unsigned short ATTRIBUTE_NODE    = 2;  
  const unsigned short TEXT_NODE         = 3;  
  const unsigned short CDATA_SECTION_NODE = 4;  
  const unsigned short ENTITY_REFERENCE_NODE = 5;  
  const unsigned short ENTITY_NODE       = 6;  
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;  
  const unsigned short COMMENT_NODE      = 8;  
  const unsigned short DOCUMENT_NODE     = 9;  
  const unsigned short DOCUMENT_TYPE_NODE = 10;  
  const unsigned short DOCUMENT_FRAGMENT_NODE = 11;  
  const unsigned short NOTATION_NODE     = 12;  
  ...  
}
```

Нам важны номера основных типов.

**Самые важные** — это `ELEMENT_NODE` под номером 1 и `TEXT_NODE` под номером 3.

Тип узла содержится в его свойстве `nodeType`.

Например, выведем все узлы-потомки `document.body`, являющиеся элементами:

```
<body>  
  <div>Читатели:</div>  
  <ul>  
    <li>Вася</li>  
    <li>Петя</li>  
  </ul>  
  
  <!-- комментарий -->  
  
  <script>  
    var childNodes = document.body.childNodes;  
    for (var i=0; i<childNodes.length; i++) {  
      // отфильтровать не-элементы  
      if (childNodes[i].nodeType !== 1) continue;  
      alert(childNodes[i]);  
    }  
  </script>  
</body>
```

## Тег: `nodeName` и `tagName`

Существует целых два свойства: `nodeName` и `tagName`, которые содержат название(тег) элемента узла.

Название HTML-тега всегда находится в верхнем регистре.

Например, для `document.body` :

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

## Какая разница между tagName и nodeName ?

Разница отражена в названиях свойств, но неочевидна.

Свойство `nodeName` определено для многих типов DOM-узлов. Свойство `tagName` — есть только у элементов (в IE<9 также у комментариев, это ошибка в браузере).

Иначе говоря, при помощи `tagName` мы можем работать только с элементами, а `nodeName` может что-то сказать и о других типах узлов. При работе только с узлами элементов имеет смысл использовать `tagName` — так короче.

## innerHTML: содержимое элемента

Свойство `innerHTML` позволяет получить HTML-содержимое узла в виде строки. В `innerHTML` можно и читать и писать.

Пример выведет на экран все содержимое `document.body` , а затем заменит его на другое:

```
<body>
  <p>Параграф</p>
  <div>Div</div>

  <script>
    alert(document.body.innerHTML); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
    alert(document.body.innerHTML); // 'Новый BODY!'
  </script>

</body>
```

`innerHTML` — очень полезное свойство и одно из самых часто используемых.

## Тонкости innerHTML

1. Для таблиц в IE9- — `innerHTML` только для чтения;
2. Добавление `innerHTML+=` осуществляет перезапись;
3. Если в `innerHTML` есть тег `script` — он не будет выполнен;
4. IE<9 обрезает `style` и `script` в начале `innerHTML` .

## outerHTML: HTML узла целиком

Свойство `outerHTML` содержит HTML узла целиком.

```
<div id="hi">Привет <b>Мир</b></div>

<script>
  var div = document.getElementById('hi');
  alert(div.outerHTML); // <div>Привет <b>Мир</b></div>
  alert(div.innerHTML); // Привет <b>Мир</b>
</script>
```

## nodeValue/data: содержимое текстового узла



**Свойство `innerHTML` есть только у узлов-элементов.**

Содержимое других узлов, например, текстовых или комментариев, доступно через два свойства: `nodeValue` и `data`.

Его тоже можно читать и обновлять. Следующий пример демонстрирует это:

```
<body>
  Привет
  <!-- Комментарий -->
  <script>
    for (var i=0; i<document.body.childNodes.length; i++) {
      alert(document.body.childNodes[i].nodeValue);
      alert(document.body.childNodes[i].data);
    }

    document.body.firstChild.data = "Здравствуйте!";
  </script>
</body>
```

В этом примере выводятся последовательно:

1. Содержимое первого узла (текстового): Привет (2 раза).
2. Содержимое второго узла (комментария): Комментарий (2 раза).
3. Содержимого третьего узла (текста между комментарием и скриптом): (там пробелы, 2 раза)
4. Свойство `nodeValue = null` для узла SCRIPT, так как это узел-элемент. А вот `data = undefined`. Это единственное различие в поведении этих свойств.
5. Наконец, последний вызов заменит содержимое `document.body.firstChild`, и это тут же отразится в документе.

Кстати, после SCRIPT есть еще один текстовый узел, но на момент работы скрипта браузер дошел в разборе документа только до SCRIPT, поэтому он не будет выведен.

## Другие свойства

У DOM-узлов есть свойства, зависящие от типа, например:

- `value` — значение для INPUT, SELECT или TEXTAREA
- `id` — идентификатор
- `href` — адрес ссылки
- ...многие другие...

Например:

```
<input type="text" id="my-input" value="значение">

<script>
  var input = document.body.children[0];

  alert(input.type); // "text"
  alert(input.id); // "my-input"
  alert(input.value); // значение
</script>
```

Полный список свойств можно получить из спецификации. В частности, для `input[type="text"]` она находится <http://www.w3.org/TR/html-markup/input.text.html>. Как правило, основные атрибуты элемента дают свойство с тем же названием.

## Итого

Основные свойства DOM-узлов:

- **nodeType**

Тип узла. Самые популярные типы: "1" - для элементов и "3" - для текстовых узлов. Только для чтения.

- **nodeName/tagName**

Название тега заглавными буквами. `nodeName` имеет специальные значения для узлов-неэлементов. Только для чтения.

- **innerHTML**

Внутреннее содержимое узла-элемента в виде HTML. Можно изменять.

- **outerHTML**

Полный HTML узла-элемента. При записи в `elem.outerHTML` переменная `elem` сохраняет старый узел.

- **nodeValue/data**

Содержимое текстового узла или комментария. Свойство `nodeValue` также определено и для других типов узлов. Можно изменять.

Узлы DOM также имеют другие свойства, в зависимости от тега. Например, у INPUT есть свойства `value` и `checked`, а у A есть `href` и т.д.

# Добавление и удаление узлов

Источник: <http://learn.javascript.ru/modifying-document>

Изменение DOM — ключ к созданию «живых» страниц.

В этой главе мы рассмотрим, как создавать новые элементы «на лету» и заполнять их данными.

## Создание элементов: createElement

Для создания элементов используются следующие методы документа:

- `document.createElement(tag)` . Создает новый элемент с указанным тегом:

```
var div = document.createElement('div');
```

- `document.createTextNode(text)` . Создает новый текстовый узел с данным текстом:

```
var textElem = document.createTextNode('Тут был я');
```

Новому элементу тут же можно поставить свойства:

```
var newDiv = document.createElement('div');
newDiv.className = 'myclass';
newDiv.id = 'myid';
newDiv.innerHTML = 'Привет, мир!';
```

## Клонирование элемента

Новый элемент можно также клонировать из существующего:

```
newElem = elem.cloneNode(true)
```

Клонировует элемент `elem`, вместе с атрибутами, включая вложенные в него.

```
newElem = elem.cloneNode(false)
```

Клонировует элемент `elem`, вместе с атрибутами, но без подэлементов.

## Добавление элемента: appendChild, insertBefore

Чтобы DOM-узел был показан на странице, его необходимо вставить в документ.

Для этого у любого элемента есть метод `appendChild` :

- `parentElem.appendChild(elem)`

Добавляет `elem` в список дочерних элементов `parentElem` . Новый узел добавляется в конец списка. Следующий пример добавляет новый элемент в уже существующий `div`:

```
<div>
  ...
```

```
</div>
<script>
  var parentElem = document.body.children[0];

  var newDiv = document.createElement('div');
  newDiv.innerHTML = 'Привет, мир!';

  parentElem.appendChild(newDiv);
</script>
```

- **parentElem.insertBefore(elem, nextSibling)**

Вставляет `elem` в список дочерних `parentElem`, перед элементом `nextSibling`. Сделать новый `div` первым дочерним можно так:

```
<div>
  ...
</div>
<script>

  var parentElem = document.body.children[0];

  var newDiv = document.createElement('div');
  newDiv.innerHTML = 'Привет, мир!';

  parentElem.insertBefore(newDiv, parentElem.firstChild);
</script>
```

Вместо `nextSibling` может быть `null`, тогда `insertBefore` работает как `appendChild`.

```
parentElem.insertBefore(elem, null);
// то же, что и:
parentElem.appendChild(elem)
```

Все методы вставки возвращают вставленный узел, например `parent.appendChild(elem)` возвращает `elem`.

## Удаление узлов: `removeChild`

Для удаления узла есть два метода:

- **parentElem.removeChild(elem)**

Удаляет `elem` из списка детей `parentElem`.

- **parentElem.replaceChild(elem, currentElem)**

Среди детей `parentElem` заменяет `currentElem` на `elem`. Оба этих метода возвращают удаленный узел. Они просто вынимают его из списка, никто не мешает вставить его обратно в DOM в будущем.

Если вы хотите переместить элемент на новое место — не нужно его удалять со старого.

**Все методы вставки автоматически удаляют вставляемый элемент со старого места.**

Конечно же, это очень удобно.

Например, поменяем элементы местами:

```
<div>Первый</div>
<div>Второй</div>
<script>
  var first = document.body.children[0];
  var last = document.body.children[1];
```

```
// нет необходимости в предварительном removeChild(last)
document.body.insertBefore(last, first); // поменять местами
</script>
```

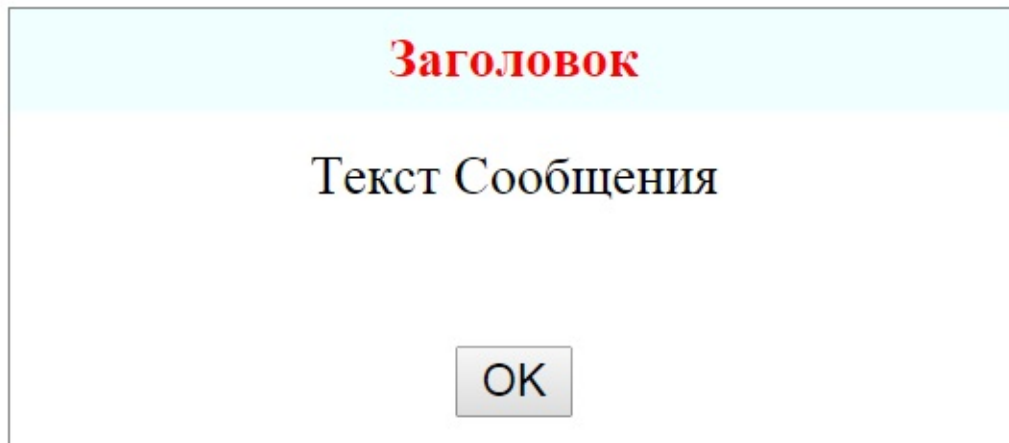
## Пример: показ сообщения

В качестве реального примера рассмотрим добавление сообщения на страницу. Чтобы показывалось посередине экрана и было красивее, чем обычный `alert`.

HTML-код для сообщения (без JS):

```
<style>
.message {
  width: 300px;
  height: 130px;
  border: 1px solid gray;
  text-align: center;
}
.message h1 {
  color: red;
  background: azure;
  font-size: 16px;
  height: 30px;
  line-height: 30px;
  margin: 0;
}
.message .content {
  height: 50px;
  padding: 10px;
}
</style>

<div class="message">
  <h1>Заголовок</h1>
  <div class="content">Текст Сообщения</div>
  <input class="ok" type="button" value="OK"/>
</div>
```



Как видно - сообщение вложено в `div` фиксированного размера `message` и состоит из заголовка `h1`, тела `content` и кнопки `OK`, которая нужна, чтобы сообщение закрыть.

Кроме того, добавлено немного стилей, чтобы как-то смотрелось.

## Создание сообщения

Для создания сложных структур DOM, как правило, используют либо готовый «шаблонный узел» и метод `cloneNode`, либо свойство `innerHTML`.

Следующая функция создает сообщение с указанным телом и заголовком.

