

# Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

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

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

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме [javascript.ru/forum](http://javascript.ru/forum).

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу [iliakan@javascript.ru](mailto:iliakan@javascript.ru) .

## Глава: События, взаимодействие с посетителем

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

---

## Содержание

### События: основы

- Введение в браузерные события

- Назначение обработчиков событий

- Использование атрибута HTML

- Использование свойства DOM-объекта

- Частые ошибки

- Доступ к элементу, this

- Недостатки назначения через опсобытие

- Специальные методы

- Методы IE<9

- Назначение обработчиков по стандарту

- Особенности специальных методов

- Кроссбраузерный способ назначения обработчиков

- Итого

- Получение объекта события

- Получение объекта события

- Для IE8-

- Кроссбраузерное решение

- Итого

- Всплытие и перехват

- Всплытие

- Текущий элемент, this

- Целевой элемент, event.target

- Прекращения всплытия

- Три стадии прохода событий

- Итого

- Действия браузера по умолчанию

- Отмена действия браузера

- Действия, которые нельзя отменить

- Итого
- Отмена выделения, невыделяемые элементы
  - Предотвращение выделения при клике
  - Снятие выделения пост-фактум
  - Свойство user-select
  - Атрибут unselectable="on"
  - Снятие выделения
- Итого

## Делегирование событий

- На примере меню
- Пример со вложенным меню
- Пример «Ба Гуа»
- Применение делегирования: действия в разметке
- Вспомогательная функция delegate
- Итого

## Шаблон проектирования "поведение" (behavior)

- Описание
- Пример
- Область применения

## Управление порядком обработки, setTimeout(...0)

- Главный цикл браузера
- Асинхронные события
- Вложенные (синхронные) события
  - Пример: событие onfocus
- Использование setTimeout(..., 0)
  - Делаем события асинхронными через setTimeout(...,0)
  - Позволить родителю обработать событие
  - Позволить действию браузера завершиться

- Итого

## События мыши

### Введение: клики, кнопка, координаты

- Типы событий мыши
  - Простые события
  - Комплексные события
  - Порядок срабатывания событий
- Получение информации о кнопке: which/button
  - Стандартные свойства
  - Свойство button для IE<9
- Правый клик: oncontextmenu
- Модификаторы shift, alt, ctrl и meta
- Координаты мыши
  - Относительно окна: clientX/Y
  - Относительно документа: pageX/Y
    - Обходной путь для IE<9
  - Демо получения координат мыши

- Итого

## События движения: "mouseover/out/move/leave/enter"

- События mouseover/mouseout, свойство relatedTarget
- Частота событий mousemove и mouseover
  - Тест для mousemove/over/out при быстром движении
- «Лишний» mouseout при уходе на потомка
- События mouseenter и mouseleave.
- Превращение mouseover/out в mouseenter/leave
- Итого

- Колёсико мыши: "wheel" и аналоги
- Отличия колёсика от прокрутки
- Зоопарк wheel в разных браузерах
- Устранение IE-несовместимостей: "fixEvent"

## Основы Drag'n'Drop

- HTML5 Drag'n'Drop
- Основная логика Drag'n'Drop
- Отмена переноса браузера
- mousemove — на document
- Правильное позиционирование
- Итого

## Drag'n'Drop объектов

- Документ
- Начало переноса
- Перенос элемента
  - Визуальное перемещение аватара
  - Полный код mousemove
  - Окончание переноса
  - Определяем элемент под курсором
- Сводим части фреймворка вместе
  - dragManager
  - Реализация переноса иконок
- Варианты расширения этого кода
  - Захватывать элемент можно только за «ручку»
  - Не всегда можно положить на draggable
    - Проверка прав
  - Индикация переноса
  - Анимация отмены переноса
- Итого

## События клавиатуры

- Тестовый стенд
- Виды и свойства событий
  - keydown и keyup
  - Какими бывают скан-коды?
  - keypress
  - Получение символа в keypress
- Отмена пользовательского ввода
  - Отмена специальных действий
  - Демо: перевод символа в верхний регистр
- Порядок наступления событий
- Автоповтор
- Итого, рецепты

## Формы: свойства элементов

- label
- input, textarea
- input type="checkbox", input type="radio"
- select, option
- Практика

## Формы: события "change", "input", "propertychange"

- Событие change
- Событие propertychange
- Событие input
- События cut, copy, paste
- Пример: поле для СМС
- Итого

## Формы: метод и событие "submit"

- Событие submit
- Метод submit

## События и методы "focus/blur"

- Пример использования focus/blur
- HTML5 и CSS3 вместо focus/blur
  - Подсветка при фокусировке
  - Автофокус
  - Плейсхолдер
- Метод elem.focus()
- Разрешаем фокус на любом элементе: tabIndex
- Событие blur и метод elem.blur()
- Делегирование с focus/blur
  - Всплывающие альтернативы: focusin/focusout
- Итого

## Событие "onscroll"

## События "onload", "onbeforeunload" и "onerror"

- Загрузка SCRIPT
  - onload
  - onerror
  - IE<9: onreadystatechange
  - Кросс-браузерное решение
- Обработчики на window
  - window.onerror
  - window.onload
  - window.onunload
  - window.onbeforeunload
- Обработчики на IMG, IFRAME, LINK
- Итого

## Событие загрузки документа "onDOMContentLoaded"

- Тонкости DOMContentLoaded
  - DOMContentLoaded и скрипты
  - DOMContentLoaded и стили
  - Автозаполнение
- IE до 9

## Проверка поддержки браузером

- События
- Элементы и атрибуты
  - Проверка метода или свойства
  - Проверка атрибута
- Итого

## Решения задач

---

## Введение в браузерные события

---

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:



Обратите внимание, внутри `alert` используются *одиночные кавычки*, так как сам атрибут находится в двойных.

Частая ошибка новичков в том, что они забывают, что код находится внутри атрибута. 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()`.

```

01 <!DOCTYPE HTML>
02 <html>
03 <head>
04   <meta charset="utf-8">
05
06   <script>
07     function countRabbits() {
08       for(var i=1; i<=3; i++) {
09         alert("Кролик номер " + i);
10       }
11     }
12   </script>
13 </head>
14 <body>
15   <input type="button" onclick="countRabbits()" value="Считать кроликов!"/>
16 </body>
17 </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":

```

1 <input id="myElement" type="button" value="Нажми меня"/>
2 <script>
3 var elem = document.getElementById('myElement');
4
5 elem.onclick = function() {
6   alert('Спасибо');
7 }
8 </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:

1. Только HTML:

```
1 <input type="button" onclick="alert('Клик!')" value="Кнопка"/>
```

2. HTML + JS:

```

1 <input type="button" id="button" value="Кнопка"/>
2 <script>
3   document.getElementById('button').onclick = function() {
4     alert('Клик!');
5   }
6 </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:

```

1 <input type="button" onclick="alert('До')" value="Нажми меня"/>
2 <script>
3   var elem = document.getElementsByTagName('input')[0];
4
5   elem.onclick = function() { // перезапишет существующий обработчик
6     alert('После');
7   }
8 </script>

```

Обработчиком можно назначить уже существующую функцию:

```

1 function sayThanks() {
2   alert('Спасибо!');
3 }
4
5 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:

```

1 document.getElementById('button').onclick = function() {
2   sayThanks(); // содержимое атрибута
3 }

```

➡ Используйте свойство, а не атрибут. Так неверно: `elem.setAttribute('onclick', func)`.

Хотя, с другой стороны, если `func` — строка, то такое присвоение будет успешным, например:

```

1 // работает, будет при клике выдавать 1
2 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:

```
1 // при нажатии на body будут ошибки
2 document.body.setAttribute('onclick', function() { alert(1) });
```

Как вы думаете, почему?

Значением атрибута может быть только строка. Любое другое значение преобразуется в строку.

Функция в строчном виде обычно даёт свой код: `"function() { alert(1) }"`.

Итак, атрибут присвоен. Теперь браузер создаст обработчик с телом из этой строки:

```
document.body.onclick = function() {
  function() { alert(1) }
}
```

Этот код попросту некорректен с точки зрения JavaScript. Локальная функция объявлена без имени — отсюда и ошибка.

#### ➡ 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:

```
1 var input = document.getElementById('button')  
2 function handler() {  
3     alert('спасибо!')  
4 }  
5 input.attachEvent( "onclick" , handler) // Назначение обработчика  
6 // ....  
7 input.detachEvent( "onclick", handler) // Удаление обработчика
```



#### Удаление требует ту же функцию

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:

```
1 input.attachEvent( "onclick" ,  
2     function() {alert('Спасибо!')}  
3 )  
4 // ....  
5 input.detachEvent( "onclick",  
6     function() {alert('Спасибо!')}  
7 )
```

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:

```
01 <input id="myElement" type="button" value="Нажми меня"/>
02
03 <script>
04   var myElement = document.getElementById("myElement")
05   var handler = function() {
06     alert('Спасибо!')
07   }
08
09   var handler2 = function() {
10     alert('Спасибо еще раз!')
11   }
12
13   myElement.attachEvent("onclick", handler); // первый
14   myElement.attachEvent("onclick", handler2); // второй
15 </script>
```



У обработчиков, назначенных с `attachEvent`, нет `this`

Обработчики, назначенные с `attachEvent` не получают `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`:

```
1 function handler() { ... }
2
3 elem.addEventListener( "click" , handler, false) // назначение обработчика
4
5 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)`:

```
01 var addEvent, removeEvent;
02
03 if (document.addEventListener) { // проверка существования метода
04     addEvent = function(elem, type, handler) {
05         elem.addEventListener(type, handler, false);
06     };
07     removeEvent = function(elem, type, handler) {
08         elem.removeEventListener(type, handler, false);
09     };
10 } else {
11     addEvent = function(elem, type, handler) {
12         elem.attachEvent("on" + type, handler);
13     };
14     removeEvent = function(elem, type, handler) {
15         elem.detachEvent("on" + type, handler);
16     };
17 }
18
19 ...
20 // использование:
21 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( on+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`.

Последний аргумент `addEventListener` мы рассмотрим позже, в статье [Всплытие и перехват \[1\]](#). Он почти всегда равен `false`.

## Получение объекта события

---

Чтобы хорошо обработать событие, недостаточно знать о том, что это «клик» или «нажатие клавиши». Могут понадобиться детали: координаты курсора, введённый символ и другие, в зависимости от события.

Эти детали браузер записывает в «объект события», к которому может обратиться обработчик.

Некоторые свойства объекта события для события `onclick`:

- ➡ `event.type` — тип события, равен `click`.
- ➡ `event.target` — элемент, по которому кликнули. В IE<9 вместо него используется свойство `event.srcElement`.
- ➡ `event.clientX` / `event.clientY` — координаты курсора в момент клика, относительно окна.
- ➡ ...Также есть информация о том, какой кнопкой был произведен клик, и другие свойства. Мы разберем их все позже.

## Получение объекта события

Все современные браузеры передают объект события первым аргументом в обработчик.

Для примера:

```
element.onclick = function(event) {  
    alert(event.clientX); // вывести координату клика  
};
```

## Для IE8-

---

IE8- вместо передачи параметра обработчику создаёт глобальный объект `window.event`. Обработчик может обратиться к нему.

Работает это так:

```
1 // обработчик без аргументов  
2 element.onclick = function() {  
3     // window.event - объект события  
4     alert( window.event.clientX );  
5 };
```

## Кроссбраузерное решение

Универсальное решение для получения объекта события:

```
1 element.onclick = function(event) {  
2     event = event || window.event; // (*)  
3  
4     // Теперь event - объект события во всех браузерах.  
5 };
```

Здесь все браузеры передадут аргумент-событие `event`, а IE создаст `window.event`, соответственно в строчке (\*) будет использовано то, что есть.

**При назначении обработчика в HTML, можно использовать переменную `event`, это будет кросс-браузерно:**

Вот так будет работать во всех браузерах:

```
<input type="button" onclick="alert(event.type)" value="Тип события">
```



Это возможно потому, что браузеры сами создают обработчик с указанным телом. IE: `function() { alert(event.type) }` — переменная `event` будет взята глобальная, а остальные — создадут обработчик с первым аргументом: `function(event) { alert(event.type) }`.

## Итого

- ➡ Объект события содержит ценную информацию о деталях события.
- ➡ Он передается первым аргументом `event` в обработчик для всех браузеров, кроме IE<9, в которых используется глобальная переменная `window.event`.

Кросс-браузерно для JavaScript-обработчика получаем объект события так:

```
1 element.onclick = function(event) {  
2     event = event || window.event;  
3  
4     // Теперь event - объект события во всех браузерах.  
5 };
```

Еще вариант:

```
1 element.onclick = function(e) {  
2     e = e || event; // если нет другой внешней переменной event  
3     ...  
4 };
```

## Всплытие и перехват

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 сработает, 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>
```

*Кликните на EM, сработает обработчик на DIV*

## Всплытие

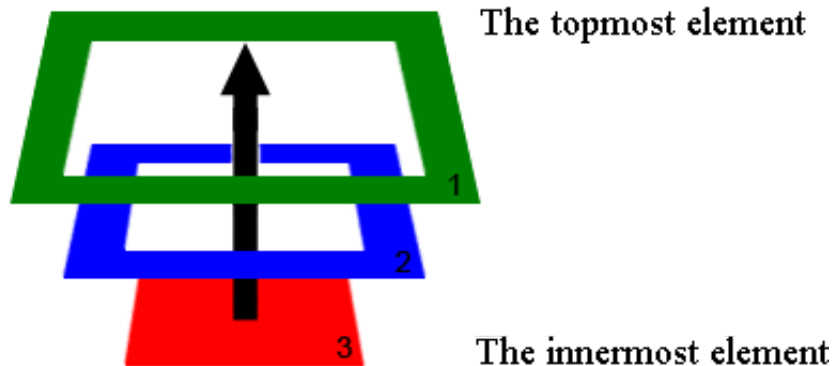
Основной принцип всплытия:

**После того, как событие сработает на самом вложенном элементе, оно также сработает на родителях, вверх по цепочке вложенности.**

Например, есть 3 вложенных блока:

```
01 <!DOCTYPE HTML>  
02 <html>  
03 <body>  
04 <link type="text/css" rel="stylesheet" href="example.css">  
05  
06 <div class="d1">1 <!-- внешний (topmost) -->  
07   <div class="d2">2  
08     <div class="d3">3 <!-- внутренний (innermost) -->  
09   </div>  
10 </div>  
11 </div>  
12  
13 </body>  
14 </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:

```
01 <!DOCTYPE HTML>
02 <html>
03 <body>
04 <link type="text/css" rel="stylesheet" href="example.css">
05
06 <div class="d1" onclick="highlight(this)">1
07   <div class="d2" onclick="highlight(this)">2
08     <div class="d3" onclick="highlight(this)">3
09   </div>
10 </div>
11 </div>
12
13 <script>
14 function highlight(elem) {
15   elem.style.backgroundColor = 'yellow';
16   alert(elem.className);
17   elem.style.backgroundColor = '';
18 }
19 </script>
20
21 </body>
22 </html>
```

Кликните по самому внутреннему DIV'у, чтобы увидеть всплытие:

1

2

3

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 `on` событие: это и кросс-браузерно, и `this` всегда есть.

## Целевой элемент, `event.target`

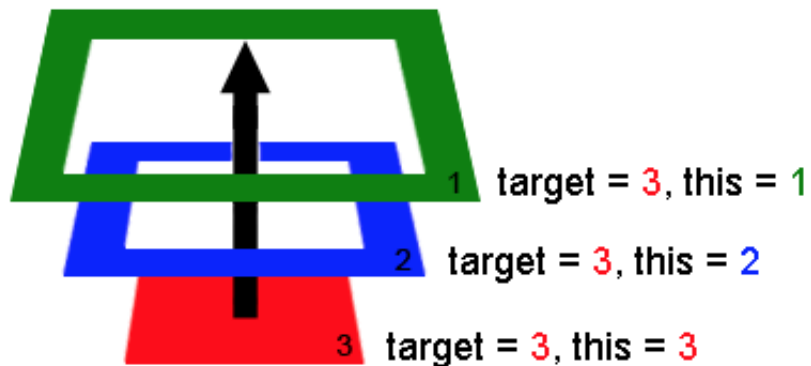
Самый глубокий элемент, который вызывает событие, называется «целевым» или «исходным» элементом.

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.



В примере каждый DIV имеет обработчик `onclick`, который выводит и `target` и `this`.

Кликните по какому-нибудь блоку, например самому вложенному:

Обратите внимание:

→ `event.target` не изменяется по мере всплытия события,

→ ..а вот `this` изменяется и подсвечивается.



- 1
- 2
- 3

```
01 <!DOCTYPE HTML>
02 <html>
03 <body>
04 <link type="text/css" rel="stylesheet" href="example.css">
05
06 <div class="d1">1
07   <div class="d2">2
08     <div class="d3">3
09     </div>
10   </div>
11 </div>
12
13 <script>
14 var divs = document.getElementsByTagName('div');
15
16 for(var i=0; i<divs.length; i++) {
17   divs[i].onclick = function(e) {
18     e = e || event
19     var target = e.target || e.srcElement
20
21     this.style.backgroundColor = 'yellow';
22
23     alert("target = " + target.className + ", this=" + this.className);
24
25     this.style.backgroundColor = '';
26   }
27 }
28 </script>
29
30 </body>
31 </html>
```

## Прекращения всплытия

---

Всплытие идет прямо наверх. Обычно оно будет всплывать до <HTML>, 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:

```
01 element.onclick = function(event) {  
02     event = event || window.event; // Кроссбраузерно получить событие  
03  
04     if (event.stopPropagation) { // существует ли метод?  
05         // Стандартно:  
06         event.stopPropagation();  
07     } else {  
08         // Вариант IE  
09         event.cancelBubble = true;  
10     }  
11 }
```

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.



**Не прекращайте всплытие без необходимости!**

Всплытие — это удобно. Не прекращайте его без явной нужды, очевидной и архитектурно прозрачной.

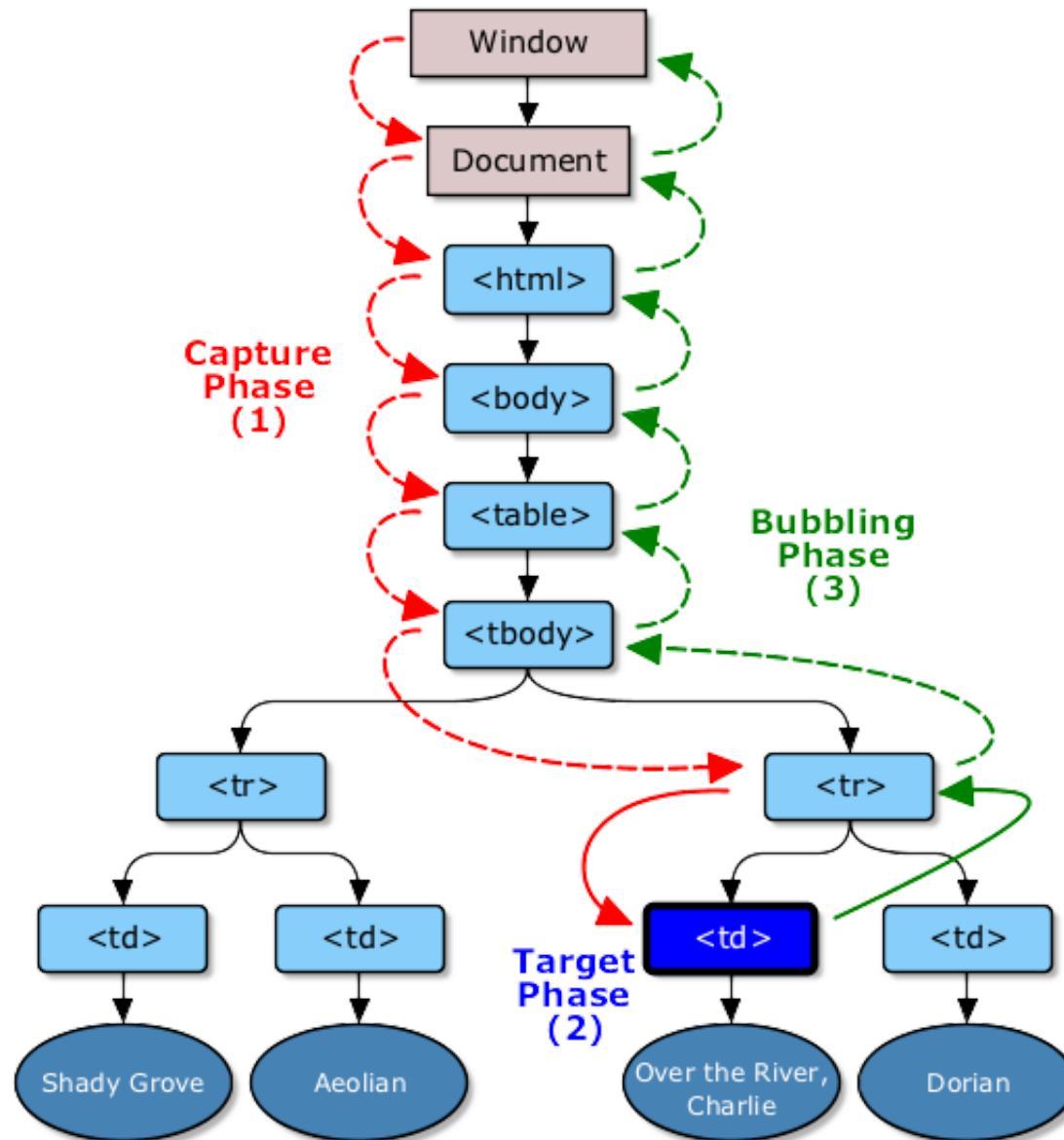
Зачастую прекращение всплытия создаёт свои подводные камни, которые потом приходится обходить.

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.

1. Событие сначала идет сверху вниз. Эта стадия называется «стадия перехвата» (capturing stage).
2. Событие достигло целевого элемента. Это — «стадия цели» (target stage).
3. 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.

Кликните на div ниже, чтобы увидеть захват в действии (не работает в IE<9):

1  
2  
3

Должно быть 1 -> 2 -> 3.

```
01 <!DOCTYPE HTML>
02 <html>
03 <body>
04 <link type="text/css" rel="stylesheet" href="example.css">
05
06 <div class="d1">1
07   <div class="d2">2
08     <div class="d3">3
09     </div>
10   </div>
11 </div>
12
13 <script>
14 var divs = document.getElementsByTagName('div');
15
16 // на каждый DIV повесить обработчик на стадии захвата
17 for(var i=0; i<divs.length; i++) {
18   divs[i].addEventListener("click", highlightThis, true);
19 }
20
21 function highlightThis() {
22   this.style.backgroundColor = 'yellow';
23   alert(this.className);
24   this.style.backgroundColor = '';
25 }
26 </script>
27
28 </body>
29 </html>
```

Теперь давайте назначим обработчики для обеих стадий.

Кликните по самому внутреннему div, чтобы увидеть порядок исполнения события (не работает IE<9):

1  
2  
3

Должно быть 1 -> 2 -> 3 -> 2 -> 1.

JavaScript-код примера:

```
1 var divs = document.getElementsByTagName('div');  
2  
3 for (var i=0; i<divs.length; i++) {  
4   divs[i].addEventListener("click", highlightThis, true);  
5   divs[i].addEventListener("click", highlightThis, false);  
6 }
```

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`.



Есть события, которые не всплывают, но которые можно захватить

На практике, стадия захвата редко используется. Но есть события, которые можно только захватить.

Например, таково [событие фокусировки на элементе \[2\]](#).

## 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

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.

1. 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:

```
1 element.onclick = function(event) {
2   event = event || window.event
3
4   if (event.preventDefault) { // если метод существует
5     event.preventDefault();
6   } else { // вариант IE<9:
7     event.returnValue = false;
8   }
9 }
```

Можно записать в одну строку:

```
..
event.preventDefault ? event.preventDefault() : (event.returnValue=false);
...
```

2. Если обработчик назначен через `on...`, то `return false` из обработчика отменяет действие браузера:

```
1 element.onclick = function(event) {
2   ...
3   return false;
4 }
```

Такой способ проще, но не будет работать, если обработчик назначен через `addEventListener/attachEvent`.

В следующем примере при клике по ссылке переход не произойдет, потому что действие браузера отключено:

```
<a href="/" onclick="return false">Нажми меня</a>
```

Нажми меня



Возвращать true не нужно

При возврате `return false` из обработчика, назначенного через `onсобытие`, 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.

```
1 <style>
2 a:focus { border: 1px solid black }
3 </style>
4 <a href="/" onclick="return false" onfocus="return false">
5   По клику произойдет фокусировка, но перехода не будет
6 </a>
```

По клику произойдет фокусировка, но перехода не будет



Действие браузера по умолчанию и всплытие взаимно независимы

**Отмена действия браузера не остановит всплытие и наоборот.**

Если хотите отменить и то и другое:

```
1 function stop(e) {
2   if (e.preventDefault) { // стандарт
3     e.preventDefault();
4     e.stopPropagation();
5   } else { // IE8-
6     e.returnValue = false;
7     e.cancelBubble = true;
8   }
9 }
```

## 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 `onclick`.
- ➡ Действие браузера само по себе, всплытие события — само по себе. Они никак не связаны.

# Отмена выделения, невыделяемые элементы

В этой главе мы рассмотрим основные способы, как делать элемент невыделяемым.

Это бывает нужно в элементах управления, чтобы выделение браузера не мешало посетителю.

## Предотвращение выделения при клике

Браузер выделяет текст при движении мышью с зажатой левой кнопкой, а также при двойном клике на элемент.

Зачастую, такое выделение не нужно, и его можно отменить.

Чтобы понять проблему — рассмотрим пример.

Например, мы хотим обработать двойной клик. Чтобы понять, что не так — попробуйте сделать двойной клик на элементе ниже.

```
<span ondblclick="alert('двойной клик!')">Текст</span>
```

Текст

Обработчик сработает. Но побочным эффектом является *выделение текста браузером*. Обычно это совсем не нужно.

Выделение появляется как действие браузера «по умолчанию» в ответ на событие мыши.

**Чтобы избежать выделения, мы должны предотвратить действие браузера по умолчанию для события `selectstart` [3] в IE и `mousedown` в других браузерах.**

Полный код элемента, который обрабатывает двойной клик без выделения:

```
<div ondblclick="alert('Тест')" onselectstart="return false" onmousedown="return false">  
  Двойной клик сюда выведет "Тест", без выделения  
</div>
```

Двойной клик сюда выведет "Тест", без выделения

При установке на родителя — все его потомки становятся невыделяемыми:



```

1  Элементы списка не выделяются при клике:
2  <ul onmousedown="return false" onselectstart="return false">
3    <li>Винни-Пух</li>
4    <li>Ослик Иа</li>
5    <li>Мудрая Сова</li>
6    <li>Кролик. Просто кролик.</li>
7  </ul>

```

Элементы списка не выделяются при клике:

- Винни-Пух
- Ослик Иа
- Мудрая Сова
- Кролик. Просто кролик.

Это работает благодаря всплытию событий.

**Этот способ не делает элемент полностью невыделяемым.** Если пользователь хочет выделить текстовое содержимое элемента, то он может сделать это. Достаточно начать выделение (зажать кнопку мыши) не на самом элементе, а рядом с ним.

## Снятие выделения пост-фактум

Вместо предотвращения выделения, можно его снять в обработчике события.

Например, попробуйте двойной клик на первый и второй элементы списка:

```

01 <ul>
02   <li ondblclick="alert(1);">Выделяется при двойном клике</li>
03   <li ondblclick="alert(2);clearSelection()">Выделение отменяется при двойном клике.</li>
04 </ul>
05
06 <script>
07   function clearSelection() {
08     if (window.getSelection()) {
09       window.getSelection().removeAllRanges();
10     } else { // старый IE
11       document.selection.empty();
12     }
13   }
14 </script>

```

- Выделяется при двойном клике
- Выделение отменяется при двойном клике.

Этот способ использует методы работы с выделением, описанные в статье [Выделение: Range, TextRange и Selection \[4\]](#).

Его стоит держать в уме, он подходит в большом количестве случаев. Например, если двойной клик используется как переключатель

элемента в редактируемое состояние — путём показа виджета или замены его на INPUT.

## Свойство user-select

Существует нестандартное CSS-свойство user-select, которые делает элемент невыделяемым. Оно планировалось в стандарте CSS3, но от него отказались.

Это свойство работает (с префиксом) в Firefox, Chrome/Safari, IE10+:

```
01 <style>
02 b {
03   -webkit-user-select: none; /* user-select -- это нестандартное свойство */
04   -moz-user-select: none;    /* поэтому нужны префиксы */
05   -ms-user-select: none;
06   -o-user-select: none; /* не поддерживается, просто на будущее */
07 }
08 </style>
09
10 Строка до..
11 <div onclick="alert('Тест')">
12   <b>Этот текст нельзя выделить в Firefox, Safari/Chrome, IE10</b>
13 </div>
14 .. Строка после
```

Строка до..  
**Этот текст нельзя выделить в Firefox, Safari/Chrome, IE10**  
.. Строка после

Читайте на эту тему также [Controlling Selection with CSS user-select \[5\]](#).

## Атрибут unselectable="on"

В IE, а также Opera эту задачу может решить атрибут unselectable [6] .

Но такая «невыделяемость»:

1. Во-первых, не наследуется. То есть, невыделяемость родителя не делает невыделяемыми детей.
2. Во-вторых, текст можно выделить, если начать выделение не на самом элементе, а рядом с ним.

```
<div onclick="alert('Тест')" unselectable="on" style="border:1px solid black">
  Этот текст невыделяем в IE и Opera, <em>кроме дочерних элементов</em>
</div>
```

В действии:

Этот текст невыделяем в IE и Opera, *кроме дочерних элементов*

Левая часть текста в IE/Opera не выделяется при двойном клике. Правую часть (em) можно выделить, т.к. на ней нет атрибута unselectable.

«Отмена выделения», как правило, не означает, что содержимое элемента вообще нельзя выделить. Обычно это сделать можно,

если нажать мышью где-нибудь «на стороне» и перевести выделение на элемент. Её главный смысл в том, что выделение нельзя *начать* на элементе.

## Снятие выделения

**Можно не делать элемент невыделяемым, а снимать выделение пост-фактум.**

Для работы с выделением в браузере есть специальные методы, описанные в статье [Выделение: Range, TextRange и Selection \[7\]](#). Они различаются в IE<9 и современных браузерах.

Пример ниже снимает выделение в обработчике события `dblclick` («двойной клик»).

```
01 <div ondblclick="clearSelection(); alert('Тест');">
02   Двойной клик сюда снимет выделение и выведет "Тест"
03 </div>
04
05 <script>
06   function clearSelection() {
07     try {
08       window.getSelection().removeAllRanges();
09     } catch(e) {
10       document.selection.empty(); // IE<9
11     }
12   }
13 </script>
```

Двойной клик сюда снимет выделение и выведет "Тест"

У этого подхода есть две особенности:

- ➡ Выделение всё же производится, но тут же снимается. Это не всегда красиво.
- ➡ Выделение при помощи передвижения зажатой мыши всё еще работает, так что посетитель имеет возможность выделить содержимое элемента.

## Итого

Для отмены выделения есть несколько способов:

1. CSS-свойство `user-select` — для Firefox, Safari/Chrome, IE10 (с префиксом).
2. Атрибут `unselectable="on"` — для IE, Opera (не наследуется)>
3. Кросс-браузерный JavaScript:

```
elem.onmousedown = elem.onselectstart = function() {
  return false;
}
```

Если говорить точнее, то выделить «невыделяемые» элементы обычно можно, если начать выделение вне списка, например сверху. Можно, конечно, поиграться с запретом выделения на уровне `<html>`, но всё же основная область применения этих способов — не защита от копирования, а отмена выделения там, где оно явно не нужно и неудобно. Например, при начале Drag'n'Drop-переноса мышь нажимается и движется, но выделять ничего не нужно.

# Делегирование событий

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:

```
1 <ul id="menu">
2   <li><a href="/php">PHP</a></li>
3   <li><a href="/html">HTML</a></li>
4   <li><a href="/javascript">JavaScript</a></li>
5   <li><a href="/flash">Flash</a></li>
6 </ul>
```

Данный пример при помощи CSS может выводиться так:



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:

```

01 // 1. вешаем обработчик
02 document.getElementById('menu').onclick = function(e) {
03
04     // 2. получаем event.target
05     var event = e || window.event;
06     var target = event.target || event.srcElement;
07
08     // 3. проверим, интересует ли нас этот клик?
09     // если клик был не на ссылке, то нет
10     if (target.tagName !== 'A') return;
11
12     // обработать клик по ссылке
13     var href = target.getAttribute('href');
14     alert(href); // в данном примере просто выводим
15     return false;
16 };

```



### Более короткое кросс-браузерное получение target

В примере выше можно было бы получить target при помощи [логических операторов \[8\]](#), вот так:

```

1 document.getElementById('menu').onclick = function(e) {
2     var target = e && e.target || event.srcElement;
3     ...
4 };

```

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:

```

01 <ul id="menu">
02 <li><a href="/php">PHP</a>
03   <ul>
04     <li><a href="/php/manual">Справочник</a></li>
05     <li><a href="/php/snippets">Сниппеты</a></li>
06   </ul>
07 </li>
08 <li><a href="/html">HTML</a>
09   <ul>
10     <li><a href="/html/information">Информация</a></li>
11     <li><a href="/html/examples">Примеры</a></li>
12   </ul>
13 </li>
14 </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 в действии, без изменения JavaScript-кода:

Проведите курсором над меню.



**Делегирование позволило 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» \[9\]](#) . 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

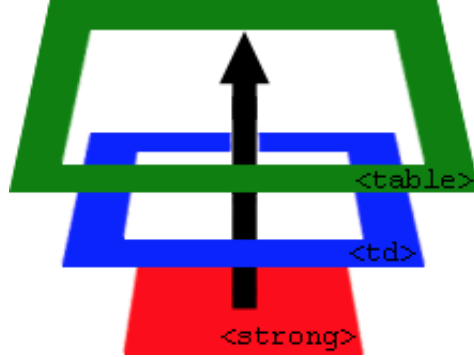
Её HTML (схематично):

```
1 <table>
2   <tr>
3     <td>...<strong>Northwest</strong>...</td>
4     <td>...</td>
5     <td>...</td>
6   </tr>
7   <tr>...еще 2 строки такого же вида...</tr>
8   <tr>...еще 2 строки такого же вида...</tr>
9 </table>
```

**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 (комментарии ниже):

```
01 table.onclick = function(event) {  
02     event = event || window.event;  
03     var target = event.target || event.srcElement; // (1)  
04  
05     while(target != this) { // (2)  
06         if (target.tagName == 'TD') { // (3)  
07             toggleHighlight(target);  
08             break;  
09         }  
10         target = target.parentNode;  
11     }  
12 };
```

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.

## Применение делегирования: действия в разметке

Ячейки таблицы и пункты меню — это примеры использования делегирования для *обработки схожих элементов*. Оно работает хорошо, поскольку действия для них примерно одинаковые.

Но делегирование позволяет использовать обработчик и для абсолютно разных действий.

Например, нам нужно сделать меню с разными кнопками: «Сохранить», «Загрузить», «Поиск» и т.д.

Первое, что может прийти в голову — это найти каждую кнопку и назначить ей свой обработчик.

Но более изящно решить задачу можно путем добавления одного обработчика на всё меню. Все клики внутри меню попадут в обработчик.

Но как нам узнать, какую кнопку нажали и как обработать это событие? Эту задачу мы можем решить, добавив каждой кнопке нужный нам метод в специальный атрибут, который назовем `data-action` (можно придумать любое название, но `data-*` является валидным в HTML5):

```
<button data-action="save">Нажмите, чтобы Сохранить</button>
```

Обработчик считывает содержимое атрибута и выполняет метод. Взгляните на рабочий пример:

```
01 <div id="menu">
02   <button data-action="save">Нажмите, чтобы Сохранить</button>
03   <button data-action="load">Нажмите, чтобы Загрузить</button>
04 </div>
05
06 <script>
07 function Menu(elem) {
08   this.save = function() { alert('сохраняю'); };
09   this.load = function() { alert('загружаю'); };
10
11   var self = this;
12
13   elem.onclick = function(e) {
14     var target = e && e.target || event.srcElement; // (*)
15     var action = target.getAttribute('data-action');
16     if (action) {
17       self[action]();
18     }
19   };
20 }
21
22 new Menu(document.getElementById('menu'));
23 </script>
```

Нажмите, чтобы Сохранить

Нажмите, чтобы Загрузить

Обратите внимание, как используется трюк с `var self = this`, чтобы сохранить ссылку на объект `Menu`. Иначе обработчик просто бы не смог вызвать методы `Menu`, потому что его *собственный* `this` ссылается на элемент.

Что в этом случае нам дает использование делегирования событий?



- ➔ Не нужно писать код, чтобы присвоить обработчик каждой кнопке. Меньше кода, меньше времени, потраченного на инициализацию.
- ➔ Структура HTML становится по-настоящему гибкой. Мы можем добавлять/удалять кнопки в любое время.
- ➔ Данный подход является семантическим. Мы можем использовать классы «`action-save`», «`action-load`» вместо `data-action`. Обработчик найдёт класс `action-*` и вызовет соответствующий метод. Это действительно очень удобно.