```
function createMessage(title, body) {
  // (1)
  var container = document.createElement('div');

  // (2)
  container.innerHTML = '<div class="message"> \
    <h1>' + title + '</h1> \
    <div class="content">' + body + '</div> \
    <input class="ok" type="button" value="OK"> \
  </div>';

  // (3)
  return container.firstChild;
}
```

Как видно, она поступает довольно хитро. Чтобы создать элемент по текстовому шаблону, она сначала создает временный элемент `container` (1), а потом записывает (2) сообщение как `innerHTML` временного элемента. Теперь готовый элемент можно получить как `container.firstChild` и вернуть в (3).

## Добавление

Полученный элемент можно добавить в DOM:

```
var messageElem = createMessage('Привет, Мир!', 'Я - элемент DOM!')

document.body.appendChild(messageElem);
```

Окончательный результат:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<link type="text/css" rel="stylesheet" href="alert.css" />
</head>
<body>

<script>

function createMessage(title, body) {
  var container = document.createElement('div');

  container.innerHTML = '<div class="message"> \
    <h1>' + title + '</h1> \
    <div class="content">' + body + '</div> \
    <input class="ok" type="button" value="OK"> \
  </div>';

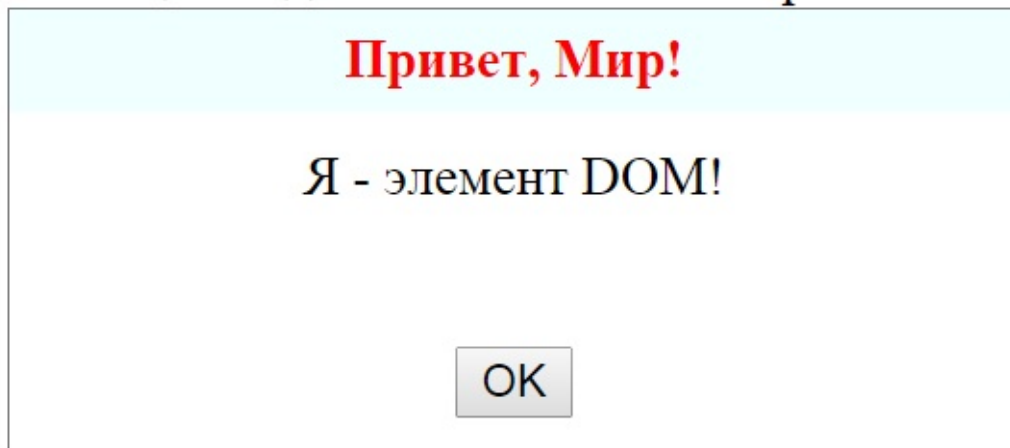
  return container.firstChild;
}

var messageElem = createMessage('Привет, Мир!', 'Я - элемент DOM!')

document.body.appendChild(messageElem);

</script>

</body>
</html>
```



## Текстовые узлы, тонкости использования

Как правило, при создании узлов и заполнении их используется `innerHTML`. Но текстовые узлы тоже имеют интересную область применения.

У них есть две особенности.

Допустим, у нас есть пустой узел DOM `elem`.

Одинаковый ли результат дадут эти скрипты?

Первый:

```
elem.appendChild(document.createTextNode(text));
```

Второй:

```
elem.innerHTML = text;
```

- `createTextNode` создает текст `'<b>текст</b>'`:

```
<div></div>
<script>
  var text = '<b>текст</b>';

  var elem = document.body.children[0];
  elem.appendChild(document.createTextNode(text));
</script>
```

- `innerHTML` присваивает HTML `<b>текст</b>`:

```
<div></div>
<script>
  var text = '<b>текст</b>';

  var elem = document.body.children[0];
  elem.innerHTML = text;
</script>
```

Итак, отличий два:

1. При создании текстового узла `createTextNode('<b>...</b>')` любые специальные символы и теги в строке будут интерпретированы как текст. А `innerHTML` вставит их как HTML.

2. Во всех современных браузерах (кроме IE<8) создание и вставка текстового узла работает гораздо быстрее, чем присвоение HTML.

## Итого

### Методы для создания узлов:

- `document.createElement(tag)` — создает элемент;
- `document.createTextNode(value)` — создает текстовый узел;
- `elem.cloneNode(deep)` — клонирует элемент, если `deep == true`, то со всеми потомками.

### Вставка и удаление узлов:

- `parent.appendChild(elem)` ;
- `parent.insertBefore(elem, nextSibling)` ;
- `parent.removeChild(elem)` ;
- `parent.replaceChild(elem, currentElem)` .

Все эти методы возвращают `elem` .

Запомнить порядок аргументов очень просто: новый(вставляемый) элемент — всегда первый.

Методы для изменения DOM также описаны в спецификации [DOM Level 1](#).



# Задачи

## Задача 1

Что выведет этот код?

```
<script>
var body = document.body;

body.innerHTML = "<!--" + body.tagName + "-->";

alert(body.firstChild.data); // что выведет?
</script>
```

## Решение

Ответ: BODY.

```
<script>
var body = document.body;

body.innerHTML = "<!--" + body.tagName + "-->";

alert(body.firstChild.data); // BODY
</script>
```

Происходящее по шагам:

1. Заменяем содержимое на комментарий. Он будет иметь вид `<!--BODY-->`, так как `body.tagName == "BODY"`. Как мы помним, свойство `tagName` в HTML всегда находится в верхнем регистре.
2. Этот комментарий теперь является первым и единственным потомком `body.firstChild`.
3. Получим значение `data` для комментария `body.firstChild`. Оно равно содержимому узла для всех узлов, кроме элементов. Содержимое комментария: "BODY".

## Задача 2

Напишите функцию, которая удаляет элемент из DOM.

Синтаксис должен быть таким: `remove(elem)`, то есть, в отличие от `parentNode.removeChild(elem)` — без родительского элемента.

```
<div>Это</div>
<div>Все</div>
<div>Элементы DOM</div>

<script>
var elem = document.body.children[0];

function remove(elem) { /* ваш код */ }
remove(elem); // <-- функция должна удалить элемент
</script>
```

## Решение

Родителя `parentNode` можно получить из `elem`.

Нужно учесть два момента.

1. Родителя может не быть (элемент уже удален или еще не вставлен).
2. Для совместимости со стандартным методом нужно вернуть удаленный элемент. Вот так выглядит решение:

```
function remove(elem) {  
  return elem.parentNode ? elem.parentNode.removeChild(elem) : elem;  
}
```

## Задача 3

Напишите функцию `insertAfter(elem, refElem)`, которая добавит `elem` после узла `refElem`.

```
<div>Это</div>  
<div>Элементы</div>  
  
<script>  
  var elem = document.createElement('div');  
  elem.innerHTML = '<b>Новый элемент</b>';  
  
  function insertAfter(elem, refElem) { /* ваш код */ }  
  
  var body = document.body;  
  
  // вставить elem после первого элемента  
  insertAfter(elem, body.firstChild); // <--- должно работать  
  
  // вставить elem за последним элементом  
  insertAfter(elem, body.lastChild); // <--- должно работать  
  
</script>
```

## Решение

Для того, чтобы добавить элемент после `refElem`, мы можем вставить его перед `refElem.nextSibling`.

Но что если `nextSibling` нет? Это означает, что `refElem` является последним потомком своего родителя и можем использовать `appendChild`.

Код:

```
function insertAfter(elem, refElem) {  
  var parent = refElem.parentNode;  
  var next = refElem.nextSibling;  
  if (next) {  
    return parent.insertBefore(elem, next);  
  } else {  
    return parent.appendChild(elem);  
  }  
}
```

Но код может быть гораздо короче, если использовать фишку со вторым аргументом `null` метода `insertBefore`:

```
function insertAfter(elem, refElem) {  
  return refElem.parentNode.insertBefore(elem, refElem.nextSibling);  
}
```

Если нет `nextSibling`, то второй аргумент `insertBefore` становится `null` и тогда `insertBefore(elem, null)` работает как `appendChild`.

В решении нет проверки на существование `refElem.parentNode`, поскольку вставка после элемента без родителя — уже ошибка, пусть она возникнет в функции, это нормально.

## Задача 4

Напишите функцию `removeChildren`, которая удаляет всех потомков элемента.

```
<table>
  <tr>
    <td>Это</td><td>Все</td><td>Элементы DOM</td>
  </tr>
</table>

<ol>
  <li>Вася</li>
  <li>Петя</li>
  <li>Маша</li>
  <li>Даша</li>
</ol>

<script>
  function removeChildren(elem) { /* ваш код */ }

  removeChildren(document.body.children[0]); // очищает таблицу
  removeChildren(document.body.children[1]); // очищает список
</script>
```

P.S. Проверьте ваше решение в IE8.

## Решение

### 1) Неправильное решение:

Для начала рассмотрим забавный пример того, как делать не надо:

```
function removeChildren(elem) {
  for(var k=0; k<elem.childNodes.length;k++) {
    elem.removeChild(elem.childNodes[k]);
  }
}
```

Если вы попробуете это на практике, то увидите, то это не работает.

Не работает потому, что `childNodes` всегда начинается 0 и автоматически смещается, когда первый потомок удален(т.е. тот, что был вторым, станет первым), поэтому такой цикл по `k` пропустит половину узлов.

### 2) Решение через DOM:

Правильное решение:

```
function removeChildren(elem) {
  while(elem.lastChild) {
    elem.removeChild(elem.lastChild);
  }
}
```

### 3) Неправильное решение (innerHTML):

Прямая попытка использовать `innerHTML` была бы неправильной:

```
function removeChildren(elem) {
  elem.innerHTML = '';
```

```
}
```

Дело в том, что в IE<9 свойство `innerHTML` на большинстве табличных элементов (кроме ячеек TH/TD) не работает. Будет ошибка.

#### 4) Верное решение (innerHTML):

Можно завернуть `innerHTML` в `try/catch`:

```
function removeChildren(elem) {  
  try {  
    elem.innerHTML = '';  
  } catch(e) {  
    while(elem.firstChild) {  
      elem.removeChild(elem.firstChild);  
    }  
  }  
}
```

## Задача 5

Напишите интерфейс для создания списка.

Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью `prompt`. Создавайте пункт и добавляйте его к UL.
2. Процесс прерывается, когда пользователь нажимает ESC.
3. Все элементы должны создаваться динамически.

Пример тут: <http://ru.lookatcode.com/files/tutorial/browser/dom/createList.html>

Если посетитель вводит теги — в списке они показываются как обычный текст.

P.S. `prompt` возвращает null, если пользователь нажал ESC.

## Решение

```
<!DOCTYPE HTML>  
<html>  
<body>  
<h1>Создание списка</h1>  
  
<script>  
  var ul = document.createElement('ul');  
  document.body.appendChild(ul);  
  
  while (true) {  
    var data = prompt("Введите текст для пункта списка", "");  
  
    if (data === null) {  
      break;  
    }  
  
    var li = document.createElement('li');  
    li.appendChild(document.createTextNode(data));  
    ul.appendChild(li);  
  }  
</script>  
  
</body>  
</html>
```

Делайте проверку на null в цикле. `prompt` возвращает это значение только если был нажат ESC.

Контент в LI добавляйте с помощью `document.createTextNode`, чтобы правильно работали `<`, `>` и т.д.

## Задача 6

Напишите функцию, которая создаёт вложенный список UL/LI (дерево) из объекта.

Например:

```
var data = {
  "Рыбы": {
    "Форель": {},
    "Щука": {}
  },
  "Деревья": {
    "Хвойные": {
      "Лиственница": {},
      "Ель": {}
    },
    "Цветковые": {
      "Берёза": {},
      "Тополь": {}
    }
  }
};
```

Синтаксис:

```
var container = document.getElementById('container');
createTree(container, data); // создаёт
```

Результат (дерево):

- Рыбы
  - Форель
  - Щука
- Деревья
  - Хвойные
    - Лиственница
    - Ель
  - Цветковые
    - Берёза
    - Тополь

Выберите один из двух способов решения этой задачи:

1. Создать строку, а затем присвоить через `container.innerHTML`.
2. Создавать узлы через методы DOM.

Если получится – сделайте оба.

Исходный код: <http://learn.javascript.ru/play/tutorial/browser/dom/build-tree-src.html>

## Решение

Решения через рекурсию.

1) <http://learn.javascript.ru/play/tutorial/browser/dom/build-tree.html>.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>

<div id="container"></div>

<script>
  var data = {
    "Рыбы":{
      "Форель":{},
      "Щука":{}
    },

    "Деревья":{
      "Хвойные":{
        "Лиственница":{},
        "Ель":{}
      },
      "Цветковые":{
        "Берёза":{},
        "Тополь":{}
      }
    }
  };

  function createTree(container, obj) {
    container.innerHTML = createTreeText(obj);
  }

  function createTreeText(obj) { // отдельная рекурсивная функция
    var li = '';
    for (var key in obj) {
      li += '<li>' + key + createTreeText(obj[key]) + '</li>';
    }
    if (li) {
      var ul = '<ul>' + li + '</ul>'
    }
    return ul || '';
  }

  var container = document.getElementById('container');
  createTree(container, data);
</script>
</body>
</html>
```

2) <http://learn.javascript.ru/play/tutorial/browser/dom/build-tree-dom.html>.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>

<div id="container"></div>

<script>
  var data = {
    "Рыбы":{
      "Форель":{},
      "Щука":{}
    },

    "Деревья":{
      "Хвойные":{
        "Лиственница":{},
        "Ель":{}
      }
    }
  };

  function createTree(container, obj) {
    container.innerHTML = createTreeText(obj);
  }

  function createTreeText(obj) { // отдельная рекурсивная функция
    var li = '';
    for (var key in obj) {
      li += '<li>' + key + createTreeText(obj[key]) + '</li>';
    }
    if (li) {
      var ul = '<ul>' + li + '</ul>'
    }
    return ul || '';
  }

  var container = document.getElementById('container');
  createTree(container, data);
</script>
</body>
</html>
```

```

    },
    "Цветковые":{
      "Берёза":{},
      "Тополь":{}
    }
  }
};

function createTree(container, obj) {
  container.appendChild( createTreeDom(obj) );
}

function createTreeDom(obj) {
  // если нет детей, то рекурсивный вызов ничего не возвращает
  // так что вложенный UL не будет создан
  if (isObjectEmpty(obj)) return;

  var ul = document.createElement('ul');

  for (var key in obj) {
    var li = document.createElement('li');
    li.innerHTML = key;

    var childrenUl = createTreeDom(obj[key]);
    if (childrenUl) li.appendChild(childrenUl);

    ul.appendChild(li);
  }

  return ul;
}

function isObjectEmpty(obj) {
  for (var key in obj) {
    return false;
  }
  return true;
}

var container = document.getElementById('container');
createTree(container, data);
</script>

</body>
</html>

```

## Задача 7

Напишите функцию, которая умеет генерировать календарь для заданной пары (месяц, год).

Календарь должен быть таблицей, где каждый день — это TD. У таблицы должен быть заголовок с названиями дней недели, каждый день — TH.

Синтаксис: `createCalendar(id, year, month)` .

Такой вызов должен генерировать текст для календаря месяца `month` в году `year` , а затем помещать его внутрь элемента с указанным `id` .

Например: `createCalendar("cal", 2012, 9)` сгенерирует в `<div id='cal'></div>` следующий календарь:

ПН	ВТ	СР	ЧТ	ПТ	СБ	ВС
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Начальный документ со стилями [http://learn.javascript.ru/play/tutorial/date/calendar\\_src.html](http://learn.javascript.ru/play/tutorial/date/calendar_src.html)

Или:

```
<!DOCTYPE HTML>
<html>
<head>
<style>
table {
  border-collapse: collapse;
}

td, th {
  border: 1px solid black;
  padding: 3px;
  text-align: center;
}

th {
  font-weight: bold;
  background-color: #E6E6E6;
}
</style>
<meta charset="utf-8">
</head>
<body>

<div id="calendar"></div>

<script>

function createCalendar(id, year, month) {
  var elem = document.getElementById(id)

  // ... ваш код, который генерирует в elem календарь
}

createCalendar('calendar', 2011, 1)

</script>
</body>
</html>
```

P.S. Достаточно сгенерировать календарь, кликабельным его делать не нужно.

## Решение

Для решения задачи сгенерируем таблицу в виде строки: `"<table>...</table>"`, а затем присвоим в `innerHTML`.

Алгоритм:

1. Создать объект даты `d = new Date(year, month-1)`. Это первый день месяца `month` (с учетом того, что месяцы в JS начинаются от 0, а не от 1).
2. Ячейки первого ряда пустые от начала и до дня недели `d.getDay()`, с которого начинается месяц. Создадим их.
3. Увеличиваем день в `d` на единицу: `d.setDate(d.getDate()+1)`, и добавляем в календарь очередную ячейку, пока не достигли следующего месяца. При этом последний день недели означает вставку перевода строки `"</tr><tr>"`.
4. При необходимости, если календарь окончился не на воскресенье – добавить пустые TD в таблицу, чтобы было все ровно.

Код решения:

```
<!DOCTYPE HTML>
<html>
```



```

<head>
<style>
table {
    border-collapse: collapse;
}

td, th {
    border: 1px solid black;
    padding: 3px;
    text-align: center;
}

th {
    font-weight: bold;
    background-color: #E6E6E6;
}
</style>
<meta charset="utf-8">
</head>
<body>

<div id="calendar"></div>

<script>
function createCalendar(id, year, month) {
    var elem = document.getElementById(id);

    var mon = month - 1; // месяцы в JS идут от 0 до 11, а не от 1 до 12
    var d = new Date(year, mon);

    var table = '<table><tr><th>пн</th><th>вт</th><th>ср</th><th>чт</th><th>пт</th><th>сб</th><th>вс</th></tr><tr>';

    // заполнить первый ряд от понедельника
    // и до дня, с которого начинается месяц
    // * * * | 1 2 3 4
    for (var i = 0; i < getDay(d); i++) {
        table += '<td></td>';
    }

    // ячейки календаря с датами
    while(d.getMonth() == mon) {
        table += '<td>'+d.getDate()+'</td>';

        if (getDay(d) % 7 == 6) { // вс, последний день - перевод строки
            table += '</tr><tr>';
        }

        d.setDate(d.getDate()+1);
    }

    // добить таблицу пустыми ячейками, если нужно
    if (getDay(d) != 0) {
        for (var i = getDay(d); i < 7; i++) {
            table += '<td></td>';
        }
    }

    // закрыть таблицу
    table += '</tr></table>';

    // только одно присваивание innerHTML
    elem.innerHTML = table;
}

function getDay(date) { // получить номер дня недели, от 0(пн) до 6(вс)
    var day = date.getDay();
    if (day == 0) day = 7;
    return day - 1;
}

createCalendar("calendar", 2012, 9)
</script>

</body>
</html>

```

# DOM events

---

Прочесть:

<http://learn.javascript.ru/introduction-browser-events> READ

<http://learn.javascript.ru/bubbling-and-capturing> READ

<http://learn.javascript.ru/default-browser-action> READ

<http://learn.javascript.ru/mouse-event> READ

# Введение в браузерные события

Источник: <http://learn.javascript.ru/introduction-browser-events>

Для реакции на действия посетителя и внутреннего взаимодействия скриптов существуют **события**.

**Событие** — это сигнал от браузера о том, что что-то произошло.

Существует много видов событий.

- DOM-события, которые инициализируются элементами DOM. Например:
  - Событие `click` происходит, когда кликнули на элемент;
  - Событие `mouseover` — когда на элемент наводится мышь;
  - Событие `focus` — когда посетитель фокусируется на элементе;
  - Событие `keydown` — когда посетитель нажимает клавишу.
- События для окна браузера. Например, `resize` — когда изменяется размер окна.
- Есть загрузки файла/документа: `load`, `readystatechange`, `DOMContentLoaded` ...

**События соединяют JavaScript-код с документом и посетителем, позволяя создавать динамические интерфейсы.**

## Назначение обработчиков событий

Есть несколько способов назначить событию обработчик. Сейчас мы их рассмотрим, начиная от самого простого.

## Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется `on<событие>`.

Например, чтобы прикрепить `click`-событие к `input`-кнопке, можно присвоить обработчик `onclick`, вот так:

```
<input id="b1" value="Нажми меня" onclick="alert('Спасибо!')" type="button"/>
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте `onclick`.

В действии:

Нажми меня

Обратите внимание, внутри `alert` используются одиночные кавычки, так как сам атрибут находится в двойных.

Запись вида `onclick="alert("клик")"` не будет работать. Если вам действительно нужно использовать именно двойные кавычки, то это можно сделать, заменив их на `&quot;`: `onclick="alert(&quot;Клик&quot;)"`.

Однако, обычно этого не требуется, так как в разметке пишутся только очень простые обработчики. Если нужно сделать что-то сложное, то имеет смысл описать это в функции, и в обработчике вызвать её.

Следующий пример по клику запускает функцию `countRabbits()`.

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">

  <script>
    function countRabbits() {
      for(var i = 1; i <= 3; i++) {
```

```

        alert("Кролик номер " + i);
    }
}
</script>
</head>
<body>
    <input type="button" onclick="countRabbits()" value="Считать кроликов!"/>
</body>
</html>

```

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому `onclick` будет работать так же, как `onClick` или `onclick ...`. Но, как правило, атрибуты пишут в нижнем регистре: `onclick`.

## Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента `on<событие>`.

Пример установки обработчика `click` элементу с `id="myElement"`:

```

<input id="myElement" type="button" value="Нажми меня"/>
<script>
var elem = document.getElementById('myElement');

elem.onclick = function() {
    alert('Спасибо');
}
</script>

```

В действии:

Нажми меня

Если обработчик задан через атрибут, то соответствующее свойство появится у элемента автоматически. Браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство `onclick`.

**Первичным является именно свойство, а атрибут — лишь способ его инициализации.**

Эти два примера кода работают одинаково:

- **Только HTML:**

```

<input type="button" onclick="alert('Клик!')" value="Кнопка"/>

```

- **HTML + JS:**

```

<input type="button" id="button" value="Кнопка"/>
<script>
    document.getElementById('button').onclick = function() {
        alert('Клик!');
    }
</script>

```

**Так как свойство, в итоге, одно, то назначить по обработчику и там и там нельзя.**

В примере ниже, назначение через JavaScript перезапишет обработчик из атрибута:

```

<input type="button" onclick="alert('до')" value="Нажми меня"/>

<script>
    var elem = document.getElementsByTagName('input')[0];

```

```
elem.onclick = function() { // перезапишет существующий обработчик
    alert('После');
}
</script>
```

Обработчиком можно назначить уже существующую функцию:

```
function sayThanks() {
    alert('Спасибо!');
}

document.getElementById('button').onclick = sayThanks;
```

## Частые ошибки

- Функция должна быть присвоена как `sayThanks`, а не `sayThanks()` :

```
document.getElementById('button').onclick = sayThanks;
```

Если добавить скобки, то `sayThanks()` — будет уже результат выполнения функции (а так как в ней нет `return`, то в `onclick` попадёт `undefined`). Нам же нужна именно функция.

А вот в разметке как раз скобки нужны:

```
<input type="button" id="button" onclick="sayThanks()"/>
```

Это различие просто объяснить. При создании обработчика браузером по разметке, он автоматически создает функцию из его содержимого. Поэтому последний пример — фактически то же самое, что:

```
document.getElementById('button').onclick = function() {
    sayThanks(); // содержимое атрибута
}
```

- Используйте свойство, а не атрибут. Так неверно: `elem.setAttribute('onclick', func)` .

Хотя, с другой стороны, если `func` — строка, то такое присвоение будет успешным, например:

```
// работает, будет при клике выдавать 1
document.body.setAttribute('onclick', 'alert(1)');
```

Браузер в этом случае сделает функцию-обработчик с телом из строки `alert(1)` .

...А вот если `func` — не строка, а функция (как и должно быть), то работать совсем не будет:

```
// при нажатии на body будут ошибки
document.body.setAttribute('onclick', function() { alert(1) });
```

Значением атрибута может быть только строка. Любое другое значение преобразуется в строку. Функция в строчном виде обычно даёт свой код: `"function() { alert(1) }"` .

- Используйте функции, а не строки.

Запись `elem.onclick = 'alert(1)'` будет работать, но не рекомендуется.

При использовании в такой функции-строке переменных из замыкания будут проблемы с JavaScript-минификаторами. Здесь мы не будем вдаваться в детали этих проблем, но общий принцип такой — функция должна быть `function`.

- Названия свойств регистрозависимы, поэтому `on<событие>` должно быть написано в нижнем регистре.

Свойство `ONCLICK` работать не будет.

## Доступ к элементу, `this`

Внутри обработчика события `this` ссылается на текущий элемент. Это можно использовать, чтобы получить свойства или изменить элемент.

В коде ниже `button` выводит свое содержимое, используя `this.innerHTML`:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

В действии:

Нажми меня

## Недостатки назначения через `on`событие

Фундаментальный недостаток описанных способов назначения обработчика — невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать ее подсвеченной, а другая — выдавать сообщение. Нужно в разных местах два обработчика повесить.

При этом новый обработчик будет затирать предыдущий. Например, следующий код на самом деле назначает один обработчик — последний:

```
input.onclick = function() { alert(1); }  
// ...  
input.onclick = function() { alert(2); } // заменит предыдущий обработчик
```

Конечно, это можно обойти разными способами, в том числе написанием фреймворка вокруг обработчиков. Но существует и другой метод назначения обработчиков, который свободен от указанного недостатка.

## Специальные методы

Для назначения обработчиков существуют специальные методы. Как правило, в браузерах они стандартные, кроме IE<9, где они похожи, но немного другие.

## Методы IE<9

Сначала посмотрим метод для старых IE, т.к. оно чуть проще.

Назначение обработчика осуществляется вызовом `attachEvent`:

```
element.attachEvent( "on"+event, handler);
```

Удаление обработчика — вызовом `detachEvent`:

```
element.detachEvent( "on"+event, handler);
```

Например:

```
var input = document.getElementById('button')
function handler() {
    alert('спасибо!')
}
input.attachEvent( "onclick" , handler) // Назначение обработчика
// ....
input.detachEvent( "onclick", handler) // Удаление обработчика
```

Обычно, обработчики ставятся. Но бывают ситуации, когда их нужно удалять или менять.