Качество кода в примере выше можно повысить, если поменять названия методов объекта с `save`, `load` на `onClickSave`, `onClickLoad`, так как это не просто методы, а методы-обработчики событий. И вызывать их, соответственно, как `self["onClick"+...]()`. Это сделает смысл методов более понятным и упростит чтение и поддержку кода.

## Вспомогательная функция `delegate`

Алгоритм делегирования можно вынести в отдельную функцию, которая будет искать ближайший элемент и вызывать на нём обработчик.

Подобные функции есть во многих фреймворках, но легко можно написать и свою.

## Итого

Делегирование событий — это здорово! Пожалуй, это один из самых полезных приёмов для работы с DOM. Он отлично подходит, если есть много элементов, обработка которых очень схожа.

Алгоритм:

1. Вешаем обработчик на контейнер.
2. В обработчике: получаем `event.target`.
3. В обработчике: если необходимо, проходим вверх цепочку `target.parentNode`, пока не найдем нужный подходящий элемент (и обработаем его), или пока не упрёмся в контейнер (`this`).

Зачем использовать:



- ➔ Упрощает инициализацию и экономит память: не нужно вешать много обработчиков.
- ➔ Меньше кода: при добавлении и удалении элементов не нужно ставить или снимать обработчики.
- ➔ Удобство изменений: можно массово добавлять или удалять элементы путём изменения `innerHTML`.

Конечно, у делегирования событий есть свои ограничения.



- Во-первых, событие должно всплывать. Большинство событий всплывают, но не все.
- Во-вторых, делегирование создает дополнительную нагрузку на браузер, ведь обработчик запускается, когда событие происходит в любом месте контейнера, не обязательно на элементах, которые нам интересны. Но обычно эта нагрузка невелика и не является проблемой.

## Шаблон проектирования "поведение" (behavior)

Шаблон проектирования «поведение» (behavior) позволяет задавать хитрые обработчики на элементе *декларативно*, установкой специальных HTML-атрибутов и классов.

Например, хочется, чтобы при клике на один элемент показывался другой. Конечно, можно поставить обработчик в атрибуте `onclick` или даже описать его где-нибудь в JS-коде страницы.

Но есть решение другое, и очень изящное.

### Описание

Поведение состоит из двух частей.

1. Элементу ставится атрибут, описывающий его поведение.
2. При помощи делегирования ставится обработчик на документ, который ловит все клики и, если элемент имеет нужный атрибут, производит нужное действие.

### Пример

Например, в `data-toggle-id` можно записать ID элемента, который будет скрываться-показываться:

```
1 <button data-toggle-id="subscribe-mail">
2   Кликните для подписки
3 </button>
4
5 <div id="subscribe-mail" style="display:none">
6   Ваша почта: <input type="email">
7 </div>
```

Код для обработки события будет проверять наличие этого атрибута и действовать, если он есть:

```
1 document.onclick = function(e) {
2   var target = e && e.target || window.event.srcElement;
3
4   var dataToggleId = target.getAttribute('data-toggle-id');
5   if (!dataToggleId) return;
6
7   var elem = document.getElementById(dataToggleId);
8   elem.style.display = elem.offsetHeight ? 'none' : 'block';
9 }
```

Результат:

Кликните для подписки

Один раз описав JS-код поведения, его можно использовать везде на странице.

## Область применения

Шаблон «поведение» удобен тем, что сколь угодно сложное JavaScript-поведение можно «навесить» на элемент одним лишь атрибутом. А можно — несколькими атрибутами на связанных элементах.

Еще примеры использования:

➡ Можно показывать подсказку при наведении, если есть атрибут:

```
<div data-tooltip="Подсказка при наведении">  
  Текст  
</div>
```

➡ Можно проверять значение поля формы при отправке, руководствуясь `type` и `data-validator`.

Можно даже сделать так, что отправка формы, при наличии атрибута, будет работать через AJAX.

```
1 <form data-ajax="1">  
2   <input type="email">  
3   <input data-validator="integer positive">  
4 </form>
```

➡ ...В других местах, где хочется задать действие в вёрстке, и это не испортит архитектуру приложения.

## Управление порядком обработки, `setTimeout(...0)`

Здесь мы разберём «продвинутые» приёмы для работы с событиями, которые используют особенности работы `setTimeout` и `setInterval` [10].

## Главный цикл браузера

В каждом окне выполняется только один *главный* поток, который занимается выполнением JavaScript, отрисовкой и работой с DOM.

Он выполняет команды последовательно и блокируется при выводе модальных окон, таких как `alert`.

Есть и другие потоки, например для сетевых коммуникаций, поэтому скачивание файлов может продолжаться пока основной поток ждёт реакции на `alert`. Но это скорее исключение, чем правило.



## Web Workers

Существует спецификация [Web Workers \[11\]](#), которая позволяет запускать дополнительные JavaScript-потоки(workers).

Они могут обмениваться сообщениями с главным потоком, но у них переменные не могут быть разделены между потоками. В каждом потоке — свои переменные, поэтому правильнее было бы назвать их не «потоками», а «процессами».

В частности, дополнительные потоки не имеют доступа к DOM. Они полезны, преимущественно, при вычислениях, в которых можно задействовать несколько процессоров одновременно.

## Асинхронные события

Большинство событий — асинхронны.

**Когда происходит асинхронное событие, оно попадает в очередь.**

Внутри браузера существует главный внутренний цикл, который проверяет очередь и обрабатывает события, запускает соответствующие обработчики и т.п.

**События добавляются в очередь в порядке поступления. Иногда даже сразу пачкой.**

Например, при клике на элементе генерируется несколько событий:

1. Сначала `mousedown` — нажата кнопка мыши.
2. Затем `mouseup` — кнопка мыши отпущена.
3. Так как это было над одним элементом, то дополнительно генерируется `click`

В действии:

```
01 <textarea rows="6" cols="40">Кликни меня
02 </textarea>
03
04 <script>
05 var area = document.getElementsByTagName('textarea')[0];
06
07 area.onmousedown = function() { this.value += 'mousedown\n'; };
08 area.onmouseup = function() { this.value += 'mouseup\n'; };
09 area.onclick = function() { this.value += 'click\n'; };
10 </script>
```

Кликни меня

**Все асинхронные события обрабатываются последовательно.**

Основной поток, когда освобождается, берёт оттуда событие, обрабатывает, затем получает из очереди следующее и так далее. Каждое событие обрабатывается полностью отдельно от других.

В примере выше будут последовательно обработаны `mousedown`, `mouseup` и `click`.

## Вложенные (синхронные) события

**Некоторые события обрабатываются синхронно, то есть поток выполнения переходит в их обработчик тут же.**

Как правило, синхронными будут события, которые инициируются вызовом метода. Исключением здесь является Internet Explorer (включая 10), в котором почти все события асинхронные.

### Пример: событие `onfocus`

Когда посетитель фокусируется на элементе, возникает событие `onfocus`. Но фокусировку можно вызвать и явно, вызовом метода `elem.focus()`.

В главе [События и методы «focus/blur» \[12\]](#) мы познакомимся с этим событием подробнее, а пока — нажмите на кнопку в примере ниже. При этом будет вызван метод `text.focus()`, который установит фокус на текстовом поле `text` и вызовет этим событие `onfocus`.

Вы увидите, что событие `onfocus` — синхронное. Его обработчик `onfocus` не становится в очередь, а выполняется тут же, до завершения `onclick`:

```
01 <input type="button" value="Нажми меня">
02 <input type="text" size="60">
03
04 <script>
05   var button = document.body.children[0];
06   var text = document.body.children[1];
07
08   button.onclick = function() {
09     text.value += ' ->В onclick ';
10
11     text.focus(); // вызов инициирует событие onfocus
12
13     text.value += ' из onclick-> ';
14   };
15
16   text.onfocus = function() {
17     text.value += ' !focus! ';
18   };
19 </script>
```

Исключением является браузер IE. В нём событие `onfocus` — асинхронное, так что будет сначала полностью обработан клик, а потом — фокус. В остальных — фокус вызовется посередине клика. Попробуйте кликнуть в IE и в другом браузере, чтобы увидеть разницу.

## Использование `setTimeout(..., 0)`

Вызов функции, запланированной `setTimeout/setInterval`, когда придёт время, добавляется в ту же самую очередь событий.

В частности, если обернуть код в функцию и запланировать её через `setTimeout(f, 0)`, то она выполнится заведомо после текущего кода и независимо от него. Таким образом, мы можем отложить обработку события.



### Вместо `setTimeout(..., 0) → setImmediate`

В этой секции, чтобы делать примеры проще, для планирования события на «ближайшее время» мы используем `setTimeout(..., 0)`. Это потому, что он встроен во все браузеры.

Вместо него рекомендуется более эффективный `setImmediate`. Этот метод и его кросс-браузерная эмуляция разобраны в главе [setImmediate \[13\]](#). Если вкратце, то `setImmediate(func)` это `setTimeout(func, 0)` без лишней задержки на «тик» таймера.

## Делаем события асинхронными через `setTimeout(..., 0)`

Как правило, обработчики событий вызываются по очереди. Мы предполагаем, что текущий обработчик завершится перед тем, как начнется другой. Вложенные события ломают это правило и смешивают обработку, что может создать неудобства.

Чтобы сделать событие асинхронным — можно обернуть соответствующий вызов `text.focus()` в `setTimeout(..., 0)`:

```
1 button.onclick = function() {  
2   ...  
3   setTimeout(function() {  
4     text.focus();  
5   }, 0);  
6   ...  
7 };
```

Далее мы рассмотрим другие случаи, когда такой трюк может быть полезен.

## Позволить родителю обработать событие

Как мы знаем, обычно событие сначала происходит на потомке, а потом всплывает к родителям.

**Что же делать, если дочерний элемент хочет обработать событие *после* того, как на событие среагирует родитель?**

Например, на событие `document.onkeydown` срабатывает общий обработчик «горячих клавиш», а на элементе `elem.onkeydown` стоит какой-то частный обработчик ввода, который должен срабатывать *после общего*.

Усложним задачу — пускай при обработке `keydown` на элементе `elem` нужно часть действий сделать прямо сейчас, а часть — после того, как событие обработает родитель.

Можно использовать фазу захвата (кроме IE8-) и разбить обработчик на две части: одна сработает на фазе захвата, вторая — на фазе всплытия. Но есть и отличный вариант с `setTimeout`.

**Решение — отложить обработку через `setTimeout(..., 0)`, и тем самым дать событию всплыть.**

Например:

```

01 <input type="button" value="Нажми меня">
02
03 <script>
04 var input = document.body.children[0];
05
06 input.onclick = function() {
07
08     function handle() {
09         input.value += ' -> input';
10     }
11
12     setTimeout(handle, 0); // отложить обработку
13 }
14
15 document.onclick = function() {
16     input.value += ' -> document';
17 }
18 </script>

```

Нажми меня

## Позволить действию браузера завершиться

Ряд событий происходят до того, как браузер обработает действие. Например, событие `onkeydown` сначала выполнит обработчик, а уже потом символ появится в поле ввода. Это сделано намеренно, чтобы обработчик мог отменить действие браузера.

Но при этом мы не можем обработать результат ввода.

Давайте посмотрим, о чём речь, на примере. В поле ниже обработчик приводит значение к верхнему регистру при каждом нажатии. Попробуйте набрать что-нибудь:

```

1 <input id="my" type="text" placeholder="keydown">
2
3 <script>
4 document.getElementById('my').onkeydown = function() {
5     this.value = this.value.toUpperCase();
6 };
7 </script>

```

keydown

Попробовали? Не работает! Значение увеличивается, *но кроме последнего символа*, т.к. браузер добавит его *после* того, как обработка `keydown` завершится.





### Вариант с keyup

Есть событие `keyup`, которое происходит после отпускания клавиши. Мы могли бы использовать его. Но в этом случае символ будет изначально показан маленьким, и поменяет регистр при отпускании клавиши.

Это выглядит странно, попробуйте:

```
1 <input id="my" type="text" placeholder="keyup">
2
3 <script>
4 document.getElementById('my').onkeyup = function() {
5   this.value = this.value.toUpperCase();
6 };
7 </script>
```

keyup

### Как заставить обработчик сработать *после* действия браузера?

Решение — отложить обработку через `setTimeout(..., 0)`:

```
01 <input id="my" type="text" placeholder="Оптимальный вариант">
02
03 <script>
04 document.getElementById('my').onkeydown = function() {
05   var self = this;
06
07   function handle() {
08     self.value = self.value.toUpperCase()
09   }
10   setTimeout(handle, 0);
11
12 };
13
14 </script>
```

Оптимальный вариан

Тогда обработчик запустится *после* добавления символа, но достаточно скоро, чтобы посетитель не заметил задержку.

## Итого

- ➡ JavaScript выполняется в едином потоке. Современные браузеры позволяют порождать подпроцессы [Web Workers \[14\]](#), они выполняются параллельно и могут отправлять/принимать сообщения, но не имеют доступа к DOM.
- ➡ Большинство событий — асинхронные. Синхронными являются вложенные события, инициированные из кода.
- ➡ Трюк с `setTimeout(func, 0)` можно применить в обработчиках:

- Чтобы сделать вложенное событие асинхронным.
- Чтобы дать событию всплыть и отработать обработчикам на родителях.
- Чтобы дать браузеру выполнить своё действие.

Вместо `setTimeout(..., 0)` рекомендуется использовать [setImmediate](#) [15].

## События мыши

---

## Введение: клики, кнопка, координаты

---

В этой статье мы познакомимся со списком событий мыши, рассмотрим их общие свойства, а также те события, которые связаны с кликом.

### 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.

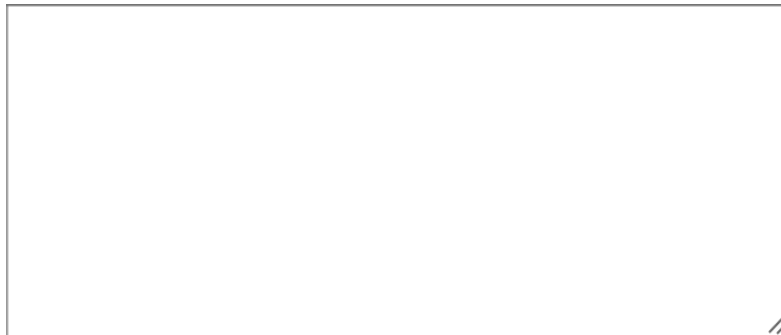
В тех случаях, когда одно действие генерирует несколько событий, их порядок фиксирован. То есть, обработчики вызовутся в порядке `mousedown` -> `mouseup` -> `click`.

Кликните по кнопке ниже и вы увидите, какие при этом происходят события. Попробуйте также двойной клик.

На тест-стенде ниже все мышинные события записываются, и если между событиями проходит больше 1 секунды, то они для удобства чтения отделяются линией. Также присутствуют свойства `which/button`, по которым можно определить кнопку мыши. Мы их рассмотрим далее.

Нажмите, чтобы увидеть события

Очистить



Каждое событие обрабатывается независимо. Например, при клике события `mouseup` + `click` возникают одновременно, но обрабатываются последовательно. Сначала полностью завершается обработка `mouseup`, затем запускается `click`.



#### Отличия `dblclick` в IE8-

Все браузеры, кроме IE8-, генерируют `dblclick` *в дополнение* к другим событиям.

То есть, обычно: `mousedown` (нажал) → `mouseup+click` (отжал) → `mousedown` (нажал) → `mouseup+click+dblclick` (отжал).

**IE8- на втором клике не генерирует `mousedown` и `click`.**

Получается `mousedown` (нажал) → `mouseup+click` (отжал) → (нажал, без события) → `mouseup+dblclick` (отжал).

**Поэтому отловить двойной клик в IE8-, отслеживая только `click`, нельзя. Нужно именно событие `dblclick`.**

## 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 \[16\]](#) & («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`.**



### Кроссбраузерно ставим свойство `which`

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:

```
1 function fixWhich(e) {  
2   if (!e.which && e.button) { // если which нет, но есть button...  
3     if (e.button & 1) e.which = 1; // левая кнопка  
4     else if (e.button & 4) e.which = 2; // средняя кнопка  
5     else if (e.button & 2) e.which = 3; // правая кнопка  
6   }  
7 }
```

## 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:

```
1 <button>Ctrl+Shift+Клики меня!</button>
2
3 <script>
4   document.body.children[0].onclick = function(e) {
5     e = e || event;
6     if (!e.ctrlKey || !e.shiftKey) return;
7     alert('Ура!');
8   }
9 </script>
```

Ctrl+Shift+Клики меня!

## 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.

Проведите мышью над полем ввода, чтобы увидеть `clientX/clientY`:

```
<input onmousemove="this.value = event.clientX+' '+event.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.

Проведите мышью над полем ввода, чтобы увидеть `pageX/pageY` (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 \[17\]](#).

Мы же здесь приведем готовый вариант, который позволяет нам получить `pageX/pageY` для старых IE:

```
01 function fixPageXY(e) {
02   if (e.pageX == null && e.clientX != null) { // если нет pageX..
03     var html = document.documentElement;
04     var body = document.body;
05
06     e.pageX = e.clientX + (html.scrollLeft || body && body.scrollLeft || 0);
07     e.pageX -= html.clientLeft || 0;
08
09     e.pageY = e.clientY + (html.scrollTop || body && body.scrollTop || 0);
10     e.pageY -= html.clientTop || 0;
11   }
12 }
```

## Демо получения координат мыши

---

Следующий пример показывает координаты мыши относительно документа, для всех браузеров.