В этом случае нужно передать в метод удаления именно функцию-обработчик. Такой вызов будет неправильным:

```
input.attachEvent( "onclick" ,
    function() {alert('Спасибо!')}
)
// ....
input.detachEvent( "onclick",
    function() {alert('Спасибо!')}
)
```

Несмотря на то, что функции работают одинаково, это две разных функции. Использование `attachEvent` позволяет добавлять несколько обработчиков на одно событие одного элемента.

Пример ниже будет работать только в IE и Opera:

```
<input id="myElement" type="button" value="Нажми меня"/>

<script>
    var myElement = document.getElementById("myElement")
    var handler = function() {
        alert('Спасибо!')
    }

    var handler2 = function() {
        alert('Спасибо еще раз!')
    }

    myElement.attachEvent("onclick", handler); // первый
    myElement.attachEvent("onclick", handler2); // второй
</script>
```

**У обработчиков, назначенных с `attachEvent`, нет `this`.** Это важная особенность и подводный камень старых IE.

## Назначение обработчиков по стандарту

Официальный способ назначения обработчиков из стандарта W3C работает во всех современных браузерах, включая IE9+.

Назначение обработчика:

```
element.addEventListener(event, handler, phase);
```

Удаление:

```
element.removeEventListener(event, handler, phase);
```

Как видите, похоже на `attachEvent/detachEvent`, только название события пишется без префикса «on».

Еще одно отличие от синтаксиса Microsoft — это третий параметр: `phase`, который обычно не используется и выставлен в `false`. Позже мы посмотрим, что он означает.

Использование этого метода — такое же, как и у `attachEvent`:

```
function handler() { ... }

elem.addEventListener( "click", handler, false) // назначение обработчика

elem.removeEventListener( "click", handler, false) // удаление обработчика
```

## Особенности специальных методов

- Можно поставить столько обработчиков, сколько вам нужно.
- Нельзя получить все назначенные обработчики из элемента.
- Браузер не гарантирует сохранение порядка выполнения обработчиков. Они могут быть назначены в одном порядке, а выполняться — в другом.
- Кроссбраузерные несовместимости.

## Кроссбраузерный способ назначения обработчиков

Можно объединить способы для IE<9 и современных браузеров, создав свои методы `addEvent(elem, type, handler)` и `removeEvent(elem, type, handler)`:

```
var addEvent, removeEvent;

if (document.addEventListener) { // проверка существования метода
  addEvent = function(elem, type, handler) {
    elem.addEventListener(type, handler, false);
  };
  removeEvent = function(elem, type, handler) {
    elem.removeEventListener(type, handler, false);
  };
} else {
  addEvent = function(elem, type, handler) {
    elem.attachEvent("on" + type, handler);
  };
  removeEvent = function(elem, type, handler) {
    elem.detachEvent("on" + type, handler);
  };
}

...
// использование:
addEvent(elem, "click", function() { alert("Привет"); });
```

Это хорошо работает в большинстве случаев, но у обработчика не будет `this` в IE, потому что `attachEvent` не поддерживает `this`.

Кроме того, в IE<8 есть проблемы с утечками памяти... Но если вам не нужно `this`, и вы не боитесь утечек (как вариант — не поддерживаете IE<8), то это решение может подойти.

## Итого

Есть три способа назначения обработчиков событий:

1. Атрибут HTML: `onclick="..."`.



2. СВОЙСТВО: `elem.onclick = function` .

3. Специальные методы:

- Для IE<9: `elem.attachEvent(он+событие, handler)` (удаление через `detachEvent` ).
- Для остальных: `elem.addEventListener(событие, handler, false)` (удаление через `removeEventListener` ).

Все способы, кроме `attachEvent` , обеспечивают доступ к элементу, на котором сработал обработчик, через `this` .

# Всплытие и перехват

Источник: <http://learn.javascript.ru/bubbling-and-capturing>

Элементы DOM могут быть вложены друг в друга. При этом обработчик, привязанный к родителю, срабатывает, даже если посетитель кликнул по потомку.

Это происходит потому, что событие всплывает.

Например, этот обработчик для `div` работает, если вы кликните по вложенному тегу `em` или `code` :

```
<div onclick="alert('Обработчик для Div сработал!')">
  <em>Кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>
</div>
```

## Всплытие

Основной принцип всплытия:

После того, как событие сработает на самом вложенном элементе, оно также сработает на родителях, вверх по цепочке вложенности.

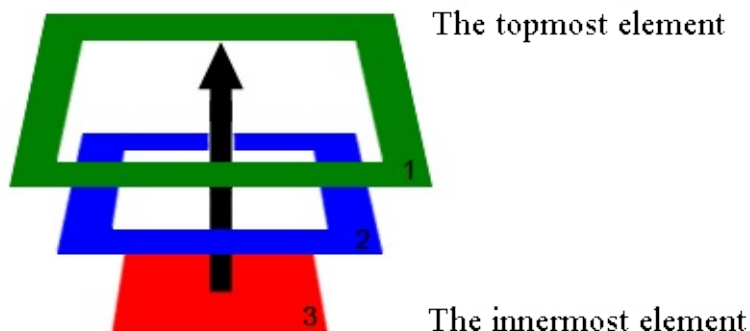
Например, есть 3 вложенных блока:

```
<!DOCTYPE HTML>
<html>
<body>
<link type="text/css" rel="stylesheet" href="example.css">

<div class="d1">1 <!-- внешний (topmost) -->
  <div class="d2">2
    <div class="d3">3 <!-- внутренний (innermost) -->
  </div>
</div>
</div>

</body>
</html>
```

Всплытие гарантирует, что клик по внутреннему `div 3` вызовет событие `onclick` сначала на внутреннем элементе 3, затем на элементе 2 и в конце концов на элементе 1.



Этот процесс называется *всплытием*, потому что события «всплывают» от внутреннего элемента вверх через родителей, подобно тому, как всплывает пузырек воздуха в воде.

Текущий элемент, `this`

Элемент, на котором сработал обработчик, доступен через `this`.

Например, повесим на клик по каждому `div` функцию `highlight`, которая подсвечивает текущий элемент:

```
<!DOCTYPE HTML>
<html>
<body>
<link type="text/css" rel="stylesheet" href="example.css">

<div class="d1" onclick="highlight(this)">1
  <div class="d2" onclick="highlight(this)">2
    <div class="d3" onclick="highlight(this)">3
  </div>
</div>
</div>

<script>
function highlight(elem) {
  elem.style.backgroundColor = 'yellow';
  alert(elem.className);
  elem.style.backgroundColor = '';
}
</script>

</body>
</html>
```

Демонстрация: <http://learn.javascript.ru/files/tutorial/browser/events/bubbling/bubble/index.html>

Также существует свойство объекта события `event.currentTarget`.

Оно имеет тот же смысл, что и `this`, поэтому с первого взгляда избыточно, но иногда гораздо удобнее получить текущий элемент из объекта события.

**IE8- при назначении обработчика через `attachEvent` не передаёт `this`.** Браузеры IE8- не предоставляют `this` при назначении через `attachEvent`. Также в них нет свойства `event.currentTarget`.

Если вы будете использовать фреймворк для работы с событиями, то это не важно, так как он всё исправит. А при разработке на чистом JS имеет смысл вешать обработчики через опсобытие: это и кросс-браузерно, и `this` всегда есть.

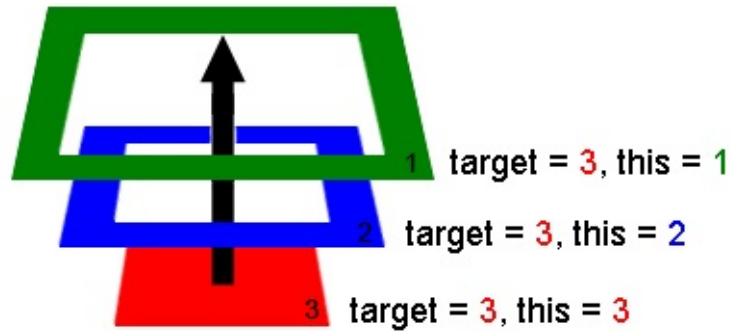
## Целевой элемент, `event.target`

Самый глубокий элемент, который вызывает событие, называется «целевым» или «исходным» элементом.

В IE<9 он доступен как `event.srcElement`, остальные браузеры используют `event.target`. Кроссбраузерное решение выглядит так:

```
var target = event.target || event.srcElement;
```

- `event.target/srcElement` - означает исходный элемент, на котором произошло событие.
- `this` - текущий элемент, до которого дошло всплытие и который запускает обработчик.



Демонстрация: <http://learn.javascript.ru/files/tutorial/browser/events/bubbling/bubble-target/index.html>

## Прекращение всплытия

Всплытие идет прямо вверх. Обычно оно будет всплывать до , а затем до document, вызывая все обработчики на своем пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Сценарий, при котором это может быть нужно:

1. На странице по правому клику показывается, при помощи JavaScript, специальное контекстное меню;
2. На странице также есть таблица, которая показывает меню, но другое, своё;
3. В случае правого клика по таблице её обработчик покажет меню и остановит всплытие, чтобы меню уровня страницы не показалось.

Код для остановки всплытия различается между IE<9 и остальными браузерами:

Стандартный код — это вызов метода:

```
event.stopPropagation()
```

Для IE<9 — это назначение свойства:

```
event.cancelBubble = true
```

Кросс-браузерное решение:

```
element.onclick = function(event) {  
  event = event || window.event; // Кроссбраузерно получить событие  
  
  if (event.stopPropagation) { // существует ли метод?  
    // Стандартно:  
    event.stopPropagation();  
  } else {  
    // Вариант IE  
    event.cancelBubble = true;  
  }  
}
```

Есть еще вариант записи в одну строчку:

```
event.stopPropagation ? event.stopPropagation() : (event.cancelBubble=true);
```

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены.

Например, если на ссылке есть два `onclick`-обработчика, то остановка всплытия для одного из них никак не скажется на другом. Это логично, учитывая то, что, как мы уже говорили ранее, браузер не гарантирует взаимный порядок выполнения этих обработчиков. Они полностью независимы, и из одного нельзя отменить другой.

Всплытие — это удобно. Не прекращайте его без явной нужды, очевидной и архитектурно прозрачной.

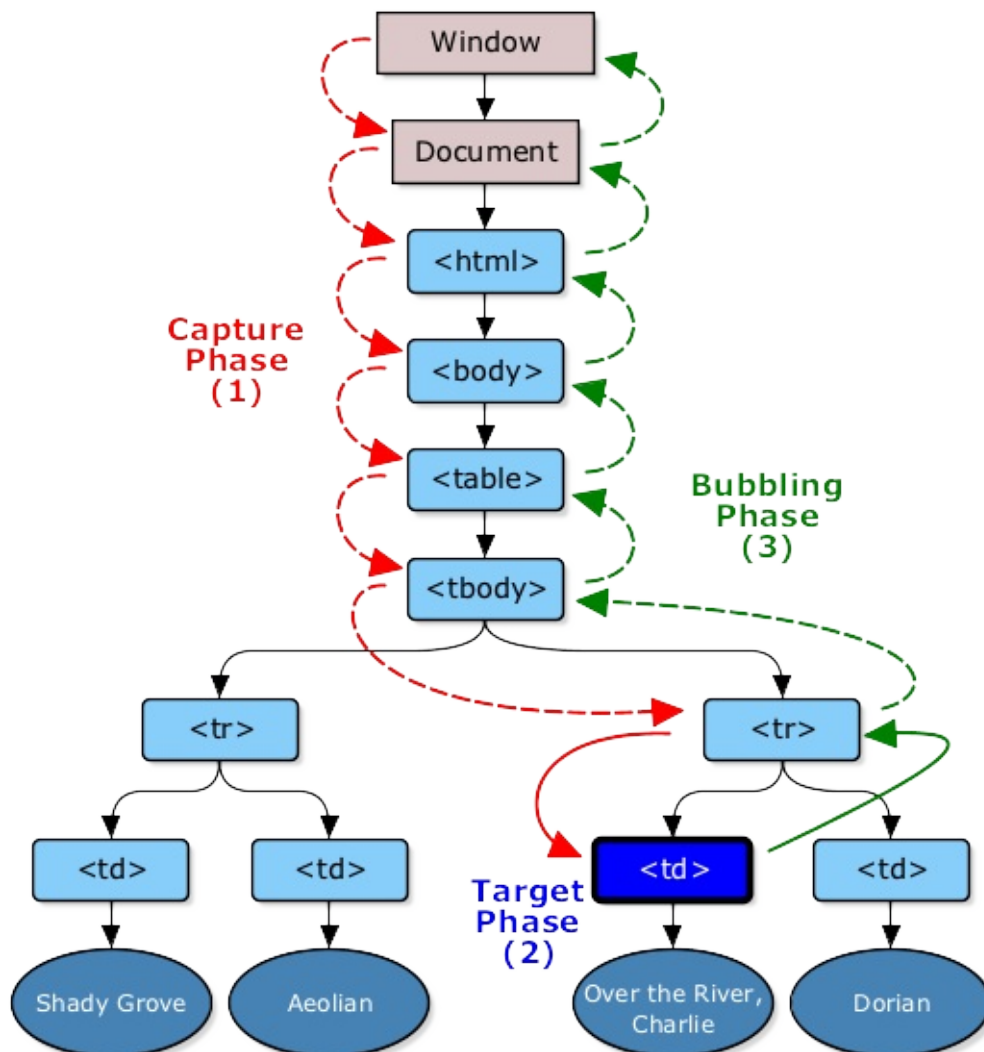
Зачастую прекращение всплытия создаёт свои подводные камни, которые потом приходится обходить.

Например, вы для одного компонента интерфейса сделали `stopPropagation` на событие `click`. А позже, на совсем другом месте страницы понадобилось отследить «клик вне элемента» — скажем, чтобы закрыть пункт меню. Обычно для этого ставят обработчик `document.onclick` и по `event.target` проверяют, внутри был клик или нет. Но над областью, где клики убиваются `stopPropagation`, такой способ будет нерабочим!

## Три стадии прохода событий

Во всех браузерах, кроме IE<9, есть три стадии прохода события.

Событие сначала идет сверху вниз. Эта стадия называется «стадия перехвата» (capturing stage). Событие достигло целевого элемента. Это — «стадия цели» (target stage). После этого событие начинает всплывать. Это — «стадия всплытия» (bubbling stage). Получается такая картина:



Третий аргумент `addEventListener` позволяет задать стадию, на которой будет поймано событие.

- Если аргумент `true`, то событие будет перехвачено по дороге вниз.
- Если аргумент `false`, то событие будет поймано при всплытии.

Стадия цели как-то особо не обрабатывается, но обработчики, назначаемые на стадии захвата и всплытия, срабатывают также на целевом элементе.

**Обработчики, добавленные другими способами, ничего не знают о стадии перехвата, а начинают работать со всплытия.**

Код:

```
<!DOCTYPE HTML>
<html>
<body>
<link type="text/css" rel="stylesheet" href="example.css">

<div class="d1">1
  <div class="d2">2
    <div class="d3">3
    </div>
  </div>
</div>

<script>
var divs = document.getElementsByTagName('div');

// на каждый DIV повесить обработчик на стадии захвата
for(var i = 0; i < divs.length; i++) {
  divs[i].addEventListener("click", highlightThis, true);
}

function highlightThis() {
  this.style.backgroundColor = 'yellow';
  alert(this.className);
  this.style.backgroundColor = '';
}
</script>

</body>
</html>
```

Демонстрация: <http://learn.javascript.ru/files/tutorial/browser/events/bubbling/capture/index.html>

Чтобы назначить обработчики для обеих стадий, добавим новое событие: `var divs = document.getElementsByTagName('div');`

```
for (var i=0; i<divs.length; i++) {
  divs[i].addEventListener("click", highlightThis, true);
  divs[i].addEventListener("click", highlightThis, false);
}
```

Как видно из примера, один и тот же обработчик можно назначить на разные стадии. При этом номер текущей стадии он, при необходимости, может получить из свойства `event.eventPhase`.

## Итого

- Событие идет сначала сверху вниз к целевому элементу (стадия захвата), затем всплывает снизу вверх. В IE<9 стадия захвата отсутствует.
- Все способы добавления обработчика используют стадию всплытия, кроме `addEventListener` с последним аргументом `true`.
- Всплытие/захват можно остановить с помощью вызова `event.stopPropagation()`. В IE<9 нужно использовать для ЭТОГО `event.cancelBubble=true`.



# Действия браузера по умолчанию

Источник: <http://learn.javascript.ru/default-browser-action>

Многие события влекут за собой действие браузера.

Например, клик по ссылке инициирует переход на новый URL, нажатие на кнопку «отправить» в форме — отправку ее на сервер, и т.п.

Если логика работы обработчика требует отменить действие браузера — это возможно.

## Отмена действия браузера

Есть два основных способа.

Первый способ — это воспользоваться объектом события. Для отмены действия браузера существует стандартный метод `event.preventDefault()`, ну а для IE<9 нужно назначить свойство `event.returnValue = false`.

Кроссбраузерный код:

```
element.onclick = function(event) {
    event = event || window.event

    if (event.preventDefault) { // если метод существует
        event.preventDefault();
    } else { // вариант IE<9:
        event.returnValue = false;
    }
}
```

Можно записать в одну строку:

```
...
event.preventDefault ? event.preventDefault() : (event.returnValue=false);
...
```

Если обработчик назначен через `on...`, то `return false` из обработчика отменяет действие браузера:

```
element.onclick = function(event) {
    ...
    return false;
}
```

Такой способ проще, но не будет работать, если обработчик назначен через `addEventListener/attachEvent`.

При возврате `return false` из обработчика, назначенного через `on...`, действие браузера будет отменено.

Иногда в коде начинающих разработчиков можно увидеть `return` других значений. Они никак не обрабатываются. Можно вернуть всё, что угодно: строку, число, `null`, `undefined` — всё, кроме `false`, игнорируется.

## Действия, которые нельзя отменить

Есть действия браузера, которые происходят до вызова обработчика. Такие действия нельзя отменить.

Например, при клике по ссылке происходит фокусировка. Большинство браузеров выделяют такую ссылку пунктирной границей. Можно и указать свой стиль в CSS для псевдоселектора `:focus`.



Фокусировку нельзя предотвратить из обработчика `onfocus`, поскольку обработчик вызывается уже после того, как она произошла.

С другой стороны, переход по URL происходит после клика по ссылке, поэтому его можно отменить.

```
<style>
a:focus { border: 1px solid black }
</style>

<a href="/" onclick="return false" onfocus="return false">
  По клику произойдет фокусировка, но перехода не будет
</a>
```

Действие браузера по умолчанию и всплытие взаимно независимы.

**Отмена действия браузера не остановит всплытие и наоборот.**

Если хотите отменить и то и другое:

```
function stop(e) {
  if (e.preventDefault) { // стандарт
    e.preventDefault();
    e.stopPropagation();
  } else { // IE8-
    e.returnValue = false;
    e.cancelBubble = true;
  }
}
```

## Итого

- Браузер имеет встроенные действия при ряде событий – переход по ссылке, отправка формы и т.п. Как правило, их можно отменить.
- Есть два способа отменить действие по умолчанию: первый – использовать `event.preventDefault()` (IE<9: `event.returnValue=false`), второй – `return false` из обработчика. Второй способ работает только если обработчик назначен через `on`-свойство.
- Действие браузера само по себе, всплытие события – само по себе. Они никак не связаны.

# Объект "событие" (event)

Источник: <http://javascript.ru/tutorial/events/intro#obekt-sobytie-event>

Объект событие всегда передается обработчику и содержит массу полезной информации о том где и какое событие произошло.

Способов передачи этого объекта обработчику существует ровно два, и они зависят от способа его установки и от браузера.

## W3C

В браузерах, работающих по рекомендациям W3C, объект события всегда передается в обработчик первым параметром.

Например:

```
function doSomething(event) {  
    // event - будет содержать объект события  
}  
  
element.onclick = doSomething;
```

При вызове обработчика объект события event будет передан ему первым аргументом.

Можно назначить и вот так:

```
element.onclick = function(event) {  
    // event - объект события  
}
```

Интересный побочный эффект - в возможности использования переменной `event` при назначении обработчика в HTML:

```
<input type="button" onclick="alert(event)" value="Жми сюда не ошибешься"/>
```

**Жми сюда не ошибешься**

Это работает благодаря тому, что браузер автоматически создает функцию-обработчик с данным телом, в которой первый аргумент `event`.

## Internet Explorer

В Internet Explorer существует глобальный объект `window.event`, который хранит в себе информацию о последнем событии. А первого аргумента обработчика просто нет.

То есть, все должно работать так:

```
// обработчик без аргументов  
function doSomething() {  
    // window.event - объект события  
}  
  
element.onclick = doSomething;
```

Обратите внимание, что доступ к `event` при назначении обработчика в HTML (см. пример выше) по-прежнему будет работать. Такой вот надежный и простой кросс-браузерный доступ к объекту события.

## Кросс-браузерное решение

Можно кросс-браузерно получить объект события, используя такой приём:

```
function doSomething(event) {
    event = event || window.event

    // Теперь event - объект события во всех браузерах.
}

element.onclick = doSomething
```

## Получение события при inline-записи

Как мы уже говорили раньше, при описании обработчика события в HTML-разметке для получения события можно использовать переменную с названием `event`.

```
<input type="button" onclick="alert(event.type)" value="Нажми меня"/>
```

Этот код в действии:

Нажми меня

Это совершенно кросс-браузерный способ, так как по стандарту `event` - название первого аргумента функции-обработчика, которую автоматом создаст браузер; ну а в IE значение `event` будет взято из глобального объекта `window`.

## Что дает объект события?

Из объекта события обработчик может узнать, на каком элементе оно произошло, каковы были координаты мыши (для событий, связанных с мышью), какая клавиша была нажата (для событий, связанных с клавиатурой), и извлечь другую полезную информацию.

Например, для события по клику мыши `onclick`, свойство `event.target` (в IE `event.srcElement`) содержит DOM-элемент, на котором этот клик произошел.

# События мыши

---

Источник: <http://learn.javascript.ru/mouse-event>

1. Введение: клики, кнопка, координаты; **READ**
2. События движения: "mouseover/out/move/leave/enter"; **READ**
3. Колёсико мыши: "wheel" и аналоги;
4. Устранение IE-несовместимостей: "fixEvent".

# Введение: клики, кнопка, координаты

---

Источник: <http://learn.javascript.ru/mouse-clicks>

## Типы событий мыши

Условно можно разделить события на два типа: «простые» и «комплексные».

### Простые события

- **mousedown**
  - Кнопка мыши нажата над элементом.
- **mouseup**
  - Кнопка мыши отпущена над элементом.
- **mouseover**
  - Мышь появилась над элементом.
- **mouseout**
  - Мышь ушла с элемента.
- **mousemove**
  - Каждое движение мыши над элементом генерирует это событие.

### Комплексные события

- **click**
  - Вызывается при клике мышью, то есть при mousedown, а затем mouseup на одном элементе
- **contextmenu**
  - Вызывается при клике правой кнопкой мыши на элементе.
- **dblclick**
  - Вызывается при двойном клике по элементу.

Комплексные можно составить из простых, поэтому в теории можно было бы обойтись вообще без них. Но они есть, и это хорошо, потому что с ними удобнее.

## Порядок срабатывания событий

Одно действие может вызывать несколько событий.

Например, клик вызывает сначала `mousedown` при нажатии, а затем `mouseup` и `click` при отпускании кнопки.

В тех случаях, когда одно действие генерирует несколько событий, их порядок фиксирован. То есть, обработчики вызовутся в порядке `mousedown -> mouseup -> click`.

Каждое событие обрабатывается независимо. Например, при клике события `mouseup + click` возникают одновременно, но обрабатываются последовательно. Сначала полностью завершается обработка `mouseup`, затем запускается `click`.

## Получение информации о кнопке: which/button

При обработке событий, связанных с кликами мыши, бывает важно знать, какая кнопка нажата.

Для получения кнопки мыши в объекте `event` есть два свойства: `which` и `button`. Они хранят кнопку в численном значении, хотя и с некоторыми браузерными несовместимостями.

На практике они используются редко, т.к. обычно обработчик вешается либо `onclick` — только на левую кнопку мыши, либо `oncontextmenu` — только на правую.

## Стандартные свойства

По стандарту, у мышиных событий есть свойство `which`, которое работает одинаково во всех браузерах, кроме IE, имеет следующие значения:

- `which == 1` - левая кнопка
- `which == 2` - средняя кнопка
- `which == 3` - правая кнопка

Это свойство работает везде, кроме IE<9.

## Свойство `button` для IE<9

Вообще говоря, старый способ Microsoft, который поддерживается IE, более универсален, чем `which`.

В нём для получения кнопки используется свойство `button`, которое является 3-х битным числом, в котором каждому биту соответствует кнопка мыши. Бит установлен в 1, только если соответствующая кнопка нажата.

Чтобы его расшифровать — нужна побитовая операция `&` («битовое И»):

- `!!(button & 1) == true` (1й бит установлен), если нажата левая кнопка,
- `!!(button & 2) == true` (2й бит установлен), если нажата правая кнопка,
- `!!(button & 4) == true` (3й бит установлен), если нажата средняя кнопка. При этом мы можем узнать, были ли две кнопки нажаты одновременно.

В современном IE поддерживаются оба способа: и `which` и `button`.

Наиболее удобный кросс-браузерный подход — это взять за основу стандартное свойство `which`. При его отсутствии (в IE8-), создавать его по значению `button`.