```
1 document.onmousemove = function(e) {  
2     e = e || window.event;  
3     fixPageXY(e);  
4  
5     document.getElementById('mouseX').value = e.pageX;  
6     document.getElementById('mouseY').value = e.pageY;  
7 }
```

Координата по X:

Координата по Y:

## 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"

---

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:

```

1 function fixRelatedTarget(e) {
2   if (!e.relatedTarget) {
3     if (e.type == 'mouseover') e.relatedTarget = e.fromElement;
4     if (e.type == 'mouseout') e.relatedTarget = e.toElement;
5   }
6 }

```



Значение `relatedTarget` (`toElement/fromElement`) может быть `null`

Такое бывает, например, когда мышь приходит из-за пределов окна у `mouseover` будет `relatedTarget = null`.

Ниже находится демо-стенд для событий. Он находится в `IFRAME`, так что значение `null` можно увидеть, если навести курсор на «Боб» сверху.

Боб

Алиса

Mouseover: элементы Mouseout: элементы

## 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.

## Тест для `mousemove/over/out` при быстром движении

---

Попробуйте это на тестовом стенде ниже. Молниеносно проведите мышью над тестовыми элементами. При этом может не быть ни одного события или их получит только красный `div`, а может быть только зеленый.

А еще попробуйте быстро вывести мышь из красного потомка сквозь зеленый. Родительский элемент будет проигнорирован.

Тест

Очистить



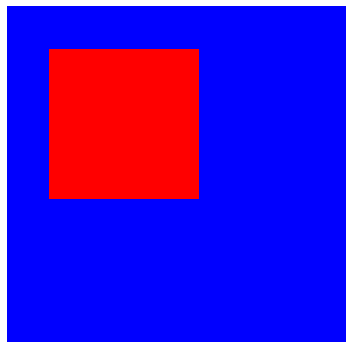
## «Лишний» `mouseout` при уходе на потомка

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.

Это можно увидеть в примере ниже. В нём красный `div` вложен в синий. На синем стоит обработчик, который записывает его `mouseover/mouseout`. Зайдите на синий элемент, а потом переведите мышь на красный — и наблюдайте за событиями:





Очистить

- При заходе на синий — на нём сработает `mouseover`.
- При переходе с синего на красный — будут выведены сразу два события:
  1. `mouseout [target: blue]` — уход с родителя.
  2. `mouseover [target: red]` — как ни странно, «обратный переход» на родителя.

На самом деле, обратного перехода нет. Событие `mouseover` появляется здесь, поскольку сработало на потомке (посмотрите на `target`), а затем всплыло.

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.

Как это влияет на поведение нашего кода, который использует эти события? Например, показывает и скрывает подсказку?

Если действия при наведении и уходе курсора простые, например `style.display = "block/none"`, то можно вообще ничего не заметить. Для подсказки будет максимум небольшое моргание.

Если же происходит что-то сложное, то бывает важно отследить момент «настоящего» ухода. Это вполне возможно. Для этого можно использовать альтернативные события `mouseenter/mouseleave`, которые мы рассмотрим далее.

## 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.

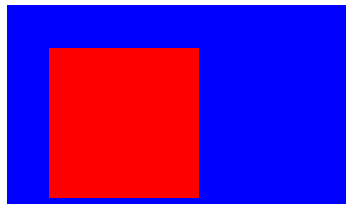
1. При переходе на потомка курсор не уходит с родителя.

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.

2. События `mouseenter/mouseleave` не всплывают.

Эти события описаны в спецификации [DOM Level 3 \[18\]](#) и поддерживаются IE6+, а также Opera 11.10+ и Firefox 10+.

Если они поддерживаются вашим браузером, то вы увидите их, проведя курсором над голубым DIV'ом ниже. Обработчик стоит только на внешнем, синем элементе. Обратите внимание: лишних событий при переходе на красного потомка нет.





Очистить

## 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:

```

01 <div style="padding:10px; margin:10px; border: 2px solid blue" id="outer">
02   <p style="border: 1px solid green">
03     Обработчики mouseover/mouseout стоят на синем родителе.
04   </p>
05   <blockquote style="border: 1px solid red">
06     ..Но срабатывают и при любых переходах по его потомкам!
07   </blockquote>
08 </div>
09 <b id="info">Тут будет информация о событиях.</b>
10
11 <script>
12 var outer = document.getElementById('outer')
13 var info = document.getElementById('info');
14
15 outer.onmouseout = function(e) {
16   e = e || event;
17   var target = e.target || e.srcElement;
18   info.innerHTML = e.type+', target:'+target.tagName;
19 };
20
21 outer.onmouseover = function(e) {
22   e = e || event;
23   var target = e.target || e.srcElement;
24   info.innerHTML = e.type+', target:'+target.tagName;
25 };
26
27 </script>

```

Обработчики mouseover/mouseout стоят на синем родителе.

..Но срабатывают и при любых переходах по его потомкам!

**Тут будет информация о событиях.**

Если двигать мышкой по этому примеру, то mouseover/out будут появляться не только при входе-уходе на outer, но и при перемещении по его потомкам.

Чтобы это исправить, напишем функцию-фильтр, которая берёт обработчик mouseout и возвращает обёртку, вызывающую его только в случаях реального ухода с элемента.

```

01  /**
02  * Возвращает обработчик,
03  * который вызывает handler при реальном уходе с элемента
04  * использование: elem.onmouseout = handleMouseLeave(function(e) { .. })
05  */
06  function handleMouseLeave(handler) {
07
08      return function(e) {
09          e = e || event; // IE
10          var toElement = e.relatedTarget || e.toElement; // IE
11
12          // проверяем, мышь ушла на элемент внутри текущего?
13          while (toElement && toElement !== this) {
14              toElement = toElement.parentNode;
15          }
16
17          if (toElement == this) { // да, мы всё еще внутри родителя
18              return; // мы перешли с родителя на потомка, лишнее событие
19          }
20
21          return handler.call(this, e);
22      };
23  }

```

Аналогично можно создать обработчик для mouseenter:

```

01  function handleMouseEnter(handler) {
02
03      return function(e) {
04          e = e || event; // IE
05          var toElement = e.relatedTarget || e.srcElement; // IE
06
07          // проверяем, мышь пришла с элемента внутри текущего?
08          while (toElement && toElement !== this) {
09              toElement = toElement.parentNode;
10          }
11
12          if (toElement == this) { // да, мышь перешла изнутри родителя
13              return; // мы перешли на родителя из потомка, лишнее событие
14          }
15
16          return handler.call(this, e);
17      };
18  }

```

Использование:

```

01 <div style="padding:10px; margin:10px; border: 2px solid blue" id="outer">
02   <p style="border: 1px solid green">На синем родителе эмулируются обработчики mouseenter/mouseleave.</p>
03   <blockquote style="border: 1px solid red">Без лишних событий.</blockquote>
04 </div>
05 <b id="info">Тут будет информация о событиях.</b>
06
07 <script>
08 var outer = document.getElementById('outer')
09 var info = document.getElementById('info');
10
11 outer.onmouseout = handleMouseLeave(function(e) {
12   e = e || event;
13   var target = e.target || e.srcElement;
14   info.innerHTML = e.type+', target:'+target.tagName;
15 });
16
17 outer.onmouseover = handleMouseEnter(function(e) {
18   e = e || event;
19   var target = e.target || e.srcElement;
20   info.innerHTML = e.type+', target:'+target.tagName;
21 });
22
23 </script>

```

На синем родителе эмулируется обработка mouseenter/mouseleave.

События не инициируются при переходе по потомкам.

Тут будет информация о событиях.

## 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`.

# Колёсико мыши: "wheel" и аналоги

Колёсико мыши используется редко. Оно есть даже не у всех мышей. Поэтому существуют пользователи, которые в принципе не могут сгенерировать такое событие.

..Но, тем не менее, его использование может быть оправдано. Например, можно добавить дополнительные удобства для тех, у кого колёсико есть.

## Отличия колёсика от прокрутки

Несмотря на то, что колёсико мыши обычно ассоциируется с прокруткой, это совсем разные вещи.

- ➔ При прокрутке срабатывает событие [onscroll \[19\]](#) — рассмотрим его в дальнейшем. Оно произойдёт *при любой прокрутке*, в том числе через клавиатуру, но *только на прокручиваемых элементах*. Например, элемент с `overflow: hidden` в принципе не может сгенерировать `onscroll`.
- ➔ А событие `wheel` является чисто «мышинным». Оно генерируется *над любым элементом* при передвижении колеса мыши. При этом не важно, прокручиваемый он или нет. В частности, `overflow: hidden` никак не препятствует обработке колеса мыши.

Кроме того, событие `onscroll` происходит после прокрутки, а `onwheel` — до прокрутки, поэтому в нём можно отменить саму прокрутку (действие браузера).

## Зоопарк wheel в разных браузерах

Событие `wheel` появилось в [стандарте \[20\]](#) не так давно. Оно поддерживается IE9+, Firefox 17+. Возможно, другими браузерами на момент чтения этой статьи.

До него браузеры обрабатывали прокрутку при помощи событий [mousewheel \[21\]](#) (все кроме Firefox) и [DOMMouseScroll \[22\]](#), [MozMousePixelScroll \[23\]](#) (только Firefox).

Самые важные свойства современного события и его нестандартных аналогов:

### **wheel**

Свойство `deltaY` — количество прокрученных пикселей по горизонтали и вертикали. Существуют также свойства `deltaX` и `deltaZ` для других направлений прокрутки.

### **MozMousePixelScroll**

Срабатывает, начиная с Firefox 3.5, только в Firefox. Даёт возможность отменить прокрутку и получить размер в пикселях через свойство `detail`, ось прокрутки в свойстве `axis`.

### **DOMMouseScroll**

Существует в Firefox очень давно, отличается от предыдущего тем, что даёт в `detail` количество строк. Если не нужна поддержка Firefox < 3.5, то не нужно и это событие.

### **mousewheel**

Срабатывает в браузерах, которые ещё не реализовали `wheel`. В свойстве `wheelDelta` — условный «размер прокрутки», обычно равен 120 для прокрутки вверх и -120 — вниз. Он не соответствует какому-либо конкретному количеству пикселей.

Чтобы кросс-браузерно отловить прокрутку и, при необходимости, отменить её, можно использовать все эти события.

Пример:

```

01 if (elem.addEventListener) {
02   if ('onwheel' in document) {
03     // IE9+, FF17+
04     elem.addEventListener ("wheel", onWheel, false);
05   } else if ('onmousewheel' in document) {
06     // устаревший вариант события
07     elem.addEventListener ("mousewheel", onWheel, false);
08   } else {
09     // 3.5 <= Firefox < 17, более старое событие DOMMouseScroll пропустим
10     elem.addEventListener ("MozMousePixelScroll", onWheel, false);
11   }
12 } else { // IE<9
13   elem.attachEvent ("onmousewheel", onWheel);
14 }
15
16 function onWheel(e) {
17   e = e || window.event;
18
19   // wheelDelta не дает возможность узнать количество пикселей
20   var delta = e.deltaY || e.detail || e.wheelDelta;
21
22   var info = document.getElementById('delta');
23
24   info.innerHTML = +info.innerHTML + delta;
25
26   e.preventDefault ? e.preventDefault() : (e.returnValue = false);
27 }

```

В действии:

Прокрутка: 0

Прокрути надо мной.





### Ошибка в IE8

В браузере IE8 (только версия 8) есть ошибка. При наличии обработчика `mousewheel` — элемент не скроллится. Иначе говоря, действие браузера отменяется по умолчанию.

Это, конечно, не имеет значения, если элемент в принципе не прокручиваемый.

## Устранение IE-несовместимостей: "fixEvent"

Если собрать все несовместимости для событий, свойственные IE, которые описаны в этой главе — их устранение можно описать единой функцией `fixEvent`.

Эта функция работает так:

```
1 | elem.onclick = function(e) {  
2 |   e = fixEvent [24](e); // в начале обработчика  
3 |   ...  
4 | }
```

Объект события, при необходимости, будет взят из `window.event`.

Функция `fixEvent` добавляет объекту события в IE следующие стандартные свойства:

- ➡ `target`
- ➡ `relatedTarget`
- ➡ `pageX/pageY`
- ➡ `which`

Можно также передать `this` вторым аргументом, и он будет записан в `currentTarget`:

```
1 | elem.onclick = function(e) {  
2 |   e = fixEvent [25](e, this);  
3 |   alert(e.currentTarget == this); // true, как и должно быть по стандарту  
4 |   ...  
5 | }
```

Код функции:

```

01 function fixEvent(e, _this) {
02     e = e || window.event;
03
04     if (!e.currentTarget) e.currentTarget = _this;
05     if (!e.target) e.target = e.srcElement;
06
07     if (!e.relatedTarget) {
08         if (e.type == 'mouseover') e.relatedTarget = e.fromElement;
09         if (e.type == 'mouseout') e.relatedTarget = e.toElement;
10     }
11
12     if (e.pageX == null && e.clientX != null ) {
13         var html = document.documentElement;
14         var body = document.body;
15
16         e.pageX = e.clientX + (html.scrollLeft || body && body.scrollLeft || 0);
17         e.pageX -= html.clientLeft || 0;
18
19         e.pageY = e.clientY + (html.scrollTop || body && body.scrollTop || 0);
20         e.pageY -= html.clientTop || 0;
21     }
22
23     if (!e.which && e.button) {
24         e.which = e.button & 1 ? 1 : ( e.button & 2 ? 3 : (e.button & 4 ? 2 : 0) );
25     }
26
27     return e;
28 }

```

Мы будем использовать эту функцию дальше в учебнике.

## Основы Drag'n'Drop

Drag'n'Drop — это возможность захватить мышью элемент и перенести его. В свое время это было замечательным открытием в области интерфейсов, которое позволило упростить большое количество операций.

Одно из самых очевидных применений Drag'n'Drop - переупорядочение данных. Это могут быть блоки, элементы списка, и вообще — любые DOM-элементы и их наборы.

Перенос мышкой может заменить целую последовательность кликов. И, самое главное, он упрощает внешний вид интерфейса: функции, реализуемые через Drag'n'Drop, в ином случае потребовали бы дополнительных полей, виджетов и т.п.

## HTML5 Drag'n'Drop

В современном стандарте HTML5 есть поддержка Drag'n'Drop при помощи [специальных событий](#) [26]. Однако, эти события слабо поддерживаются в IE<10, а в IE10 поддержка есть, но местами отличается от стандарта.

Их важная особенность заключается в том, что *они позволяют обработать перенос файла в браузер*, и при этом дают JavaScript доступ к содержимому этих файлов. Например, человек перенес картинку в редактор, JavaScript может загрузить её на сервер и тут же отобразить. Для этой цели HTML5 события — единственное возможное решение.

События HTML5 заслуживают отдельного обсуждения как способ реализовать простые вещи просто (для тех браузеров, в которых это

работает). Поэтому их использованию будет посвящена отдельная статья.

С другой стороны, у них есть и недостатки. Ряд сценариев сложно или нельзя реализовать при помощи встроенных событий (например, вставку между элементами).

Реализация переноса при помощи событий мыши надежна, кросс-браузерна, и позволяет делать всё, что захотим.

**Далее речь пойдет о реализации Drag'n'Drop при помощи событий мыши. Изложенные методы применяются в различных элементах управления для обработки действий «захватить - потянуть - отпустить».**

## Основная логика Drag'n'Drop

Для организации Drag'n'Drop нужно:

1. При помощи события `mousedown` отследить нажатие на переносимом элементе
2. При нажатии — подготовить мячик к перемещению: дать ему `position: absolute`. При этом показать его на том же месте, назначив `left/top` по координатам курсора. Далее отслеживать движение мыши через `mousemove` и передвигать переносимый элемент по странице. Мяч уже имеет `position: absolute`, так что это делается путем смены `left/top`.
3. При отпускании кнопки мыши, то есть наступлении события `mouseup` — остановить перенос элемента и произвести все действия, связанные с окончанием Drag'n'Drop.

В следующем примере эти шаги реализованы. Мяч можно двигать мышью:

```
01 var ball = document.getElementById('ball');
02
03 ball.onmousedown = function(e) { // отследить нажатие
04     var self = this;
05     e = fixEvent [27](e);
06
07     // подготовить к перемещению
08     // разместить на том же месте, но в абсолютных координатах
09     this.style.position = 'absolute';
10     moveAt(e);
11     // переместим в body, чтобы мяч был точно не внутри position:relative
12     document.body.appendChild(this);
13
14     this.style.zIndex = 1000; // показывать мяч над другими элементами
15
16     // передвинуть мяч под координаты курсора
17     function moveAt(e) {
18         self.style.left = e.pageX-25+'px'; // 25 - половина ширины/высоты мяча
19         self.style.top = e.pageY-25+'px';
20     }
21
22     // перемещать по экрану
23     document.onmousemove = function(e) {
24         e = fixEvent(e);
25         moveAt(e);
26     }
27
28     // отследить окончание переноса
29     this.onmouseup = function() {
30         document.onmousemove = self.onmouseup = null;
31     }
32 }
```

В действии:

Кликните по мячу и тяните, чтобы двигать его.



Попробуйте этот пример в Firefox, в IE. Он не совсем работает, верно?

Это потому, что мы не учли ряд тонкостей, которые сейчас рассмотрим.

## Отмена переноса браузера

При нажатии мышью браузер начинает выполнять свой собственный, встроенный Drag'n'Drop для изображений, который и портит наш перенос.

Чтобы браузер не вмешивался, нужно отменить действие для события dragstart:

```
ball.ondragstart = function() {  
    return false;  
}
```

Исправленный пример:

```
01 var ball = document.getElementById('ball2');  
02  
03 ball.onmousedown = function(e) {  
04     var self = this;  
05     e = fixEvent [28](e);  
06  
07     this.style.position = 'absolute';  
08     moveAt(e);  
09     document.body.appendChild(this);  
10  
11     this.style.zIndex = 1000;  
12  
13     function moveAt(e) {  
14         self.style.left = e.pageX-25+'px';  
15         self.style.top = e.pageY-25+'px';  
16     }  
17  
18     document.onmousemove = function(e) {  
19         e = fixEvent(e);  
20         moveAt(e);  
21     }  
22  
23     this.onmouseup = function() {  
24         document.onmousemove = self.onmouseup = null;  
25     }  
26 }  
27  
28 ball.ondragstart = function() {  
29     return false;  
30 };
```

В действии:

Кликните по мячу и тяните, чтобы двигать его.



## mousemove — на document

**Почему событие `mousemove` в примере отслеживается на `document`, а не на `ball`?**

..И правда, почему? С первого взгляда кажется, что мышь всегда над мячом и событие можно повесить на сам мяч, а не на документ.

Однако, на самом деле **мышь во время переноса не всегда над мячом**. Вспомните, браузер регистрирует `mousemove` часто, но не для каждого пикселя.

Быстрое движение курсора вызовет `mousemove` уже не над мячом, а, например, в дальнем конце страницы. Вот почему мы должны отслеживать `mousemove` на всём `document`.

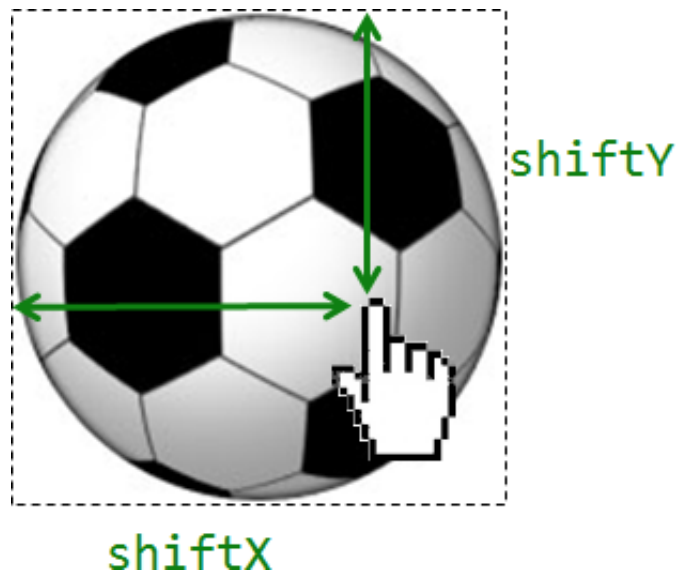
## Правильное позиционирование

В примерах выше мяч позиционируется в центре под курсором мыши:

```
self.style.left = e.pageX - 25 + 'px';  
self.style.top = e.pageY - 25 + 'px';
```

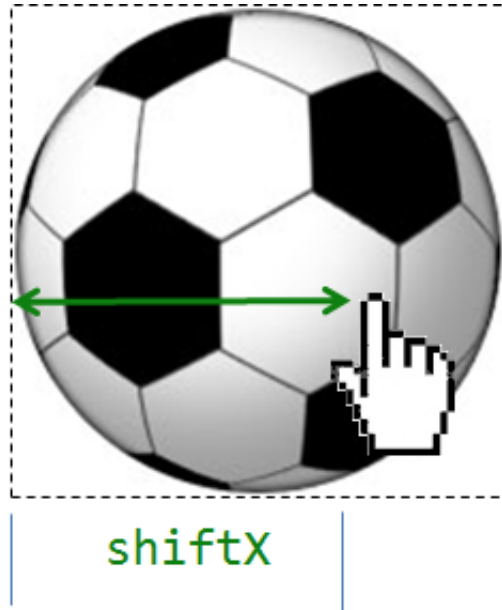
Число 25 здесь — половина длины мячика. Оно использовано здесь потому, что если поставить `left/top` ровно в `pageX/pageY`, то мячик прилипнет верхним-левым углом к курсору мыши. Будет некрасиво.

**Для правильного переноса необходимо, чтобы изначальный сдвиг курсора относительно элемента сохранялся.**



→ Когда человек нажимает на мячик `mousedown` — курсор сдвинут относительно левого-верхнего угла мяча на расстояние `shiftX/shiftY`.

Получить значения `shiftX/shiftY` легко: достаточно вычесть из координат курсора левую-верхнюю границу мячика.



`getCoords(ball).left`    `pageX`

```
// onmousedown  
shiftX = e.pageX - getCoords(ball).left;  
shiftY = e.pageY - getCoords(ball).top;
```

→ Далее при переносе мяча мы располагаем его `left/top` с учетом сдвига, то есть вот так:

```
// onmousemove  
self.style.left = e.pageX - shiftX + 'px';  
self.style.top = e.pageY - shiftY + 'px';
```

**Пример с правильным позиционированием:**

В этом примере позиционирование осуществляется не на 25px, а с учётом изначального сдвига.