Функция, которая добавляет `which`, если его нет:

```
function fixWhich(e) {
  if (!e.which && e.button) { // если which нет, но есть button...
    if (e.button & 1) e.which = 1; // левая кнопка
    else if (e.button & 4) e.which = 2; // средняя кнопка
    else if (e.button & 2) e.which = 3; // правая кнопка
  }
}
```

## Правый клик: `oncontextmenu`

При клике правой кнопкой мыши браузер показывает свое контекстное меню. Это является его действием по умолчанию:

```
<button oncontextmenu="alert('Клик!');">Правый клик сюда</button>
```

Правый клик сюда

Но если установлен обработчик события, то он может отменить действие по умолчанию и, тем самым, предотвратить появление встроенного меню.

В примере ниже меню не будет:

```
<button oncontextmenu="alert('Клик!');return false">Правый клик сюда</button>
```

Правый клик сюда

## Модификаторы shift, alt, ctrl и meta

Во всех событиях мыши присутствует информация о нажатых клавишах-модификаторах.

Соответствующие свойства:

- `shiftKey`
- `altKey`
- `ctrlKey`
- `metaKey` (для Mac)

Например, кнопка ниже сработает только на Ctrl+Shift+Клик:

```
<button>Ctrl+Shift+Кликни меня!</button>

<script>
  document.body.children[0].onclick = function(e) {
    e = e || event;
    if (!e.ctrlKey || !e.shiftKey) return;
    alert('Ура!');
  }
</script>
```

## Координаты мыши

Все мышинные события предоставляют текущие координаты курсора в двух видах: относительно окна и относительно документа.

### Относительно окна: `clientX/Y`

Есть отличное кросс-браузерное свойство `clientX` ( `clientY` ), которое содержит координаты относительно `window` .

При этом, например, если ваше окно размером 500x500, а мышь находится в центре, тогда и `clientX` и `clientY` будут равны 250.

Если прокрутите вниз, влево или вверх не сдвигая при этом мышь, то значения `clientX/clientY` не изменятся, потому что они считаются относительно окна, а не документа.

```
<input onmousemove="this.value = event.clientX+' '+event.clientY">
```

Проведите мышью над полем ввода, чтобы увидеть `clientX/clientY` :



### Относительно документа: `pageX/Y`

Обычно, для обработки события нам нужно знать позицию мыши относительно документа, вместе со скроллом. Для этого стандарт W3C предоставляет пару свойств `pageX/pageY` .

Если ваше окно размером 500x500, а мышь находится в центре, тогда и `pageX` и `pageY` будут равны 250. Если вы прокрутите страницу на 250 пикселей вниз, то значение `pageY` станет равным 500.

Итак, пара `pageX/pageY` содержит координаты относительно левого верхнего угла документа, вместе со всеми прокрутками.

Эти свойства поддерживаются всеми браузерами, кроме IE<9.

```
<input onmousemove="this.value = event.pageX+' '+event.pageY">
```

### Обходной путь для IE<9:

В IE до версии 9 не поддерживаются свойства `pageX/pageY`, но их можно получить, прибавив к `clientX/clientY` величину прокрутки страницы.

Более подробно о её вычислении вы можете прочитать в разделе прокрутка страницы.

Мы же здесь приведем готовый вариант, который позволяет нам получить `pageX/pageY` для старых IE:

```
function fixPageXY(e) {
  if (e.pageX == null && e.clientX != null) { // если нет pageX..
    var html = document.documentElement;
    var body = document.body;

    e.pageX = e.clientX + (html.scrollLeft || body && body.scrollLeft || 0);
    e.pageX -= html.clientLeft || 0;

    e.pageY = e.clientY + (html.scrollTop || body && body.scrollTop || 0);
    e.pageY -= html.clientTop || 0;
  }
}
```

### Демо получения координат мыши:

Следующий пример показывает координаты мыши относительно документа, для всех браузеров.

```
document.onmousemove = function(e) {
  e = e || window.event;
  fixPageXY(e);

  document.getElementById('mouseX').value = e.pageX;
  document.getElementById('mouseY').value = e.pageY;
}
```

## Итого

События мыши имеют следующие свойства:

- Кнопка мыши: `which` (для IE<9: нужно ставить из `button` )
- Элемент, вызвавший событие: `target`
- Координаты, относительно окна: `clientX/clientY` Координаты, относительно документа: `pageX/pageY` (для IE<9: нужно ставить по `clientX/Y` и прокрутке) Если зажата спец. клавиша, то стоит соответствующее свойство: `altKey`, `ctrlKey`, `shiftKey` или `metaKey` (Mac).



# События движения: "mouseover/out/move/leave/enter"

Источник: <http://learn.javascript.ru/mousemove-events>

В этой главе мы рассмотрим события, возникающие при движении мыши над элементами.

## События `mouseover/mouseout`, свойство `relatedTarget`

Событие `mouseover` происходит, когда мышь появляется над элементом, а `mouseout` — когда уходит из него.

В этих событиях мышь переходит с одного элемента на другой. Оба этих элемента можно получить из свойств объекта события.

- **`mouseover`**
  - Элемент под курсором — `event.target` (IE: `srcElement`).
  - Элемент, с которого курсор пришел — `event.relatedTarget` (IE: `fromElement`).
- **`mouseout`**
  - Элемент, с которого курсор пришел — `event.target` (IE: `srcElement`).
  - Элемент под курсором — `event.relatedTarget` (IE: `toElement`).

Как вы видите, спецификация W3C объединяет `fromElement` и `toElement` в одно свойство `relatedTarget`, которое работает, как `fromElement` для `mouseover` и как `toElement` для `mouseout`.

В IE это свойство можно поставить так:

```
function fixRelatedTarget(e) {
  if (!e.relatedTarget) {
    if (e.type == 'mouseover') e.relatedTarget = e.fromElement;
    if (e.type == 'mouseout') e.relatedTarget = e.toElement;
  }
}
```

Значение `relatedTarget` (`toElement/fromElement`) может быть `null`. Такое бывает, например, когда мышь приходит из-за пределов окна у `mouseover` будет `relatedTarget = null`.

## Частота событий `mousemove` и `mouseover`

Событие `mousemove` срабатывает при передвижении мыши. Но это не значит, что каждый пиксель экрана порождает отдельное событие!

События `mousemove` и `mouseover/mouseout` срабатывают с такой частотой, с которой это позволяет внутренний таймер браузера.

Это означает, что если вы двигаете мышью очень быстро, то DOM-элементы, через которые мышь проходит на большой скорости, могут быть пропущены.

- Курсор может быстро перейти над родительским элементом в дочерний, при этом не вызвав событий на родителе, как будто он через родителя никогда не проходил.
- Курсор может быстро выйти из дочернего элемента без генерации событий на родителе.

Несмотря на некоторую концептуальную странность такого подхода, он весьма разумен. Хотя браузер и может пропустить промежуточные элементы, он гарантирует, что если уж мышь зашла на элемент (сработало событие `mouseover`), то при выходе с него сработает и `mouseout`. Не может быть `mouseover` без `mouseout` и наоборот.

Так что эти события позволяют надёжно обрабатывать заход на элемент и уход с него.

## «Лишний» mouseout при уходе на потомка

Представьте ситуацию — курсор зашел на элемент. Сработал `mouseover` на нём. Потом курсор идёт на дочерний... И, оказывается, на элементе-родителе при этом происходит `mouseout` ! Как будто курсор с него ушёл, хотя он всего лишь перешёл на потомка.

Это происходит потому, что согласно браузерной логике, курсор мыши может быть только над одним элементом — самым глубоким в DOM (и верхним по `z-index` ).

Так что если он перешел на потомка — значит ушёл с родителя.

Получается, что при переходе на потомка курсор уходит `mouseout` с родителя, а затем тут же переходит `mouseover` на него. Причем возвращение происходит за счёт всплытия `mouseover` с потомка.

## События `mouseenter` и `mouseleave`.

События `mouseenter/mouseleave` похожи на `mouseover/mouseout` . Они тоже срабатывают, когда курсор заходит на элемент и уходит с него, но с двумя отличиями.

1. При переходе на потомка курсор не уходит с родителя.
2. То есть, эти события более интуитивно понятны. Курсор заходит на элемент — срабатывает `mouseenter` , а затем — неважно, куда он внутри него переходит, `mouseleave` будет, когда курсор окажется за пределами элемента.

События `mouseenter/mouseleave` не всплывают.

## Преобразование `mouseover/out` в `mouseenter/leave`

Для браузеров, в которых нет поддержки этих событий, можно повесить обработчик на `mouseover/mouseout` , а лишние события — фильтровать.

При `mouseout` можно получить элемент, на который осуществляется переход ( `e.relatedTarget` ), и проверить — является ли новый элемент потомком родителя. Если да — мышь с родителя не уходила, игнорировать это событие. При `mouseover` — аналогичным образом проверить, мышь «пришла» с потомка? Если с потомка, то это не настоящий переход на родителя, игнорировать. Посмотрим, как это выглядит, на примере кода:

```
<div style="padding:10px; margin:10px; border: 2px solid blue" id="outer">
  <p style="border: 1px solid green">
    Обработчики mouseover/mouseout стоят на синем родителе.
  </p>
  <blockquote style="border: 1px solid red">
    ..Но срабатывают и при любых переходах по его потомкам!
  </blockquote>
</div>
<b id="info">Тут будет информация о событиях.</b>

<script>
var outer = document.getElementById('outer')
var info = document.getElementById('info');

outer.onmouseout = function(e) {
  e = e || event;
  var target = e.target || e.srcElement;
  info.innerHTML = e.type+' , target:'+target.tagName;
};

outer.onmouseover = function(e) {
  e = e || event;
  var target = e.target || e.srcElement;
  info.innerHTML = e.type+' , target:'+target.tagName;
};

</script>
```

## Итого

У `mouseover` , `mousemove` , `mouseout` есть следующие особенности:

1. События `mouseover` и `mouseout` — единственные, у которых есть вторая цель: `relatedTarget` ( `toElement/fromElement` в IE).
2. Событие `mouseout` срабатывает, когда мышь уходит с родительского элемента на дочерний. Используйте `mouseenter/mouseleave` или фильтруйте их, чтобы избежать излишнего реагирования.
3. При быстром движении мыши события `mouseover` , `mousemove` , `mouseout` могут пропускать промежуточные элементы. Мышь может моментально возникнуть над потомком, миновав при этом его родителя.

События `mouseleave/mouseenter` поддерживаются не во всех браузерах, но их можно эмулировать, отсеивая лишние срабатывания `mouseover/mouseout` .

# Задачи

## Задача 1

Используя JavaScript, сделайте так, чтобы при клике на кнопку исчезал элемент с `id="hide"` .

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <input type="button" id="hider" value="Нажмите, чтобы спрятать текст"/>
  <div id="hide">Текст</div>
<script>
  /* ваш код */
</script>
</body>
</html>
```

## Решение

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <input type="button" id="hider" value="Нажмите, чтобы спрятать текст"/>
  <div id="hide">Текст</div>
<script>
  document.getElementById('hider').onclick = function() {
    document.getElementById('hide').style.display = 'none';
  }
</script>
</body>
</html>
```

## Задача 2

Создайте кнопку, при клике на которую, она будет скрывать сама себя.

## Решение

Решение задачи заключается в использовании `this` в обработчике.

```
<input type="button" onclick="this.style.display='none'" value="Нажми, чтобы меня спрятать"/>
```

## Задача 3

В переменной `button` находится кнопка.

Изначально обработчиков на ней нет.

Что будет выведено при клике после выполнения кода?

```
button.addEventListener("click", function() { alert("1"); }, false);

button.removeEventListener("click", function() { alert("1"); }, false);
```

```
button.onclick = function() { alert(2); };
```

## Решение

Ответ: будет выведено 1 и 2.

Первый обработчик сработает, так как он не убран вызовом `removeEventListener`. Для удаления обработчика нужно передать в точности ту же функцию (ссылку на нее), что была назначена, а в коде передается такая же с виду функция, но, тем не менее, это другой объект.

Для того, чтобы удалить функцию-обработчик, нужно где-то сохранить ссылку на неё, например так:

```
function handler() {  
  alert("1");  
}  
  
button.addEventListener("click", handler, false);  
button.removeEventListener("click", handler, false);
```

Обработчик `button.onclick` работает независимо и в дополнение к назначенному в `addEventListener`.

## Задача 4

Есть список сообщений. Добавьте каждому сообщению по кнопке для его скрытия. Картинка для кнопки удаления:



Расположение кнопок:

## Лошадь



Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади, вымершей в дикой природе, за исключением небольшой популяции лошади Пржевальского.

## Осёл



Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства и культуры человека. Все одомашненные ослы относятся к африканским ослам.

## Корова, а также пара слов о диком быке, о волах и о тёлках.



Коро́ва — самка домашнего быка, одомашненного подвида дикого быка, парнокопытного жвачного животного семейства полорогих. Самцы вида называются быками, молодняк — телятами, кастрированные самцы — волами. Молодых (до первой стельности) самок называют тёлками.

```
<!DOCTYPE HTML>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="messages.css">
  <meta charset="utf-8">
  <style>
    body {
      margin: 10px auto;
      width: 470px;
    }
    h3 {
      margin: 0;
      padding-bottom: .3em;
      font-size: 1.1em;
    }
    p {
      margin: 0;
      padding: 0 0 .5em;
    }
    .pane {
      background: #edf5e1;
      padding: 10px 20px 10px;
      border-top: solid 2px #c4df9b;
    }
  </style>
</head>

<body>

<div>
  <div class="pane">
    <h3>Лошадь</h3>
    <p>Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой
  </div>
```

```

<div class="pane">
  <h3>Осёл</h3>
  <p>Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйс
</div>
<div class="pane">
  <h3>Корова, а также пара слов о диком быке, о волах и о тёлках. </h3>
  <p>Корова — самка домашнего быка, одомашненного подвид дикого быка, парнокопытного жвачного животного семейства
</div>
</div>

</body>
</html>

```

Как лучше отобразить кнопку справа-сверху: через `position: absolute` или `float` ?

Исходный документ <http://learn.javascript.ru/play/tutorial/browser/events/messages-src/index.html>

## Решение

### Алгоритм решения:

1. Разработать структуру HTML/CSS. Позиционировать кнопку внутри сообщения;
2. Найти все кнопки;
3. Присвоить им обработчики;
4. Обработчик будет ловить событие на кнопке и удалять соответствующий элемент.

### Вёрстка:

Исправьте HTML/CSS, чтобы кнопка была в нужном месте сообщения. Кнопку лучше сделать как `div`, а картинка — будет его `background`. Это более правильно, чем `img`, т.к. в данном случае картинка является оформлением кнопки, а оформление должно быть в CSS.

Расположить кнопку справа можно при помощи `position: relative` для `pane`, а для кнопки `position: absolute + right/top`. Так как `position: absolute` вынимает элемент из потока, то кнопка может перекрывать текст заголовка. Чтобы этого не произошло, можно добавить `padding-right` к заголовку.

Потенциальным преимуществом способа с `position` по сравнению с `float` в данном случае является возможность поместить элемент кнопки в HTML после текста, а не до него.

### Обработчики

Для того, чтобы получить кнопку из контейнера, можно найти все `IMG` в нём и выбрать из них кнопку по `className`. На каждую кнопку можно повесить обработчик.

### Решение

```

<!DOCTYPE HTML>
<html>
<head>
  <link rel="stylesheet" href="messages.css">
  <meta charset="utf-8">
  <style>
    body {
      margin: 10px auto;
      width: 470px;
    }
    h3 {
      margin: 0;
      padding-bottom: .3em;
      padding-right: 20px;
      font-size: 1.1em;
    }
    p {
      margin: 0;
      padding: 0 0 .5em;
    }
  </style>
</head>
<body>
  <div class="pane">
    <h3>Осёл</h3>
    <p>Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйс
  </div>
  <div class="pane">
    <h3>Корова, а также пара слов о диком быке, о волах и о тёлках. </h3>
    <p>Корова — самка домашнего быка, одомашненного подвид дикого быка, парнокопытного жвачного животного семейства
  </div>
</div>
</body>
</html>

```

```

    }
    .pane {
        background: #edf5e1;
        padding: 10px 20px 10px;
        border-top: solid 2px #c4df9b;
        position: relative;
    }

    .remove-button {
        position: absolute;
        top: 10px;
        right: 10px;
        cursor: pointer;
        display: block;
        background: url(delete.gif) no-repeat;
        width: 16px;
        height: 16px;
    }
</style>
</head>

<body>

<div id="messages-container">
    <div class="pane">
        <h3>Лошадь</h3>
        <p>Домашняя лошадь – животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади.</p>
        <span class="remove-button"></span>
    </div>
    <div class="pane">
        <h3>Осёл</h3>
        <p>Домашний осёл или ишак – одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства человека.</p>
        <span class="remove-button"></span>
    </div>
    <div class="pane">
        <h3>Корова, а также пара слов о диком быке, о волках и о тёлках.</h3>
        <p>Корова – самка домашнего быка, одомашненного подвида дикого быка, парнокопытного жвачного животного семейства Bovidae.</p>
        <span class="remove-button"></span>
    </div>
</div>

<script>
    var spans = document.getElementById('messages-container').getElementsByTagName('span');

    for(var i=0; i<spans.length; i++) {
        var span = spans[i];
        if (span.className != 'remove-button') continue;

        span.onclick = function() {
            var el = this.parentNode;
            el.parentNode.removeChild(el);
        };
    }
</script>

</body>
</html>

```

Также решение показано тут: <http://learn.javascript.ru/play/tutorial/browser/events/messages/index.html>

Для поиска элементов span с нужным классом в нём используется `getElementsByTagName` с фильтрацией. К сожалению, это единственный способ, доступный в IE 6,7. Если же эти браузеры вам не нужны, то гораздо лучше – искать элементы при помощи `querySelector` или `getElementsByClassName`.

## Задача 5

Почему в этом документе `return false` не работает?

```

<script>
    function handler() {
        alert("...");
        return false;
    }
</script>

```



```
<a href="http://w3.org" onclick="handler()">w3.org</a>
```

По замыслу, переход на w3.org при клике должен отменяться. Однако, на самом деле он происходит.

В чём дело и как поправить, сохранив `onclick` в HTML?

## Решение

Дело в том, что обработчик из атрибута `onclick` делается браузером как функция с заданным телом.

То есть, он будет таким:

```
function(event) {  
    handler()  
}
```

При этом возвращаемое `handler` значение никак не используется и не влияет на результат.

Рабочий вариант:

```
<script>  
    function handler() {  
        alert("...");  
        return false;  
    }  
</script>  
  
<a href="http://w3.org" onclick="return handler()">w3.org</a>
```

Альтернатива – передать и использовать объект события для вызова `event.preventDefault()` (или кросс-браузерного варианта для поддержки старых IE).

```
<script>  
    function handler(event) {  
        alert("...");  
        event.preventDefault ? event.preventDefault() : (event.returnValue=false);  
    }  
</script>  
  
<a href="http://w3.org" onclick="handler(event)">w3.org</a>
```

## Задача 6

Эта задача состоит из трёх частей.

1. Сделайте список, элементы которого можно выделять кликом.
2. Добавьте мульти-выделение. Если клик с нажатым `ctrl` (`cmd` под Mac), то элемент добавляется-удаляется из выделенных.
3. Добавьте выделение промежутков. Если происходит клик с нажатым `shift`, то к выделению добавляется промежуток элементов от предыдущего кликнутого до этого. При этом не важно, какое именно действие делал предыдущий клик. Это похоже на то, как работает файловый менеджер в ряде ОС, но чуть проще, так как конкретная реализация выделений различается у разных ОС, и её точное воспроизведение не входит в эту задачу. Исходный документ <http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-src.html> или исходный код:

```

<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <style>
    .selected {
      background: #0f0;
    }
    li {
      cursor: pointer;
    }
  </style>
</head>
<body>

<ul>
<li>Кристофер Робин</li>
<li>Винни-Пух</li>
<li>Ослик Иа</li>
<li>Мудрая Сова</li>
<li>Кролик. Просто кролик.</li>
</ul>

<script>

// ... ваш код

// --- вспомогательная функция, может понадобится для части 3 ---
// http://learn.javascript.ru/compare-document-position
function compareDocumentPosition(a, b) {
  return a.compareDocumentPosition ?
    a.compareDocumentPosition(b) :
    (a != b && a.contains(b) && 16) +
    (a != b && b.contains(a) && 8) +
    (a.sourceIndex >= 0 && b.sourceIndex >= 0 ?
      (a.sourceIndex < b.sourceIndex && 4) +
      (a.sourceIndex > b.sourceIndex && 2) :
      1);
}

</body>
</html>

```

Клик на элементе выделяет только его. `ctrl ( Cmd )` + Клик добавляет/убирает элемент из выделенных. `Shift` + Клик добавляет промежуток от последнего кликнутого к выделению.

P.S. В этой задаче можно считать, что в элементах списка может быть только текст, без вложенных тегов.

P.P.S. Обработка одновременного нажатия `ctrl ( Cmd )` и `Shift` может быть любой.

## Решение

### Решение, шаг 1

Выделение одного элемента (<http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-1.html>):

```

<script>

var ul = document.getElementsByTagName('ul')[0];

// --- обработчики ---

ul.onclick = function(e) {
  e = e || event;
  var target = e.target || event.srcElement;

  // возможно, клик был внутри списка UL, но вне элементов LI
  if (target.tagName != "LI") return;

  selectSingle(target);
}

```

```

ul.onselectstart = ul.onmousedown = function() {
    return false;
};

// --- функции для выделения ---

function selectSingle(li) {
    deselectAll();
    li.className = 'selected';
}

// снятие выделения со всех - перебором
// это медленнее, чем запоминать список текущих выделенных,
// но подойдет для не очень больших (до 1000 элементов) списков
function deselectAll() {
    for(var i=0; i<ul.children.length; i++) {
        ul.children[i].className = '';
    }
}

</script>

```

Обратите внимание, так как мы поддерживаем выделение самостоятельно, то браузерное выделение отключено.

Кроме того, в реальном коде лучше добавлять/удалять классы более точными методами, а не прямым присвоением `className`.

## Решение, шаг 2

Выделение с `ctrl` (<http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-2.html>):

```

<script>

var ul = document.getElementsByTagName('ul')[0];

// --- обработчики ---

ul.onclick = function(e) {
    e = e || event;
    var target = e.target || event.srcElement;

    // возможно, клик был внутри списка UL, но вне элементов LI
    if (target.tagName != "LI") return;

    var isMac = navigator.platform.indexOf("Mac") != -1;
    if(isMac ? e.metaKey : e.ctrlKey) { // для Mac проверяем Cmd, т.к. Ctrl + click там контекстное меню
        toggleSelect(target);
    } else {
        selectSingle(target);
    }
}

ul.onselectstart = ul.onmousedown = function() {
    return false;
};

// --- функции для выделения ---

function toggleSelect(li) {
    li.className = (li.className == '') ? 'selected' : '';
}

function deselectAll() {
    for(var i=0; i<ul.children.length; i++) {
        ul.children[i].className = '';
    }
}

function selectSingle(li) {
    deselectAll();
    li.className = 'selected';
}

</script>

```

### Решение, шаг 3

Выделение с Shift (<http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-3.html>):

```
<script>

var ul = document.getElementsByTagName('ul')[0];

var lastClickedLi = null;

// --- обработчики ---

ul.onclick = function(e) {
  e = e || event;
  var target = e.target || event.srcElement;

  // возможно, клик был внутри списка UL, но вне элементов LI
  if (target.tagName != "LI") return;

  var isMac = navigator.platform.indexOf("Mac") != -1;
  if(isMac ? e.metaKey : e.ctrlKey) { // для Mac проверяем Cmd, т.к. Ctrl + click там контекстное меню
    toggleSelect(target);
  } else if (e.shiftKey) {
    selectFromLast(target);
  } else {
    selectSingle(target);
  }

  lastClickedLi = target;
}

ul.onselectstart = ul.onmousedown = function() {
  return false;
}

// --- функции для выделения ---

function toggleSelect(li) {
  li.className = (li.className == '') ? 'selected' : '';
}

function selectFromLast(target) {
  var startElem = lastClickedLi || ul.children[0];

  var isLastClickedBefore = compareDocumentPosition(startElem, target) & 4;

  if (isLastClickedBefore) {
    for(var elem = startElem; elem != target; elem = elem.nextSibling) {
      elem.className = 'selected';
    }
  } else {
    for(var elem = startElem; elem != target; elem = elem.previousSibling) {
      elem.className = 'selected';
    }
  }
  elem.className = 'selected';
}

function deselectAll() {
  for(var i=0; i<ul.children.length; i++) {
    ul.children[i].className = '';
  }
}

function selectSingle(li) {
  deselectAll();
  li.className = 'selected';
}

// --- вспомогательная функция ---
// http://learn.javascript.ru/compare-document-position
function compareDocumentPosition(a, b) {
  return a.compareDocumentPosition ?
    a.compareDocumentPosition(b) :
    (a != b && a.contains(b) && 16) +
    (a != b && b.contains(a) && 8) +
```

```
(a.sourceIndex >= 0 && b.sourceIndex >= 0 ?  
  (a.sourceIndex < b.sourceIndex && 4) +  
  (a.sourceIndex > b.sourceIndex && 2) :  
  1);  
}  
  
</script>
```

# DOM events

---

<http://learn.javascript.ru/event-delegation>

# Делегирование событий

Источник: <http://learn.javascript.ru/event-delegation>

Если у вас есть много элементов, события на которых нужно обрабатывать похожим образом, то не стоит присваивать отдельный обработчик каждому.

Вместо этого, назначьте один обработчик общему родителю. Из него можно получить целевой элемент `event.target`, понять на каком потомке произошло событие и обработать его.

Эта техника называется **делегированием** и очень активно применяется в современном JavaScript.

## На примере меню

Делегирование событий позволяет удобно организовывать деревья и вложенные меню.