```

01 var ball = document.getElementById('ball3');
02
03 ball.onmousedown = function(e) {
04     var self = this;
05     e = fixEvent [29](e);
06
07     var coords = getCoords(this);
08     var shiftX = e.pageX - coords.left;
09     var shiftY = e.pageY - coords.top;
10
11     this.style.position = 'absolute';
12     document.body.appendChild(this);
13     moveAt(e);
14
15     this.style.zIndex = 1000; // над другими элементами
16
17     function moveAt(e) {
18         self.style.left = e.pageX - shiftX + 'px';
19         self.style.top = e.pageY - shiftY + 'px';
20     }
21
22     document.onmousemove = function(e) {
23         e = fixEvent(e);
24         moveAt(e);
25     };
26
27     this.onmouseup = function() {
28         document.onmousemove = self.onmouseup = null;
29     };
30
31 }
32
33 ball.ondragstart = function() {
34     return false;
35 };

```

В действии:

Кликните по мячу и тяните, чтобы двигать его.



Различие особенно заметно, если захватить мяч за правый-нижний угол. В предыдущем примере мячик «прыгнет» серединой под курсор, в этом — будет плавно переноситься с текущей позиции.

## Итого

Мы рассмотрели «минимальный каркас» Drag 'n' Drop.

Его компоненты:

1. События `mousedown` -> `document.onmousemove` -> `onmouseup`.

2. Передвижение с учётом изначального сдвига `shiftX/shiftY`.
3. Отмена действия браузера по событию `dragstart`.

На этой основе можно сделать очень многое.

- ➡ При `mouseup` можно обработать окончание переноса, произвести изменения в данных, если они нужны.
- ➡ Во время самого переноса можно подсвечивать элементы, над которыми проходит элемент.

Это и многое другое мы рассмотрим в статье про [Drag'n'Drop объектов \[30\]](#).

## Drag'n'Drop объектов

---

В [предыдущей статье \[31\]](#) мы рассмотрели основы Drag'n'Drop. Здесь мы построим на этой основе фреймворк, предназначенный для переноса *объектов* — элементов списка, узлов дерева и т.п.

Почти все javascript-библиотеки реализуют Drag'n'Drop так, как написано (или хуже 😊). Зная, что и как, вы сможете легко написать свой код переноса или поправить, адаптировать существующую библиотеку под себя.

### Документ

Как пример задачи — возьмём документ с иконками браузера («объекты переноса»), которые можно переносить в компьютер («цель переноса»):

- ➡ Элементы, которые можно переносить (иконки браузеров), помечены атрибутом `draggable`.
- ➡ Элементы, на которые можно положить (компьютер), имеют атрибут `droppable`.

```
1 
2 
3 
4 
5 
6
7 <p>Браузер переносить сюда:</p>
8
9 
```

Работающий пример с переносом.





Браузер переносить сюда:

Далее мы рассмотрим, как делается фреймворк для таких переносов, а в перспективе — и для более сложных.

Требования:

- ➡ Поддержка большого количества элементов без «тормозов».
- ➡ Продвинутое возможности по анимации переноса.
- ➡ Удобная обработка успешного и неудачного переноса.

## Начало переноса

Чтобы начать перенос элемента, мы должны отловить нажатие левой кнопки мыши на нём. Для этого используем событие `mousedown`.

..И здесь нас ждёт первая особенность переноса объектов. На что ставить обработчик?

**Переносимых элементов может быть много.** В нашем документе-примере это всего лишь несколько иконок, но если мы хотим переносить элементы списка или дерева, то их может быть 100 штук и более.

Назначать обработчик на каждый DOM-элемент — нецелесообразно. Проще всего решить эту задачу делегированием.

**Повесим обработчик `mousedown` на контейнер, который содержит переносимые элементы**, и будем определять нужный элемент поиском ближайшего `draggable` вверх по иерархии от `event.target`. В качестве контейнера здесь и далее будем брать `document`.

Найденный переносимый объект сохраним в переменной `dragObject` и начнём двигать.

Код обработчика `mousedown`:

```

01 var dragObject = {};
02
03 document.onmousedown = function(e){
04     e = fixEvent [32](e);
05
06     if (e.which != 1) {
07         return; // нажатие правой кнопкой не запускает перенос
08     }
09
10     // найти ближайший draggable, пройдя по цепочке родителей target
11     var elem = findDraggable(e.target, document);
12
13     if (!elem) return; // не нашли, клик вне draggable объекта
14
15     // запомнить переносимый объект
16     dragObject.elem = elem;
17
18     // запомнить координаты, с которых начал перенос объекта
19     dragObject.downX = e.pageX;
20     dragObject.downY = e.pageY;
21 }

```

В обработчике используется функция `findDraggable` для поиска переносимого элемента по `event.target`:

```

1 function findDraggable(event) {
2     var elem = event.target;
3
4     // найти ближайший draggable, пройдя по цепочке родителей target
5     while(elem != document && elem.getAttribute('draggable') == null) {
6         elem = elem.parentNode;
7     }
8     return elem == document ? null : elem;
9 }

```

**Здесь мы пока не начинаем перенос, потому что нажатие на элемент вовсе не означает, что его собираются куда-то двигать.** Возможно, на нём просто кликают.

Это важное различие. Нужно отличать перенос от клика, в том числе — от клика, который сопровождается нечаянным перемещением на пару пикселей (рука дрогнула).

Иначе при клике элемент будет сниматься со своего места, и потом тут же возвращаться обратно (никуда не положили). Это лишняя работа и, вообще, выглядит некрасиво.

**Поэтому в `mousedown` мы запоминаем, какой элемент и где был зажат, а начало переноса будем инициировать из `mousemove`, если он передвинут хотя бы на несколько пикселей.**

## Перенос элемента

Первой задачей обработчика `mousemove` является инициировать начало переноса, если элемент передвинули в зажатом состоянии.

Ну а второй задачей — отображать его перенос при каждом передвижении мыши. Схематично, обработчик имеет такой вид:

```

01 document.onmousemove = function(e) {
02     if (!dragObject.elem) return; // элемент не зажат
03
04     e = fixEvent [33](e);
05
06     if ( !dragObject.avatar ) { // элемент нажат, но пока не начали его двигать
07         ... начать перенос, присвоить dragObject.avatar = переносимый элемент
08     }
09
10     ...отобразить перенос элемента...
11 }

```

Здесь мы видим новое свойство `dragObject.avatar`. При начале переноса *аватар* делается из элемента и сохраняется в свойство `dragObject.avatar`.

**«Аватар» — это DOM-элемент, который перемещается по экрану.**

Почему бы не перемещать по экрану сам `draggable`-элемент? Зачем, вообще, нужен аватар?

Дело в том, что иногда сам элемент передвигать неудобно, например он слишком большой. А удобно создать некоторое визуальное представление элемента, и его уже переносить. Аватар дает такую возможность.

А в простейшем случае аватаром можно будет сделать сам элемент, и это не повлечёт дополнительных расходов.

## Визуальное перемещение аватара

---

Для того, чтобы отобразить перенос аватара, достаточно поставить ему `position: absolute` и менять координаты `left/top`.

Для использования абсолютных координат относительно документа, аватар должен быть прямым потомком `BODY`.

Следующий код готовит аватар к переносу:

```

1 // в начале переноса:
2 if (avatar.parentNode != document.body) {
3     document.body.appendChild(avatar); // переместить в BODY, если надо
4 }
5 avatar.style.zIndex = 9999; // сделать, чтобы элемент был над другими
6 avatar.style.position = 'absolute';

```

... А затем его можно двигать:

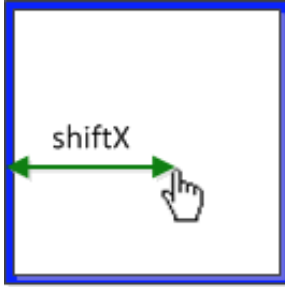
```

1 // при каждом движении мыши
2
3 avatar.style.left = новая координата + 'px';
4 avatar.style.top = новая координата + 'px';

```

### Как вычислять новые координаты `left/top` при переносе?

Чтобы элемент сохранял свою позицию под курсором, необходимо при нажатии запомнить его изначальный сдвиг относительно курсора, и сохранять его при переносе.



Этот сдвиг по горизонтали обозначен `shiftX` на рисунке выше. Аналогично, есть `shiftY`. Они вычисляются как расстояние между курсором и левой/верхней границей элемента при `mousedown`. Детали вычислений описаны в главе [Основы Drag'n'Drop \[34\]](#).

Таким образом, при `mousemove` мы будем назначать элементу координаты курсора с учетом сдвига `shiftX/shiftY`:

```
avatar.style.left = e.pageX - shiftX + 'px';  
avatar.style.top = e.pageY - shiftY + 'px';
```

## Полный код `mousemove`

---

Код `mousemove`, решающий задачу начала переноса и позиционирования аватара:

```

01 document.onmousemove = function(e) {
02     if (!dragObject.elem) return; // элемент не зажат
03
04     e = fixEvent [35](e);
05
06     if ( !dragObject.avatar ) { // если перенос не начат...
07
08         // посчитать дистанцию, на которую переместился курсор мыши
09         var moveX = e.pageX - dragObject.downX;
10         var moveY = e.pageY - dragObject.downY;
11         if ( Math.abs(moveX) < 3 && Math.abs(moveY) < 3 ) {
12             return; // ничего не делать, мышь не передвинулась достаточно далеко
13         }
14
15         dragObject.avatar = createAvatar(e); // захватить элемент
16         if (!dragObject.avatar) {
17             dragObject = {}; // аватар создать не удалось, отмена переноса
18             return; // возможно, нельзя захватить за эту часть элемента
19         }
20
21         // аватар создан успешно
22         // создать вспомогательные свойства shiftX/shiftY
23         var coords = getCoords/avatar);
24         dragObject.shiftX = dragObject.downX - coords.left;
25         dragObject.shiftY = dragObject.downY - coords.top;
26
27         startDrag(e); // отобразить начало переноса
28     }
29
30     // отобразить перенос объекта при каждом движении мыши
31     dragObject.avatar.style.left = e.pageX - dragObject.shiftX + 'px';
32     dragObject.avatar.style.top = e.pageY - dragObject.shiftY + 'px';
33
34     return false;
35 }

```

Здесь используются две функции для начала переноса: `createAvatar(e)` и `startDrag(e)`.

Функция `createAvatar(e)` создает аватар. В нашем случае в качестве аватара берется сам `draggable` элемент. После создания аватара в него записывается функция `avatar.rollback`, которая задает поведение при отмене переноса.

Как правило, отмена переноса влечет за собой разрушение аватара, если это был клон, или возвращение его на прежнее место, если это сам элемент.

В нашем случае для отмены переноса нужно запомнить старую позицию элемента и его родителя.

```

01 function createAvatar(e) {
02
03     // запомнить старые свойства, чтобы вернуться к ним при отмене переноса
04     var avatar = dragObject.elem;
05     var old = {
06         parent: avatar.parentNode,
07         nextSibling: avatar.nextSibling,
08         position: avatar.position || '',
09         left: avatar.left || '',
10         top: avatar.top || '',
11         zIndex: avatar.zIndex || ''
12     };
13
14     // функция для отмены переноса
15     avatar.rollback = function() {
16         old.parent.insertBefore(avatar, old.nextSibling);
17         avatar.style.position = old.position;
18         avatar.style.left = old.left;
19         avatar.style.top = old.top;
20         avatar.style.zIndex = old.zIndex;
21     };
22
23     return avatar;
24 }

```

Функция `startDrag(e)` иницирует начало переноса и позиционирует аватар на странице.

```

1 function startDrag(e) {
2     var avatar = dragObject.avatar;
3
4     document.body.appendChild(avatar);
5     avatar.style.zIndex = 9999;
6     avatar.style.position = 'absolute';
7 }

```

## Окончание переноса

---

Окончание переноса происходит по событию `mouseup`.

Его обработчик можно поставить на аватаре, т.к. аватар всегда под курсором и `mouseup` происходит на нем. Но для универсальности и большей гибкости (вдруг мы захотим перемещать аватар *рядом* с курсором?) поставим его, как и остальные, на `document`.

Задача обработчика `mouseup`:

1. Обработать успешный перенос, если он идет (существует аватар)
2. Очистить данные `dragObject`.

Это дает нам следующий код:

```

01 document.onmouseup = function(e) {
02     // (1) обработать перенос, если он идет
03     if (dragObject.avatar) {
04         e = fixEvent [36](e);
05         finishDrag(e);
06     }
07
08     // в конце mouseup перенос либо завершился, либо даже не начинался
09     // (2) в любом случае очистим "состояние переноса" dragObject
10     dragObject = {};
11 }

```

Для завершения переноса в функции `finishDrag(e)` нам нужно понять, на каком элементе мы находимся, и если над `draggable` — обработать перенос, а нет — откатиться:

```

1 function finishDrag(e) {
2     var dropElem = findDroppable(e);
3
4     if (dropElem) {
5         ... успешный перенос ...
6     } else {
7         ... отмена переноса ...
8     }
9 }

```

## Определяем элемент под курсором

Чтобы понять, над каким элементом мы остановились — используем метод `document.elementFromPoint(clientX, clientY)` [37], который мы обсуждали в разделе [координаты](#) [38]. Этот метод получает координаты *относительно окна* и возвращает самый глубокий элемент, который там находится.

Функция `findDroppable(event)`, описанная ниже, использует его и находит самый глубокий элемент с атрибутом `draggable` под курсором мыши:

```

01 // возвратит ближайший draggable или null
02 // предварительный вариант findDroppable, исправлен ниже!
03 function findDroppable(event) {
04
05     // взять элемент на данных координатах
06     var elem = document.elementFromPoint(event.clientX, event.clientY);
07
08     // пройти вверх по цепочке родителей в поисках draggable
09     while(elem != document && elem.getAttribute('draggable') == null) {
10         elem = elem.parentNode;
11     }
12
13     return elem == document ? null : elem;
14 }

```

Обратите внимание — для `elementFromPoint` нужны координаты относительно окна `clientX/clientY`, а не `pageX/pageY`.

Вариант выше — предварительный. Он не будет работать. Если попробовать применить эту функцию, будет все время возвращать один и тот же элемент! А именно — *текущий переносимый*. Почему так?

.. Дело в том, что в процессе переноса под мышкой находится именно аватар. При начале переноса ему даже `z-index` ставится большой,

чтобы он был поверх всех остальных.

**Аватар перекрывает остальные элементы. Поэтому функция `document.elementFromPoint()` увидит на текущих координатах именно его.**

Чтобы это изменить, нужно либо поправить код переноса, чтобы аватар двигался *рядом* с курсором мыши, либо поступить проще:

1. Спрятать аватар.
2. Вызывать `elementFromPoint`.
3. Показать аватар.

Напишем вспомогательную функцию `getElementUnderClientXY(elem, clientX, clientY)`, которая это делает:

```
01  /* получить элемент на координатах clientX/clientY, под elem */
02  function getElementUnderClientXY(elem, clientX, clientY) {
03      // сохранить старый display и спрятать переносимый элемент
04      var display = elem.style.display || '';
05      elem.style.display = 'none';
06
07      // получить самый вложенный элемент под курсором мыши
08      var target = document.elementFromPoint(clientX, clientY);
09
10      // показать переносимый элемент обратно
11      elem.style.display = display;
12
13      if (!target || target == document) { // такое бывает при выносе за границы окна
14          target = document.body; // поправить значение, чтобы был именно элемент
15      }
16
17      return target;
18  }
```

Правильный код `findDraggable`:

```
1  function findDraggable(event) {
2      var elem = getElementUnderClientXY(dragObject.avatar, event.clientX, event.clientY);
3
4      while(elem != document && elem.getAttribute('draggable') == null) {
5          elem = elem.parentNode;
6      }
7
8      return elem == document ? null : elem;
9  }
```

## Сводим части фреймворка вместе

Сейчас в нашем распоряжении находятся основные фрагменты кода и решения всех технических подзадач.

Приведем их в нормальный вид.

### dragManager

---

Перенос будет координироваться единым объектом. Назовем его `dragManager`.



Для его создания используем не обычный синтаксис `{...}`, а вызов `new function`. Это позволит прямо при создании объявить дополнительные переменные и функции в замыкании, которыми могут пользоваться методы объекта, а также назначить обработчики:

```
01 var dragManager = new function() {  
02     var dragObject = {};  
03     var self = this; // для доступа к себе из обработчиков  
04     function onMouseDown(e) { ... }  
05     function onMouseMove(e) { ... }  
06     function onMouseUp(e) { ... }  
07     document.onmousedown = onMouseDown;  
08     document.onmousemove = onMouseMove;  
09     document.onmouseup = onMouseUp;  
10     this.onDragEnd = function(dragObject, dropElem) { };  
11     this.onDragCancel = function(dragObject) { };  
12 }  
13  
14  
15  
16  
17
```

Всю работу будут выполнять обработчики `onMouse*`, которые оформлены как локальные функции. В данном случае они ставятся на `document` через `on...`, но вы легко можете поменять это на `addEventListener/attachEvent`.

Внутренний объект `dragObject` будет содержать информацию об объекте переноса.

У него будут следующие свойства:

**elem**

Текущий зажатым мышью объект, если есть (ставится в `mousedown`).

**avatar**

Элемент-аватар, который передвигается по странице.

**downX/downY**

Координаты, на которых был клик `mousedown`

**shiftX/shiftY**

Относительный сдвиг курсора от угла элемента, вспомогательное свойство вычисляется в начале переноса.

**Задачей `dragManager` является общее управление переносом.** Для обработки его окончания вызываются методы `onDrag*`, которые назначаются внешним кодом.

Разработчик, подключив `dragManager`, описывает в этих методах, что делать при начале и завершении переноса.



Если в вашем распоряжении есть современный JavaScript-фреймворк, то можно заменить вызовы методов `onDrag*` на генерацию соответствующих событий.

## Реализация переноса иконок

С использованием этого фреймворка перенос иконок браузеров в компьютер реализуется просто:

```

01 dragManager.onDragEnd = function(dragObject, dropElem) {
02
03     // успешный перенос, показать улыбку классом computer-smile
04     dropElem.className = 'computer computer-smile';
05
06     // скрыть переносимый объект
07     dragObject.elem.style.display = 'none';
08
09     // убрать улыбку через 0.2 сек
10     setTimeout(function() { dropElem.className = 'computer'; }, 200);
11 };
12
13 dragManager.onDragCancel = function(dragObject) {
14     // откат переноса
15     dragObject.avatar.rollback();
16 };

```

Результат:



Браузер переносить сюда:

## Варианты расширения этого кода

Существует масса возможных применений Drag'n'Drop. Реализациях всех из них здесь превратила бы довольно простой фреймворк в страшнейшего монстра.

Поэтому мы разберем варианты расширений и их реализации, чтобы вы, при необходимости, легко могли написать то, что нужно.

### Захватывать элемент можно только за «ручку»

**Функция `createAvatar(e)` может быть модифицирована, чтобы захватывать элемент можно было, только ухватившись за определенную зону.**

Например, окно чата можно «взять», только захватив его за заголовок.

Для этого достаточно проверять по `e.target`, куда, всё же, нажал посетитель. Если взять элемент на данных координатах нельзя, то `createAvatar` может вернуть `false`, и перенос не будет начат.

## Не всегда можно положить на `draggable`

---

Бывает так, что не на любое место в `draggable` можно положить элемент.

Само решение «можно или нет» может зависеть как от переносимого элемента (или «аватара» как его полномочного представителя), так и от конкретного места в `draggable`, над которым посетитель отпустил кнопку мыши.

Например: в админке есть дерево всех объектов сайта: статей, разделов, посетителей и т.п.

- ➡ В этом дереве есть узлы различных типов: «статьи», «разделы» и «пользователи».
- ➡ Все узлы являются переносимыми объектами.
- ➡ Узел «статья» (`draggable`) можно переносить в «раздел» (`draggable`), а узел «пользователи» — нельзя. Но и то и другое можно поместить в «корзину».

**Эта задача решается добавлением проверки в `findDraggable(e)`. Эта функция знает и об аватаре и о событии.** При попытке положить в «неправильное» место функция `findDraggable(e)` должна возвращать `null`.

### Проверка прав

Рассмотрим эту задачу поглубже, так как она встречается часто.

Есть две наиболее частые причины, по которым аватар нельзя положить на потенциальный `draggable`.

Первую мы уже рассмотрели: «несовпадение типов». Это как раз тот случай, когда «узлы-разделы» дерева админки могут принимать только «статьи», и не могут — «посетителей».

В этом случае мы можем проверить это в `findDraggable`, так как вся необходимая информация о типах у нас есть.

Вторая причина — посложнее.

**Может не хватать прав на такое действие с `draggable/draggable`, в рамках наложенных на посетителя ограничений.** Например, человек не может положить статью именно в данный раздел.

А возможно, объект переноса удален из базы администратором, и браузер об этом (пока) не знает — мы должны корректно обрабатывать все случаи, включая этот.

Здесь сложность в том, что **окончательное решение знает только сервер**. Значит, нужно сделать запрос. А элемент после отпускания мыши не может «зависнуть» над элементом в ожидании ответа, нужно его куда-то положить.

**Как правило, применяют «оптимистичный» сценарий, по которому мы считаем, что перенос обычно успешен.**

При нём посетитель кладет объект туда, куда он хочет.

Затем, в коде `onDragEnd`:

1. Визуально обрабатывается завершение переноса, как будто все ок.
2. Производится асинхронный запрос к серверу, содержащий информацию о переносе.
3. Сервер обрабатывает перенос и возвращает ответ, все ли в порядке.
4. Если нет — выводится ошибка и возвращается `avatar.rollback()`. Аватар в этом случае должен предусматривать возможность отката после успешного завершения.

**Процесс общения с сервером сопровождается индикацией загрузки и, при необходимости, блокировкой новых операций переноса до получения подтверждения.**

## Индикация переноса

---

Можно добавить методы `onDragEnter/onDragMove/onDragLeave` для интеграции с внешним кодом, которые вызываются при заходе (`onDragEnter`), при каждом передвижении (`onDragMove`), и при выходе из `draggable` (`onDragLeave`).

При этом бывает, что нужно поддерживать перенос *в элемент*, но и перенос *между элементами*. Разберем два варианта такой ситуации:

### Поддержка трех видов переноса: «между», «под», «над» `draggable`

Этот сценарий встречается, когда можно вставить статью как *в* существующий «узел-раздел» дерева, так и *между* разделами.

В этом случае `draggable` делится на 3 части по высоте: 25% - 50% - 25%, берутся координаты элемента и определяется попадание координаты события на нужную часть.

В параметры `dragObject` добавляется флаг, обозначающий, куда по отношению к найденному `draggable` происходит перенос.

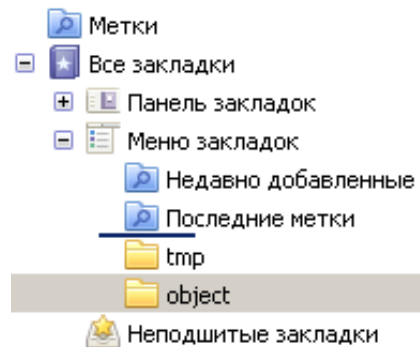
### Поддержка переноса только «между»

Здесь есть два варианта.

1. Во-первых, можно разделить `draggable` в отношении 50%/50% и по координатам смотреть, куда мы попали.
2. В некоторых случаях, например при вставке между `draggable`-элементами списка, можно считать, что *элемент вставляется перед тем LI, над которым проходит*.

А чтобы вставить после последнего — нужно перетащить аватар на сам DOM-элемент списка, но не на существующий `draggable`, а в пустое место, которое оставляется внизу для этой цели.

Индикацию «переноса между» удобнее всего делать либо раздвижением элементов, либо показом полосы-индикатора `border-top/border-bottom`, как показано на рисунке ниже:



## Анимация отмены переноса

---

Отмену переноса и возврат аватара на место можно красиво анимировать.


Один из частых вариантов — скольжение объекта обратно к исходному месту, откуда его взяли. Для этого достаточно поправить `avatar.revert()` соответствующим образом.

## Итого

Алгоритм Drag'n'Drop:

1. При `mousedown` запомнить координаты нажатия.
2. При `mousemove` инициировать перенос, как только зажаты элемент передвинули на 3 пикселя или больше.
  1. Создать аватар, если можно начать перенос с этой точки `draggable`.
  2. Перемещать его по экрану. Новые координаты ставить по `e.pageX/pageY` с учетом изначального сдвига элемента относительно курсора.
3. При `mouseup` обработать завершение переноса. Элемент под аватаром получить по координатам, предварительно спрятав аватар.

Получившийся попутно с обсуждением технических моментов Drag'n'Drop фреймворк обладает рядом особенностей:



- ➡ Он прост. Он действительно очень прост.
- ➡ Он полностью кросс-браузерный, не содержит хаков.
- ➡ Он позволяет работать с большим количеством потенциальных `draggable/droppable`.
- ➡ Его легко расширить и поменять.

Вы можете получить файлы и посмотреть итоговое демо в песочнице: <http://learn.javascript.ru/play/tutorial/browser/dnd/dragDemo/index.html>.

В зависимости от ваших задач, вы можете либо использовать его как отправную точку, либо реализовать свой.

ООП-вариант фреймворка находится в статье [Применяем ООП: Drag'n'Drop++ \[39\]](#).

## События клавиатуры

Здесь мы рассмотрим основные «клавиатурные» события и работу с ними.

### Тестовый стенд

Для того, чтобы лучше понять, как работают события клавиатуры, можно использовать тестовый стенд.

Попробуйте различные варианты нажатия клавиш в текстовом поле.

Предотвратить действие по умолчанию для:

☐ **keydown**

☐ **keypress**

☐ **keyup**

Игнорировать:

☐ **keydown**

☐ keypress

☐ keyup

Сфокусируйтесь на поле и нажмите какую-нибудь клавишу.

Поле для тестов

Журнал:

Очистить

```

01 document.getElementById('kinput').onkeydown = khandle;
02 document.getElementById('kinput').onkeyup = khandle;
03 document.getElementById('kinput').onkeypress = khandle;
04
05 function khandle(e) {
06     e = e || event;
07     if (document.forms.keyform[e.type + 'Ignore'].checked) return;
08
09     var evt = e.type;
10     while (evt.length < 10) evt += ' '
11     showmesg(evt +
12         ' keyCode=' + e.keyCode +
13         ' which=' + e.which +
14         ' charCode=' + e.charCode +
15         ' fromCharCode=' + String.fromCharCode(e.keyCode || e.charCode) +
16         (e.shiftKey ? ' +shift' : '') +
17         (e.ctrlKey ? ' +ctrl' : '') +
18         (e.altKey ? ' +alt' : '') +
19         (e.metaKey ? ' +meta' : '') +
20         (e.repeat ? ' +repeat' : ''), 'key'
21     )
22
23     if (document.forms.keyform[e.type + 'Stop'].checked) {
24         e.preventDefault ? e.preventDefault() : (e.returnValue = false);
25     }
26 }

```

По мере чтения статьи, если возникнут вопросы — возвращайтесь к этому стенду.

## Виды и свойства событий

### keydown и keyup

**События keydown/keyup происходят при нажатии/отпускании клавиши и позволяют получить её *скан-код* в свойстве keyCode.**

Скан-код клавиши одинаков в любой раскладке и в любом регистре. Например, клавиша **z** может означать символ "z", "Z" или "я", "Я" в русской раскладке, но её *скан-код* будет всегда одинаков: 90.

В действии:

```
<input onkeydown="this.nextSibling.innerHTML = event.keyCode"><b></b>
```

Нажмите клавишу, скан-код будет спр

### Какими бывают скан-коды?








**Для буквенно-цифровых клавиш, скан-код будет равен коду соответствующей заглавной английской буквы/цифры.**

Например, при нажатии клавиши **S** (не важно, каков регистр и раскладка) её скан-код будет равен "S".charCodeAt(0).

С другими символами, например, знаками пунктуации, ситуация тоже понятна. Есть таблица кодов, которую можно взять, например, из статьи Джона Уолтера: [JavaScript Madness: Keyboard Events \[40\]](#), или же можно нажать на нужную клавишу на [тестовом стенде \[41\]](#) и получить код.

Когда-то в этих кодах была масса кросс-браузерных несовместимостей. Сейчас всё проще — таблицы кодов в различных браузерах почти полностью совпадают.

Но некоторые несовместимости, всё же, остались. Вы можете увидеть их в таблице ниже. Слева — клавиша с символом, а справа — скан-коды в различных браузерах.

Клавиша	Firefox	Safari/Chrome/IE	Opera
	59	186	59
	107	187	61
	109	189	45
	188	188	44
	190	190	46
	191	191	47
	220	220	92

## keypress

**Событие `keypress` возникает сразу после `keydown`, если нажата *символьная* клавиша, т.е. нажатие приводит к появлению символа.**

Любые буквы, цифры генерируют `keypress`. Управляющие клавиши, такие как `Ctrl`, `Shift`, `F1`, `F2`.. — `keypress` не генерируют.

Событие `keypress` позволяет получить *код символа*. В отличие от скан-кода, он специфичен именно для символа и различен для "z" и "я".

Код символа хранится в свойствах: `charCode` и `which`. Здесь скрывается целое «гнездо» кросс-браузерных несовместимостей, разбираться с которыми нет никакого смысла — запомнить сложно, а на практике нужна лишь одна «правильная» функция, позволяющая получить код везде.

## Получение символа в `keypress`

**Кросс-браузерная функция для получения символа из события `keypress`:**

```
01 // event.type должен быть keypress
02 function getChar(event) {
03     if (event.which == null) { // IE
04         if (event.keyCode < 32) return null; // спец. символ
05         return String.fromCharCode(event.keyCode)
06     }
07
08     if (event.which!=0 && event.charCode!=0) { // все кроме IE
09         if (event.which < 32) return null; // спец. символ
10         return String.fromCharCode(event.which); // остальные
11     }
12
13     return null; // спец. символ
14 }
```

Для общей информации — вот основные браузерные особенности, учтённые в `getChar(event)`:



1. Во всех браузерах, кроме IE, у события `keypress` есть свойство `charCode`, которое содержит код символа.
2. При этом у Opera есть некоторые баги со специальными клавишами. Для некоторых из них она «забывает» указать `charCode`, например, для «Backspace». А другие браузеры в этом случае код указывают.
3. Браузер IE для `keypress` не устанавливает `charCode`. Вместо этого он записывает код символа в `keyCode` (в `keydown/keyup` там хранится скан-код).

Также, посмотрите — в функции выше используется проверка `if(event.which!=0)`, а не более короткая `if(event.which)`. Это не случайно! При `event.which=null` первое сравнение даст `true`, а второе — `false`.

В действии:

```
<input onkeypress="this.nextSibling.innerHTML = getChar(event)+''><b></b>
```

Наберите символ, он будет справа



### Неправильный getChar

В сети вы можете найти другую функцию того же назначения:

```
function getChar(event) {  
    return String.fromCharCode(event.keyCode || event.charCode);  
}
```

Она работает неверно для многих специальных клавиш, потому что не фильтрует их. Например, она возвращает символ амперсанда "&", когда нажата клавиша «Стрелка Вверх». Кроме того, она глючит на Backspace в Opera.



### Свойства-модификаторы

Как и у других событий, связанных с пользовательским вводом, поддерживаются свойства `shiftKey`, `ctrlKey`, `altKey` и `metaKey`.

Они установлены в `true`, если нажаты клавиши-модификаторы — соответственно, `Shift`, `Ctrl`, `Alt` и `Cmd` для Mac.

## Отмена пользовательского ввода

Появление символа можно предотвратить, если отменить действие браузера на `keydown/keypress`:

Попробуйте что-нибудь ввести в этих полях:

```
<input onkeydown="return false" type="text" size="30">  
<input onkeypress="return false" type="text" size="30">
```

Попробуйте что-нибудь ввести в этих полях (не получится):

В IE и Safari/Chrome отмена действия браузера при keydown также предотвращает само событие keypress.

Некоторые мобильные устройства не генерируют keypress/keydown, а сразу вставляют текст в поле. Отменить ввод для них таким образом нельзя. В следующих главах мы разберём [события для элементов форм \[42\]](#), которые позволяют реагировать на такой ввод.



При keydown/keypress значение ещё старое

На момент срабатывания keydown/keypress *клавиша ещё не обработана браузером.*

Поэтому в обработке значение `input.value` — старое, т.е. до ввода. Это можно увидеть в примере ниже. Вводите символы `abcd..`, а справа будет текущее `input.value`: `abc..`

А что, если мы хотим обработать `input.value` именно после ввода? Самое простое решение — использовать событие `keyup`, либо запланировать обработчик через `setTimeout(.., 0)`.

## Отмена специальных действий

Отменять можно не только символ, а любое действие клавиш.

Например, при отмене `Backspace` — символ не удалится. А при отмене `Page Down` — страница не прокрутится.

Конечно же, есть действия, которые в принципе нельзя отменить, в первую очередь — те, которые происходят на уровне операционной системы. Комбинация `Alt+F4` инициирует закрытие браузера в Windows, что бы мы ни делали в JavaScript.

## Демо: перевод символа в верхний регистр

В примере ниже действие браузера отменяется с помощью `return false`, а вместо него в `input` добавляется значение в верхнем регистре:

```
01 <input id="my" type="text" size="2">
02 <script>
03 document.getElementById('my').onkeypress = function(e) {
04     e = e || window.event;
05
06     // спец. сочетание - не обрабатываем
07     if (e.ctrlKey || e.altKey || e.metaKey) return;
08
09     var char = getChar(event || window.event);
10
11     if (!char) return; // спец. символ - не обрабатываем
12
13     this.value = char.toUpperCase();
14
15     return false;
16 }
17 </script>
```

В действии:

## Порядок наступления событий

События keydown/keyup возникают всегда. А keypress — по-разному.

Есть три основных варианта нажатия клавиш. Они перечислены в следующей таблице.

Тип	Нажать на стенде	События	Описание
Печатные клавиши		keydown keypress – keyup	<p>Нажатие вызывает keydown и keypress. Когда клавишу отпускают, срабатывает keyup.</p> <p>Исключение — CapsLock под MacOS.</p> <div><p>Обработка CapsLock под MacOS глючит невероятно:</p><ul style="list-style-type: none"><li>➔ В Safari/Chrome: при включении только keydown, при отключении только keyup.</li><li>➔ В Firefox: при включении и отключении только keydown.</li><li>➔ В Opera: вообще нет события.</li></ul><p>Попробуйте на тестовом стенде.</p></div>
Специальные клавиши		keydown – keyup	<p>Нажатие вызывает keydown. Когда клавишу отпускают, срабатывает keyup.</p> <p>Некоторые браузеры могут генерировать и keypress, например IE для . Попробуйте нажать  на тестовом стенде в разных браузерах — и увидите: в IE keypress есть, а в остальных — нет.</p>
Сочетания с печатной клавишей		keydown keypress? – keyup	<p>Браузеры под Windows — не генерируют keypress, браузеры под MacOS — генерируют.</p> <p>Если сочетание вызвало браузерный диалог («Сохранить файл», «Открыть» и т.п.), то keypress/keyup не будет.</p> <p>В <i>русской</i> раскладке под MacOS в браузерах Safari/Firefox сочетания с  не дают keyCode (это ошибка в браузерах). При этом для Cmd происходит keypress с <i>английским</i> кодом символа. Для проверки нажмите на тестовом стенде выше  +  или  + .</p> <p><b>Таким образом, если использовать для обработки сочетаний лишь keydown, то в русской раскладке под MacOS в Safari/Firefox будут проблемы.</b></p> <p>Для сочетаний с  эти проблемы можно обойти, добавив обработку keypress.</p>

## Автоповтор

При долгом нажатии клавиши возникает *автоповтор*. По стандарту, должны генерироваться многократные события

keydown (+keypress), и вдобавок стоять свойство [repeat=true \[43\]](#) у объекта события.

То есть поток событий должен быть такой:

```
1 keydown
2 keypress
3 keydown
4 keypress
5 ..повторяется, пока клавиша не отжата...
6 keyup
```

Однако в реальности на это полагаться нельзя. На момент написания статьи, под Firefox(Linux) генерируется и keyup:

```
1 keydown
2 keypress
3 keyup
4 keydown
5 keypress
6 keyup
7 ..повторяется, пока клавиша не отжата...
8 keyup
```

...А Chrome под MacOS не генерирует keypress. В общем, «зоопарк». Полагаться можно только на keydown при каждом автонажатии и keyup по отпусканию клавиши.

## Итого, рецепты

Если обобщить данные из этой таблицы и статьи, то можно сделать ряд выводов:

1. Для реализации горячих клавиш, включая сочетания — используем keydown.  
Проблемы здесь также возникают в MacOS. Для обработки сочетаний с **Cmd** в русской раскладке требуется еще и keypress.  
Попробуйте сами на стенде и увидите.
2. Если нужен именно символ — используем keypress. При этом функция `getChar` позволит получить символ и отфильтровать лишние срабатывания.
3. Ловля CapsLock глючит под MacOS. Её можно организовать при помощи проверки `navigator.userAgent` и `navigator.platform` (кроме Opera).



На момент срабатывания keypress/keydown символ ещё не введён

Например, при работе с полем ввода в обработчике `input.onkeydown` можно получить введённый символ, но не новое значение `input`, так как браузер ещё не добавил его. И не добавит, если обработчик возвратит `false`.

Чтобы обработать полное значение `input`, можно запланировать вызов функции через [setTimeout\(handler, 0\) \[44\]](#).

Для работы с INPUT и рядом других элементов, существуют [события формы \[45\]](#). Например `oninput`, которое срабатывает после любого ввода в поле. Оно надёжнее, так как происходит в том числе после вставки значения мышкой, вообще без клавиатуры.

# Формы: свойства элементов

Рассмотрим основные элементы формы и средства для работы с ними.

## label

Элемент `label` — один из самых важных в формах. Его иногда забывают, поэтому мы рассмотрим его первым.

Его основная особенность — клик на `label` засчитывается как клик на элементе формы, к которому он относится. Это позволяет посетителям кликать на большой красивой метке, а не на маленьком квадратике `input type="checkbox" (radio)`. Конечно, это очень удобно.

У него два вида использования:

1. Дать метке атрибут `for`, равный `id` соответствующего `input`:

```
01 <table>
02   <tr>
03     <td><label for="agree">Согласен с правилами</label></td>
04     <td><input id="agree" type="checkbox"></td>
05   </tr>
06   <tr>
07     <td><label for="not-a-robot">Я не робот</label></td>
08     <td><input id="not-a-robot" type="checkbox"></td>
09   </tr>
10 </table>
```

Согласен с правилами	<input type="checkbox"/>
Я не робот	<input type="checkbox"/>

2. Завернуть элемент в `label`. В этом случае можно обойтись без дополнительных атрибутов:

```
<label>Клихни меня <input type="checkbox"></label>
```

Клихни меня	<input type="checkbox"/>
-------------	--------------------------

Для других элементов формы, кроме `input type="checkbox/radio"`, клик на метке вызывает фокусировку или клик на элементе:

```

1 <label>Клик на мне - это фокус на селекте:
2 <select><option>1</option><option>2</option></select>
3 </label>
4 <br>
5 <label>Клик на мне равен клику на кнопке:
6 <button onclick="alert(123)">alert(123)</button>
7 </label>

```

Клик на мне - это фокус на селекте:

Клик на мне равен клику на кнопке:

## input, textarea

Для большинства типов input значение доступно на чтение-запись в value:

```

input.value = "Новое значение";
textarea.value = "Новый текст";

```



Не используйте `textarea.innerHTML`

Для элементов `textarea` также доступно свойство `innerHTML`, но лучше им не пользоваться: оно хранит только HTML, изначально присутствовавший в элементе. Кроме того, оно не преобразует HTML-entities.

Например, сравните значения `value/innerHTML` до того, как что-то ввели (будут разные!) и после (`innerHTML` не поменяется):

```

1 <textarea> &lt;a&gt; </textarea>
2 <script>
3   var area = document.body.children[0];
4 </script>
5
6 <button onclick="alert(area.value); alert(area.innerHTML);">
7   Вывести value и innerHTML
8 </button>

```

<a>

Вывести value и innerHTML

## input type="checkbox", input type="radio"

Для этих элементов можно узнать или установить текущее «отмеченное» состояние.

Оно находится в свойстве `checked` (`true/false`).

```
if (input.checked) {  
  alert("Чекбокс выбран");  
}
```

Обратите внимание на разницу значения атрибута и свойства. Атрибут имеет значение «как указано в HTML», а свойство — логическое, в соответствии со стандартом.

```
1 <input type="checkbox" checked>  
2  
3 <script>  
4 var input = document.body.children[0];  
5  
6 alert(input.checked); // true  
7  
8 alert(input.getAttribute('checked')); // пустая строка  
9 </script>
```

## select, option

Элементы типа `select`, как и `input`, поддерживают свойство `value`.

Он обычно возвращает значение (`value`) выбранной опции, ну а в случае `<select multiple>` — значение первой из них.

```
var selectedOptionValue = select.value;
```

**Элемент селекта в JavaScript можно выбрать двумя путями: поставив значение `select.value`, либо установив свойство `select.selectedIndex`:**

```
select.selectedIndex = 0; // первый элемент
```

Установка `selectedIndex = -1` очистит выбор.

**Опции доступны через `select.options`.**

Если `select` допускает множественный выбор (атрибут `multiple`), то значения можно получить/установить, сделав цикл по `select.options`:

```

01 <form name="form">
02   <select name="genre" multiple>
03     <option value="blues" selected>Мягкий блюз</option>
04     <option value="rock" selected>Жёсткий рок</option>
05     <option value="classic">Классика</option>
06   </select>
07 </form>
08
09 <script>
10 var form = document.body.children[0];
11 var select = form.elements.genre;
12
13 for (var i=0; i<select.options.length; i++) {
14   var option = select.options[i];
15   if(option.selected) {
16     alert(option.value);
17   }
18 }
19 </script>

```

Спецификация: [HTMLSelectElement \[46\]](#) .



### new Option

В стандарте [the Option Element \[47\]](#) есть любопытный короткий синтаксис для создания элемента с тегом option:

```
option = new Option( text, value, defaultSelected, selected);
```

Параметры:

- ➡ text — содержимое,
- ➡ value — значение,
- ➡ defaultSelected и selected поставьте в true, чтобы сделать элемент выбранным.

Его можно использовать вместо `document.createElement('option')`, например:

```
var option = new Option("Текст", "value");
// создаст <option value="value">Текст</option>
```

Такой же элемент, но выбранный:

```
var option = new Option("Текст", "value", true, true);
```





## Дополнительные свойства option

У элементов `option` также есть особые свойства, которые могут оказаться полезными (см. [The Option Element \[48\]](#)):

### **selected**

выбрана ли опция

### **index**

номер опции в списке селекта

### **text**

Текстовое содержимое опции (то, что видит посетитель).

## Практика

## Формы: события "change", "input", "propertychange"

На элементах формы происходят события клавиатуры и мыши, но есть и несколько других, особенных событий.

### Событие change

Событие `change` происходит при изменении значения элемента формы. По [стандарту \[49\]](#), оно должно происходить *после того, как элемент теряет фокус*.

В реальности оно так и происходит для текстовых элементов.

Например, пока вы набираете что-то в текстовом поле ниже — события нет. Но как только вы уведёте фокус на другой элемент, например, нажмёте кнопку — произойдет событие `onchange`.

```
<input type="text" onchange="alert(this.value)">  
<input type="button" value="Кнопка">
```

Для остальных же элементов: `select`, `input type=checkbox/radio` наблюдается кросс-браузерный зоопарк. Браузеры стараются генерировать событие при выборе значения еще до потери фокуса элементом.



## Зоопарк для change на input type=checkbox/radio

- Элементы checkbox/radio при изменении мышью инициируют событие *тут же* везде, кроме IE<9. В IE<9 они ждут потери фокуса.
- Элемент select также генерирует событие *тут же* при выборе значения везде, кроме Opera и IE<9. В Opera/IE<9 они также генерируются при переборе значений с клавиатуры клавишами вверх-вниз.

Вы можете увидеть это самостоятельно в примере ниже:

```
1 <select onchange="alert(this.value)">
2   <option value="1">1</option>
3   <option value="2">2</option>
4   <option value="3">3</option>
5 </select>
6 <input type="checkbox" onchange="alert(this.checked)">
7 <input type="button" value="Кнопка">
```

Выводы:

- Для того, чтобы видеть изменение checkbox/radio тут же — в IE<9 нужно повесить обработчик на событие `click` (оно произойдет и при изменении значения с клавиатуры) или воспользоваться событием `propertychange`, описанным ниже.
- С `select` обычно ничего такого не делают. В Opera/IE<9 будет работать немного по-другому при использовании клавиатуры для выбора.

## Событие `propertychange`

Это событие происходит только в IE, при любом изменении свойства. Оно позволяет отлавливать изменение тут же.

Если поставить его на checkbox в IE<9, то получится «правильное» событие `change`:

```

01 <input type="checkbox"> Чекбокс с "onchange", работающим везде одинаково
02 <script>
03 var checkbox = document.body.children[0];
04
05 if("onpropertychange" in checkbox) {
06     // если поддерживается (IE)
07     checkbox.onpropertychange = function() {
08         if (event.propertyName == "checked") { // имя свойства
09             alert(checkbox.checked);
10         }
11     };
12 } else {
13     // остальные браузеры
14     checkbox.onchange = function() {
15         alert(checkbox.checked);
16     };
17 }
18 </script>

```

☐ Чекбокс с "onchange", работающим везде одинаково

Это событие также срабатывает при изменении значения текстового элемента, но по каким-то причинам, известным лишь в Индии - кроме удаления символов.

Попробуйте набрать что-то в текстовом поле, а затем удалить. При удалении значение в span не будет обновляться.

Пример — *только для IE*:

```

01 <input type="text"> onpropertychange: <span id="result"></span>
02 <script>
03 var input = document.body.children[0];
04
05 input.onpropertychange = function() {
06     if (event.propertyName == "value") {
07         document.getElementById('result').innerHTML = input.value;
08     }
09 }
10 </script>

```

onpropertychange:



## События изменения DOM

В браузерах, за исключением IE<9 и Safari/Chrome, существует аналогичное событие [DOMAttrModified \[50\]](#), а также ряд других событий на изменение и добавление узлов.

Они описаны в стандарте [Mutation Events \[51\]](#). Но эта часть стандарта реализована в браузерах частично, с многочисленными ошибками, и объявлена устаревшей.

На практике такие события нужны, в основном, при отладке. А современные отладчики (Chrome/Safari, Firefox) позволяют вставить точку останова при изменении DOM и без этих событий.

## Событие input

Событие input срабатывает *тут же* при изменении значения текстового элемента и поддерживается всеми браузерами, кроме IE<9.

В IE9 оно поддерживается частично, а именно — *не возникает при удалении символов*.

Пример использования (не работает в IE<9):

```
1 <input type="text"> oninput: <span id="result"></span>
2 <script>
3   var input = document.body.children[0];
4
5   input.oninput = function() {
6     document.getElementById('result').innerHTML = input.value;
7   }
8 </script>
```

oninput:

## События cut, copy, paste

Эти события используются редко, но иногда бывают полезны. Они происходят при вырезании/вставке/копировании значения в поле.

При этом действие браузера можно отменить, как это сделано в примере ниже:

```

01 <input type="text"> event: <span id="result"></span>
02 <script>
03 var input = document.body.children[0];
04
05 input.oncut = input.oncopy = input.onpaste = function(e) {
06     e = e || event;
07     document.getElementById('result').innerHTML = e.type + ' ' + input.value;
08     return false;
09 }
10 </script>

```

event:

Эти события не дают доступ к данным, которые вставляются.

Также существуют события `beforecut`/`beforecopy`/`beforepaste`. Они интересны тем, что `return false` из `beforecut`/`beforepaste` позволяет включить вырезание/вставку для нередатируемых элементов. В дальнейшем эти действия можно будет обработать в событиях `cut`/`paste`

К сожалению, они нормально работают только в IE.

```

<div onbeforecut="return false" onbeforepaste="return false">
Для этого DIV включены команды меню Вырезать/Вставить (IE)
</div>

```

Для этого DIV включены команды меню Вырезать/Вставить (IE)

## Пример: поле для СМС

Сделаем поле для СМС, рядом с которым должно показываться число символов, обновляющееся при каждом изменении поля.

Как такое реализовать?

Событие `input` придумано специально для таких случаев. И оно идеально решает задачу во всех браузерах, кроме IE.

Но в IE<9 это событие не поддерживается, а в IE9 не работает при удалении. Что делать?

Вместо `input` для IE можно использовать `propertychange`, но оно не работает при удалении символов.

Впрочем, это не так страшно. Удаление нажатием `Delete`/`BackSpace` замечательно поймает `keyup`. А вот удаление командой «вырезать» из меню — сможет отловить лишь `oncut`.

Получается вот такая комбинация:

```
01 <input type="text"> символов: <span id="result"></span>
02 <script>
03 var input = document.body.children[0];
04
05 function showCount() {
06     document.getElementById('result').innerHTML = input.value.length;
07 }
08
09 input.oncut = input.onkeyup = input.oninput = showCount;
10 input.onpropertychange = function() {
11     if (event.propertyName == "value") showCount();
12 }
13 </script>
```

СИМВОЛОВ:

Теперь всё отлично, только вот событие cut срабатывает *до вырезания*, поэтому значение поля в нём еще не обновлено. В результате при вырезании мышью в IE количество символов не изменится. Как исправить этот маленький недостаток, мы увидим далее, при рассмотрении трюка [setTimeout\(.., 0\)](#) [52].

### Итого

События изменения данных:

Событие	Описание	Особенности
change	Изменение значения любого элемента формы. Для текстовых элементов срабатывает при потере фокуса.  Для select/checkbox/radio браузеры иницируют его при выборе.	Для Opera/IE<9 срабатывает при <i>переборе</i> значений select с клавиатуры.  Для IE<9 на чекбоксах ждёт потери фокуса.
input	Событие срабатывает только на текстовых элементах. Оно не ждет потери фокуса в отличие от change.	В IE<9 не поддерживается, в IE9 не работает при удалении.
propertychange	Только для IE. Универсальное событие для отслеживания изменения свойств элементов. Имя изменённого свойства содержится в event.propertyName.  Аналог в других браузерах, кроме Safari/Chrome — <a href="#">DOMAttrModified</a> [53].	Не срабатывает при удалении символов из текстового поля.
cut/copy/paste	Срабатывают при вставке/копировании/удалении текста. В них можно отменить действие браузера, и тогда вставке/копирования/удаления не произойдёт.	Вставляемое значение получить нельзя: на момент срабатывания события в элементе всё ещё <i>старое</i> значение.

Для того, чтобы на чекбоксах в IE<9 отлавливать изменение тут же — можно повесить обработчик на click или использовать событие propertychange в дополнение к change.

**В IE<9 события change(и аналогичные) не всплывают.**

Можно увидеть это на следующем примере:

```

1 <form oninput="log(event)" onchange="log(event)" onpropertychange="log(event)" oncut="log(event)">
2 <input type="text"> <span id="log"></span>
3 </form>
4
5 <script>
6 function log(event) {
7   document.getElementById('log').innerHTML += ' '+event.type;
8 }
9 </script>

```

Для всех браузеров, кроме IE<9 события будут ловиться на FORM и отображаться справа от INPUT.

## Формы: метод и событие "submit"

JavaScript может делать с формами почти всё, что угодно. Единственное, что он не может — это самостоятельно заполнять поля `<input type="file">`, без инициативы пользователя.

Наиболее частое применение — это *валидация* (проверка) формы перед отправкой. Для неё существует событие `submit`.

### Событие `submit`

Чтобы отправить форму на сервер, у посетителя есть два способа:

1. **Первый** — это нажать кнопку `<input type="submit">` или `<input type="image">`.
2. **Второй** — нажать **Enter** на поле, когда оно в фокусе.

В обоих случаях будет сгенерировано событие `submit`. Если отменить в нём действие браузера, то форма не отправится на сервер.

Как правило, это используется для проверки данных в форме. Если что-то не так — выводится ошибка и отправка отменяется.

Посмотрим это на живом примере. Оба способа выдадут сообщение, форма не будет отправлена:

```

1 <form onsubmit="alert('submit!');return false">
2   Первый: Enter в текстовом поле <input type="text" value="Текст"><br>
3   Второй: Нажать на "Отправить": <input type="submit" value="Отправить">
4 </form>
5
6 </button>

```

Первый: Enter в текстовом поле

Второй: Нажать на "Отправить":

Ожидаемое поведение:

1. Перейдите в текстовое поле и нажмите Enter, будет событие, но форма не отправится на сервер благодаря `return false` в обработчике.
2. То же самое произойдет при клике на `<input type="submit">`.



#### Взаимосвязь событий `submit` и `click`

При отправке формы путём нажатия Enter на текстовом поле, на элементе `<input type="submit">` везде, кроме IE<9, генерируется событие `click`.

Это довольно забавно, учитывая что клика-то и не было.