Давайте для начала обсудим одноуровневое меню:

```
<ul id="menu">
  <li><a href="/php">PHP</a></li>
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/flash">Flash</a></li>
</ul>
```

Клики по пунктам меню будем обрабатывать при помощи JavaScript. Пунктов меню в примере всего несколько, но может быть и много. Конечно, можно назначить каждому пункту свой персональный onclick-обработчик, но что если пунктов 50, 100, или больше? Неужели нужно создавать столько обработчиков? Конечно же, нет!

**Применим делегирование: назначим один обработчик для всего меню, а в нём уже разберёмся, где именно был клик и обработаем его:**

Алгоритм:

1. Вешаем обработчик на внешний элемент (меню).
2. В обработчике: получаем `event.target`.
3. В обработчике: смотрим, где именно был клик и обрабатываем его. Возможно и такое, что данный клик нас не интересует, например если он был на пустом месте.

Код:

```
// 1. вешаем обработчик
document.getElementById('menu').onclick = function(e) {

  // 2. получаем event.target
  var event = e || window.event;
  var target = event.target || event.srcElement;

  // 3. проверим, интересует ли нас этот клик?
  // если клик был не на ссылке, то нет
  if (target.tagName !== 'A') return;

  // обработать клик по ссылке
  var href = target.getAttribute('href');
  alert(href); // в данном примере просто выводим
  return false;
};
```

## Более короткое кросс-браузерное получение target

В примере выше можно было бы получить `target` при помощи логических операторов, вот так:

```
document.getElementById('menu').onclick = function(e) {
    var target = e && e.target || event.srcElement;
    ...
};
```

Работать этот код будет следующим образом:

1. Если это не IE<9, то есть первый аргумент `e` и свойство `e.target`. При этом сработает левая часть оператора ИЛИ `||`.
2. Если это старый IE, то первого аргумента нет, левая часть сразу становится `false` и вычисляется правая часть ИЛИ `||`, которая в этом случае и является аналогом свойства `target`.

Полный код примера: <http://learn.javascript.ru/play/tutorial/browser/events/delegation/menu/index.html>

## Пример со вложенным меню

Обычное меню при использовании делегирования легко и непринуждённо превращается во вложенное.

У вложенного меню остается похожая семантическая структура:

```
<ul id="menu">
<li><a href="/php">PHP</a>
  <ul>
    <li><a href="/php/manual">Справочник</a></li>
    <li><a href="/php/snippets">Сниппеты</a></li>
  </ul>
</li>
<li><a href="/html">HTML</a>
  <ul>
    <li><a href="/html/information">Информация</a></li>
    <li><a href="/html/examples">Примеры</a></li>
  </ul>
</li>
</ul>
```

С помощью CSS можно организовать скрытие вложенного списка `ul` до того момента, пока соответствующий `li` не наведут курсор. Такое скрытие-появление элементов можно реализовать и при помощи JavaScript, но если что-то можно сделать в CSS — лучше использовать CSS.

Пример: <http://learn.javascript.ru/play/tutorial/browser/events/delegation/menu-nested/index.html>

**Делегирование позволяет перейти от обычного меню к вложенному без добавления новых обработчиков.**

Можно добавлять новые пункты меню или удалять ненужные. Так как применено делегирование, то обработчик подхватит новые элементы автоматически.

## Пример «Ба Гуа»

Теперь рассмотрим более сложный пример — диаграмму «Ба Гуа». Это таблица, отражающая древнюю китайскую философию.

Вот она:



## Bagua Chart: Direction, Element, Color, Meaning

<b>Northwest</b> Metal Silver Elders	<b>North</b> Water Blue Change	<b>Northeast</b> Earth Yellow Direction
<b>West</b> Metal Gold Youth	<b>Center</b> All Purple Harmony	<b>East</b> Wood Blue Future
<b>Southwest</b> Earth Brown Tranquility	<b>South</b> Fire Orange Fame	<b>Southeast</b> Wood Green Romance

Её HTML (схематично):

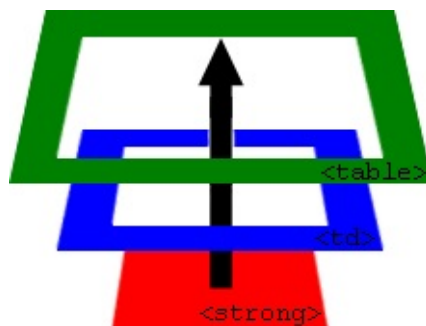
```
<table>
  <tr>
    <td>...<strong>Northwest</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>...еще 2 строки такого же вида...</tr>
  <tr>...еще 2 строки такого же вида...</tr>
</table>
```

Пример: <http://learn.javascript.ru/play/tutorial/browser/events/delegation/bagua/index.html>

В этом примере важно то, как реализована подсветка элементов — через делегирование.

Вместо того, чтобы назначать обработчик для каждой ячейки, назначен один обработчик для всей таблицы. Он использует `event.target`, чтобы получить элемент, на котором произошло событие, и подсветить его.

Обратим внимание: клик может произойти на вложенном теге, внутри `td`. Например, на теге `<strong>`. А затем он всплывает вверх:



В ячейках таблицы могут появиться и другие элементы. Это означает, что нельзя просто проверить `target.tagName`.

Для того, чтобы найти `TD`, на котором был клик, нам нужно пройти вверх по цепочке родителей от `target`. Если в процессе этого мы дойдём до `td`, то это означает, что клик был внутри этой ячейки.

Код:

```
table.onclick = function(event) {
    event = event || window.event;
    var target = event.target || event.srcElement; // (1)

    while(target != this) { // (2)
        if (target.tagName == 'TD') { // (3)
            toggleHighlight(target);
            break;
        }
        target = target.parentNode;
    }
};

function highlight(node) {
    if (highlightedCell) {
        highlightedCell.style.backgroundColor = '';
    }
    highlightedCell = node;
    node.style.backgroundColor = 'red';
}
```

В этом коде делается следующее:

1. Мы кросс-браузерно получаем самый глубокий вложенный элемент, на котором произошло событие. Это `event.target` (в IE<9: `event.srcElement`). Это может быть `strong` или `td`. А возможно и такое, что клик попал в область между ячейками (если у таблицы задано расстояние между ячейками `cellspacing`). В этом случае целевым элементом будет `tr` или даже `table`.
2. В цикле (2) мы поднимаемся вверх по цепочке родителей, пока не дойдем до таблицы. Значение `this` в обработчике — это элемент, на котором сработал обработчик, то есть сама таблица, так что проверка `target != this` проверяет, дошли ли мы до неё. Так как сам обработчик стоит на таблице, то рано или поздно мы должны к ней прийти.
3. Если, поднимаясь вверх, мы дошли до `td` — стоп, эта та ячейка, внутри которой произошел клик. Она-то нам и нужна. Подсветим её и завершим цикл.

В том случае, если клик был вне `td`, цикл `while` просто дойдет до таблицы (рано или поздно, будет `target == this`) и прекратится.

А теперь представьте себе, что в таблице не 9, а 1000 или 10.000 ячеек. **Делегирование позволяет обойтись всего одним обработчиком для любого количества ячеек.**

# Задачи

## Задача 1

Дан список сообщений. Добавьте каждому сообщению кнопку для его удаления.

Используйте делегирование событий. Один обработчик для всего.

В результате, при нажатии на крестик, сообщение удаляется(скрывается).

### Лошадь



Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади, вымершей в дикой природе, за исключением небольшой популяции лошади Пржевальского.

### Осёл




Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства и культуры человека. Все одомашненные ослы относятся к африканским ослам.

### Корова, а также пара слов о диком быке, о волах и о тёлках.



Коро́ва — самка домашнего быка, одомашненного подвида дикого быка, парнокопытного жвачного животного семейства полорогих. Самцы вида называются быками, молодняк — телятами, кастрированные самцы — волами. Молодых (до первой стельности) самок называют тёлками.

Изображение кнопки: 

Исходный код <http://learn.javascript.ru/play/tutorial/browser/events/messages-delegate-src/index.html> или:

```
<!DOCTYPE HTML>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="messages.css">
  <meta charset="utf-8">
  <style>
    body {
      margin: 10px auto;
      width: 470px;
    }
    h3 {
```

```

        margin: 0;
        padding-bottom: .3em;
        padding-right: 20px;
        font-size: 1.1em;
    }
    p {
        margin: 0;
        padding: 0 0 .5em;
    }
    .pane {
        background: #edf5e1;
        padding: 10px 20px 10px;
        border-top: solid 2px #c4df9b;
        position: relative;
    }

    .remove-button {
        position: absolute;
        top: 10px;
        right: 10px;
        cursor: pointer;
        display: block;
        background: url(delete.gif) no-repeat;
        width: 16px;
        height: 16px;
    }
</style>
</head>

<body>

<div id="messages-container">
  <div class="pane">
    <h3>Лошадь</h3>
    <p>Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади.
    <span class="remove-button"></span>
  </div>
  <div class="pane">
    <h3>Осёл</h3>
    <p>Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства человека.
    <span class="remove-button"></span>
  </div>
  <div class="pane">
    <h3>Корова, а также пара слов о диком быке, о волах и о тёлках. </h3>
    <p>Корова — самка домашнего быка, одомашненного подвид дикого быка, парнокопытного жвачного животного семейства Bovidae.
    <span class="remove-button"></span>
  </div>
</div>

<script>

  /* ваш код */

</script>
</body>
</html>

```

## Решение:

Шаг 1:

Поставьте обработчик `click` на контейнере. Он должен проверять, произошел ли клик на кнопке удаления ( `target` ), и если да, то удалять соответствующий ей DIV.

Шаг 2:

```

document.getElementById('messages-container').onclick = function(e) {

    e = e || window.event;
    var target = e.target || e.srcElement;

    // без цикла, т.к. мы точно знаем, что внутри нет тегов
    if (target.className != 'remove-button') return;

```

```
target.parentNode.style.display = 'none';
}
```

Свой вариант:

```
document.querySelector('#messages-container').addEventListener('click', function(e){
    var event = e || window.event;
    var target = event.target || event.srcElement;
    if (target.className !== 'remove-button') return;
    target.parentNode.style.display = 'none';
}, false);
```

## Задача 2:

Создайте дерево, которое по клику на заголовок скрывает-показывает детей:

- Животные
  - Млекопитающие
    - Коровы
    - Ослы
    - Собаки
    - Тигры
  - Другие
    - Змеи
    - Птицы
    - Ящерицы
- Рыбы
  - Аквариумные
    - Гуппи
    - Скалярии
  - Морские
    - Морская форель

Исходный документ <http://learn.javascript.ru/play/tutorial/browser/events/tree-src/index.html> или:

```
<!DOCTYPE HTML>
<html>
<head><meta charset="utf-8"></head>
<body>

<ul class="tree">
  <li>Животные
    <ul>
      <li>Млекопитающие
        <ul>
          <li>Коровы</li>
          <li>Ослы</li>
          <li>Собаки</li>
          <li>Тигры</li>
        </ul>
      </li>
      <li>Другие
        <ul>
          <li>Змеи</li>
          <li>Птицы</li>
          <li>Ящерицы</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

```

</li>
<li>Рыбы
  <ul>
    <li>Аквариумные
      <ul>
        <li>Гуппи</li>
        <li>Скалярии</li>
      </ul>
    </li>
    <li>Морские
      <ul>
        <li>Морская форель</li>
      </ul>
    </li>
  </ul>
</li>
</ul>

</body>
</html>

```

Требования:

- Использовать делегирование.
- Клик вне текста заголовка (на пустом месте) ничего делать не должен.
- При наведении на заголовок — он становится жирным, реализовать через CSS.
- При двойном клике на заголовке, его текст не должен выделяться.

P.S. При необходимости HTML/CSS дерева можно изменить.

## Решение

Дерево устроено как вложенный список.

Клики на все элементы можно поймать, повесив единый обработчик `onclick` на внешний `ul`.

Как поймать клик на заголовке? Элемент `li` является блочным, поэтому нельзя понять, был ли клик на тексте, или справа от него.

Проблема в верстке, в том что `li` занимает всю ширину. Можно кликнуть справа от текста, это все еще `li`.

Один из способов это поправить — обернуть заголовки в дополнительный элемент `span`, и обрабатывать только клики внутри `span` 'ов.

Мы могли бы это сделать в HTML, но давайте для примера используем JavaScript. Следующий код ищет все `li` и оборачивает текстовые узлы в `span`.

```

var treeUl = document.getElementsByTagName('ul')[0];

var treeLis = treeUl.getElementsByTagName('li');

for(var i = 0; i < treeLis.length; i++) {
  var li = treeLis[i];

  var span = document.createElement('span');
  li.insertBefore(span, li.firstChild); // добавить пустой SPAN
  span.appendChild(span.nextSibling); // переместить в него заголовок
}

```

Теперь можно отслеживать клики на заголовках.

Так как `SPAN` — инлайновый элемент, он всегда такого же размера как текст.