```
1 <form onsubmit="alert('submit');return false">
2   <input type="text" size="30" value="Нажми здесь Enter">
3   <input type="submit" value="Submit" onclick="alert('click')">
4 </form>
```



#### В IE<9 событие `submit` не всплывает

В IE<9 событие `submit` не всплывает. Впрочем, если вешать обработчик `submit` на сам элемент формы, без использования делегирования, то это не создаёт проблем.

## Метод `submit`

Чтобы отправить форму на сервер из JavaScript — нужно вызвать на элементе формы метод `form.submit()`.

При этом само событие `submit` не генерируется. Предполагается, что если программист вызывает метод `form.submit()`, то он выполнил все проверки.

Это используют, в частности, для искусственной генерации и отправки формы.

## События и методы "focus/blur"

Событие `focus` вызывается тогда, когда пользователь *фокусируется на элементе*, например кликает на `INPUT`.

Событие `blur` вызывается, когда фокус исчезает, например посетитель кликает на другом месте экрана.



По умолчанию не все элементы поддерживают эти события. Перечень элементов немного разнится от браузера к браузеру, например, список для IE описан в MSDN [54]. Но есть элементы, которые поддерживают фокусировку всегда. В первую очередь это те, с которыми посетитель может взаимодействовать: INPUT, SELECT, А и т.п.

С другой стороны, элементы для форматирования, такие как DIV, SPAN, TABLE и другие — по умолчанию не поддерживают это событие, т.е. на них нельзя «фокусироваться». Впрочем, существует способ включить фокусировку и для них.

## Пример использования focus/blur

Поле из примера ниже при фокусировке(кликните на него) становится подсвеченным, а при снятии фокуса(клик в другом месте) теряет подсветку:

```
01 <input type="text">
02
03 <style>
04 .highlight { background: #FA6; }
05 </style>
06
07 <script>
08 var input = document.getElementsByTagName('input')[0];
09
10 input.onfocus = function() {
11     this.className = 'highlight';
12 }
13
14 input.onblur = function() {
15     this.className = '';
16 }
17 </script>
```



Этот пример — учебный. В современных браузерах для этого можно обойтись и без JavaScript.

Текущий элемент, на котором фокус, доступен как `document.activeElement`.

## HTML5 и CSS3 вместо focus/blur

Некоторые задачи могут быть решены средствами современного HTML/CSS. На текущий момент единственный браузер, который не очень поддерживает соответствующие свойства — это IE<10.

Поэтому, прежде чем переходить к JavaScript, мы рассмотрим три примера.

### Подсветка при фокусировке

Стилизация полей ввода при фокусировке во всех браузерах, кроме IE<8, может быть решена средствами CSS (CSS2.1), а именно — селектором `:focus`:

```

1 <style>
2 input:focus {
3     background: #FA6;
4     outline: none; /* убрать рамку */
5 }
6 </style>
7 <input type="text">
8
9 <p>Селектор :focus выделит элемент при фокусировке на нем (кроме IE<8) и уберёт рамку, которой браузер выделяет этот элемент по умолчанию.</p>

```



Селектор :focus выделит элемент при фокусировке на нем (кроме IE<8) и уберёт рамку, которой браузер выделяет этот элемент по умолчанию.

В IE (включая более старые) скрыть фокус также может установка специального атрибута [hideFocus \[55\]](#) .

## Автофокус

При загрузке страницы, если на ней существует элемент с атрибутом autofocus — браузер автоматически фокусируется на этом элементе. Работает во всех браузерах, кроме IE<10.

```

1 | <input type="text" name="search" autofocus>

```

Кросс-браузерно можно сделать вызовом JavaScript:

```

1 | <input type="text" name="search">
2 | <script>
3 |     document.getElementsByName('search')[0].focus();
4 | </script>

```

## Плейсхолдер

*Плейсхолдер* — это значение-подсказка внутри INPUT, которое автоматически исчезает при фокусировке и существует, пока посетитель не начал вводить текст.

Во всех браузерах, кроме IE<10, это реализуется специальным атрибутом placeholder:

```
<input type="text" placeholder="E-mail">
```

Работает везде, кроме IE до 10



В некоторых браузерах этот текст можно стилизовать:

```

1 <style>
2 .my::-webkit-input-placeholder, .my::-moz-placeholder {
3   color: red;
4   font-style: italic;
5 }
6 </style>
7
8 <input class="my" type="text" placeholder="E-mail">
9 Стилизуется в Safari/Chrome/FF

```

Стилизуется в Safari/Chrome/FF

## Метод `elem.focus()`

На элементе, который поддерживает фокусировку, можно сфокусироваться программно.

Например, можно показать форму и тут же сфокусироваться на ее поле, чтобы посетитель мог вводить текст:

```

01 <form style="display:none" name="form">
02   Искать <input style="text" name="search">
03 </form>
04
05 <input type="button" value="Показать форму поиска" onclick="showForm()">
06
07 <script>
08 function showForm() {
09   var form = document.forms.form;
10   form.style.display = 'block';
11   form.elements.search.focus();
12 }
13 </script>

```

## Разрешаем фокус на любом элементе: `tabindex`

Несмотря на то, что не все элементы могут иметь фокус по умолчанию, существует атрибут `tabindex`, который позволяет устранить подобную несправедливость.

Когда вы присваиваете `tabindex="число"` элементу:

- ➡ Появляется возможность на нем сфокусироваться.
- ➡ Пользователь при нажатии клавиши «Tab» переходит от элемента с наименьшим положительным значением `tabindex` к следующему.

Исключением является специальное значение `tabindex="0"`, которое делает элемент всегда последним.

→ Значение `tabindex=-1` означает, на элементе можно фокусироваться, но только программно. Клавиша «Tab» будет его игнорировать. Сработает только метод `focus()`.

В примере ниже есть список элементов. Кликните на любой из них и нажмите «tab».

```
01 Кликните на первый элемент списка и нажмите Tab. Внимание! Дальнейшие нажатия Tab могут вывести за границы
    iframe'a с примером.
02 <ul>
03 <li tabindex="1" onfocus="showFocus(this)">Один</li>
04 <li tabindex="0" onfocus="showFocus(this)">Ноль</li>
05 <li tabindex="2" onfocus="showFocus(this)">Два</li>
06 <li tabindex="-1" onfocus="showFocus(this)">Минус один</li>
07 </ul>
08
09 <style>
10   li { cursor: pointer; }
11   :focus { border: 1px dashed green; outline: 0; }
12 </style>
```

Кликните на первый элемент списка и нажмите Tab. Внимание! Дальнейшие нажатия Tab могут вывести за границы iframe'a с примером.

- Один
- Ноль
- Два
- Минус один

Порядок перемещения по клавише «Tab» в примере выше должен быть таким: 1 - 2 - 0 (ноль всегда последний). Продвинутые пользователи частенько используют «Tab» для навигации, и ваше хорошее отношение к ним будет вознаграждено 😊

Некоторые браузеры выделяют элемент с фокусом пунктирной рамочкой. В примере выше это выделение убрано при помощи CSS-свойства `outline: 0` и заменено собственной рамкой.

## Событие `blur` и метод `elem.blur()`

**Событие `blur` — это противоположность `focus`. Оно срабатывает, когда элемент теряет фокус.**

Метод `elem.blur()` заставляет элемент потерять фокус, если таковой имеется.

Зачастую это событие используется для проверки («валидации») значения.

В следующем примере поле ввода проходит валидацию при потере фокуса. Наберите что-нибудь в поле «возраст» примера ниже и завершите ввод, нажав Tab или кликнув в другое место страницы. Введенное значение будет автоматически проверено:

```

01 <label>Введите ваш возраст: <input type="text" name="age"></label>
02
03 <div id="error">здесь будет ошибка, если вы введёте не число</div>
04 <style> .error { background: red; } </style>
05
06 <script>
07 var input = document.getElementsByName('age')[0];
08 var errorHandler = document.getElementById('error');
09
10 input.onblur = function() {
11     if (isNaN(this.value)) { // введено не число
12         // показать ошибку
13         this.className = 'error';
14         errorHandler.innerHTML = 'Вы ввели не число. Исправьте, пожалуйста'
15     }
16 }
17
18 input.onfocus = function() {
19     // сбросить состояние "ошибка", если оно есть
20     if (this.className == 'error') {
21         this.className = '';
22         errorHandler.innerHTML = '';
23     }
24 }
25 </script>

```

Введите ваш возраст:

здесь будет ошибка, если вы введёте не число

Обратите внимание, здесь кроме `blur` использован также обработчик события `focus`, чтобы повторная фокусировка снимала состояние «ошибки».

Конечно, проверка по каждому клику будет здесь уместнее, но иногда значение следует фиксировать только по окончании ввода. Для этого и служит событие `blur`.

## Делегирование с `focus/blur`

События `focus` и `blur` не всплывают.

Таким образом, мы не можем использовать делегирование с ними. Например, мы не можем сделать так, чтобы при фокусировке в форме она подсвечивалась:

```
1 <form onfocus="this.className='focused'">
2   <input type="text" name="name" value="Ваше имя">
3   <input type="text" name="surname" value="Ваша фамилия">
4 </form>
5
6 <style> .focused { border: 2px solid red; } </style>
```

В примере выше стоит обработчик `onfocus` на форме, но он не работает, т.к. при фокусировке на любом `INPUT` событие `focus` срабатывает только на этом элементе и не всплывает вверх.

Что делать? Неужели мы должны присваивать обработчик каждому полю или все-таки есть возможность использовать делегирование?

## Всплывающие альтернативы: `focusin/focusout`

---

➡ В современных стандартах есть события `focusin/focusout` — то же самое, что и `focus/blur`, только они всплывают.

Поддерживается в IE6+, а также в Safari/Chrome и Opera. Во всех браузерах, кроме IE, должны быть назначены не через `on-` свойство, а при помощи `elem.addEventListener`.

Единственный современный браузер, который их не поддерживает — это Firefox.

➡ Во всех браузерах, кроме IE<9, события `focus/blur` можно поймать *на стадии захвата*, то есть указав последний аргумент `true` для `addEventListener` вместо обычного `false`.

Как правило, используют две ветки кода: `focus/blur` ловят на фазе захвата везде, кроме старых IE, в которых используют `focusin/focusout`.

Подсветка формы в примере ниже работает во всех браузерах.

```

01 <form name="form">
02   <input type="text" name="name" value="Ваше имя">
03   <input type="text" name="surname" value="Ваша фамилия">
04 </form>
05 <style>
06   .focused { border: 2px solid red; }
07 </style>
08
09 <script>
10 function onFormFocus() { this.className = 'focused'; }
11 function onFormBlur() { this.className = ''; }
12
13 var form = document.forms.form;
14
15 if (form.addEventListener) {
16   form.addEventListener('focus', onFormFocus, true);
17   form.addEventListener('blur', onFormBlur, true);
18 } else { // IE<9
19   form.onfocusin = onFormFocus;
20   form.onfocusout = onFormBlur;
21 }
22 </script>

```



## Итого

Есть 3 способа сфокусироваться на элементе:

1. Кликнуть мышью
2. Перейти на него клавишей Tab
3. Вызвать `elem.focus()`

➡ По умолчанию многие элементы не могут получить фокус. Например, если вы кликните по DIV, то фокусировка на нем не произойдет.

Но это можно изменить, если поставить элементу атрибут `tabIndex`. Этот атрибут также дает возможность контролировать порядок перехода при нажатии «Tab».

➡ События `focus` и `blur` не всплывают.

В случаях, когда всплытие необходимо, нужно использовать `focusin/focusout` в IE и `focus/blur` на стадии захвата для других браузеров.

## Событие "onscroll"

Событие происходит, когда элемент прокручивается. Его могут генерировать только прокручиваемые элементы.

При прокрутке окна срабатывает событие `window.onscroll`.

Например, следующая функция при прокрутке окна выдает количество прокрученных пикселей:

```
1 window.onscroll = function() {  
2   var scrolled = window.pageYOffset || document.documentElement.scrollTop;  
3   document.getElementById('showScroll').innerHTML = scrolled + 'px';  
4 }
```

В действии:

Текущая прокрутка = **прокрутите окно**

Более подробно о том, как получить значение прокрутки или прокрутить документ — читайте в статье [Размеры и прокрутка элементов \[56\]](#).

## События "onload", "onbeforeunload" и "onerror"

Браузер позволяет отслеживать загрузку внешних ресурсов — скриптов, ифреймов, картинок и других.

Для этого есть два события:

- ➡ `onload` — если загрузка успешна.
- ➡ `onerror` — если при загрузке произошла ошибка.

Кроме того, обработчик `window.onerror` вызывается при ошибках JavaScript, которые не были пойманы `try...catch`.

## Загрузка SCRIPT

Рассмотрим задачу. В браузере работает сложный интерфейс и, чтобы создать очередной компонент, нужно загрузить его скрипт с сервера.

Подгрузить внешний скрипт — достаточно просто:

```
var script = document.createElement('script');  
script.src = "my.js";  
document.documentElement.appendChild(script);
```

...Но, после подгрузки нужно запустить функцию создания компонента. Как определить, что скрипт загрузился?

### onload

Здесь нам и поможет событие `onload`. Оно сработает, когда скрипт загрузился и выполнился.

```
1 var script = document.createElement('script');  
2 script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js";  
3 document.documentElement.appendChild(script);  
4  
5 script.onload = function() {  
6   alert(jQuery);  
7 }
```

..А что, если загрузка скрипта не удалась? Например, такого скрипта на сервере нет (ошибка 404).



## onerror

Любые ошибки загрузки (но не выполнения) скрипта отслеживаются обработчиком `onerror`:

```
1 var script = document.createElement('script');
2 script.src = "http://ajax.googleapis.com/404.js"
3 document.documentElement.appendChild(script);
4
5 script.onerror = function() {
6     alert("Ошибка: " + this.src);
7 }
```

## IE<9: onreadystatechange

Примеры выше работают во всех браузерах, кроме IE<9.

В IE для отслеживания загрузки есть другое событие: `onreadystatechange`. Оно срабатывает многократно, при каждом обновлении состояния загрузки.

Текущая стадия процесса находится в `script.readyState`:

### loading

В процессе загрузки.

### loaded

Получен ответ с сервера — скрипт или ошибка. Скрипт на фазе `loaded` может быть ещё не выполнен.

### complete

Скрипт выполнен.

Например, рабочий скрипт:

```
1 var script = document.createElement('script');
2 script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js";
3 document.documentElement.appendChild(script);
4
5 script.onreadystatechange = function() {
6     alert(this.readyState); // loading -> loaded -> complete
7 }
```

Скрипт с ошибкой:

```
1 var script = document.createElement('script');
2 script.src = "http://ajax.googleapis.com/404.js";
3 document.documentElement.appendChild(script);
4
5 script.onreadystatechange = function() {
6     alert(this.readyState); // loading -> loaded
7 }
```

**Стадии могут пропускаться.** Если скрипт в кэше браузера — он сразу даст `complete`. Вы можете увидеть это, если несколько раз запустите первый пример.

**Нет особой стадии для ошибки.** В примере выше это видно, обработка останавливается на `loaded`.

Итак, самое надёжное средство для IE<9 поймать загрузку (и ошибку) — это повесить обработчик на событие `loaded`. Так как скрипт может

быть ещё не выполнен к этому моменту, то вызов функции лучше сделать через `setTimeout(.., 0)`.

На тот случай, если скрипт прыгнет сразу на `complete` — поставим обработчик и на этой стадии тоже.

Пример вызывает `afterLoad` после загрузки скрипта. Работает только в IE:

```
01 var script = document.createElement('script');
02 script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js";
03 document.documentElement.appendChild(script);
04
05 function afterLoad() {
06     alert("Загрузка завершена: " + typeof(jQuery));
07 }
08
09 script.onreadystatechange = function() {
10     if (this.readyState == "complete") { // на случай пропуска loaded
11         afterLoad(); // (2)
12     }
13     if (this.readyState == "loaded") {
14         setTimeout(afterLoad, 0); // (1)
15     }
16     // убираем обработчик, чтобы не сработал на complete
17     this.onreadystatechange = null;
18 }
19
20 }
```

Вызов (1) выполнится при первой загрузке скрипта, а (2) — при второй, когда он уже будет в кэше, и стадия станет сразу `complete`.

Функция `afterLoad` может и не обнаружить `jQuery`, если при загрузке была ошибка, причём не важно какая — файл не найден или синтаксис скрипта ошибочен.

## Кросс-браузерное решение

---

Для кросс-браузерности поставим обработчик на все три события: `onload`, `onerror`, `onreadystatechange`.

Пример ниже выполняет функцию `afterLoad` после загрузки скрипта.

Работает везде:

```

01 var script = document.createElement('script');
02 script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js";
03 document.documentElement.appendChild(script);
04
05 function afterLoad() {
06     alert("Загрузка завершена: " + typeof(jQuery));
07 }
08
09 script.onload = script.onerror = function() {
10     if (!this.executed) { // выполнится только один раз
11         this.executed = true;
12         afterLoad();
13     }
14 };
15
16 script.onreadystatechange = function() {
17     var self = this;
18     if (this.readyState == "complete" || this.readyState == "loaded") {
19         setTimeout(function() { self.onload() }, 0); // сохранить "this" для onload
20     }
21 };

```

## Обработчики на window

### window.onerror

Обработчик [window.onerror](#) [57] вызывается в случае ошибки, выпавшей из всех блоков try...catch:

Аргументы обработчика:

**message**

Сообщение об ошибке.

**source**

Файл

**lineno**

Номер строки с ошибкой

Например:

```

01 <script>
02 window.onerror = function(message, source, lineno) {
03     alert("Ошибка:" + message + "\n" +
04         "файл:" + source + "\n" +
05         "строка:" + lineno);
06 };
07
08 </script>
09
10 <body>
11     <button onclick="alert(blablaba);">alert(blablaba)</button>
12 </body>

```

Этот документ в ифрейме:

```
alert(blablaba)
```

Реакция браузера на ошибку (запуск отладчика?) может быть отменена через `return false` из обработчика.

## window.onload

Обработчик `window.onload` срабатывает, когда загружается вся страница, включая ресурсы на ней — стили, картинки, ифреймы и т.п.

Пример ниже выведет `alert` лишь после полной загрузки окна, включая `IFRAME`, стиль, картинки:

```
1 <iframe src="http://example.com/" style="height:60px"></iframe>
2 <link type="text/css" rel="stylesheet" href="/some.css">
3 
4 <script>
5   window.onload = function() {
6     alert('Документ и все ресурсы загружены')
7   }
8 </script>
```

## window.onunload

Когда человек уходит со страницы или закрывает окно, срабатывает `window.unload`. В нём можно, например, закрыть вспомогательные рорир-окна, но отменить сам переход нельзя.

Это позволяет другое событие — `window.onbeforeunload`.

## window.onbeforeunload

Если посетитель инициировал переход на другую страницу или нажал «закрыть окно», то обработчик `onbeforeunload` может приостановить процесс и спросить подтверждение.

Для этого ему нужно вернуть строку, которую браузеры покажут посетителю, спрашивая — нужно ли переходить. Например:

```
window.onbeforeunload = function() {
  return "Данные не сохранены. Точно перейти?";
};
```



Firefox игнорирует текст, он показывает своё сообщение

Firefox игнорирует текст, а всегда показывает своё сообщение.

Это сделано в целях безопасности.

Кликните на кнопку в `IFRAME` 'е ниже, чтобы поставить обработчик, а затем по ссылке, чтобы увидеть его в действии:

## Обработчики на IMG, IFRAME, LINK

Поддержка событий для других типов ресурсов:

### IMG

Поддерживает onload/onerror во всех браузерах.

### IFRAME

Поддерживает onload во всех браузерах. Это событие срабатывает как при успешной загрузке, так и при ошибке.

### LINK (стили)

Поддерживает onload/onerror в *некоторых* браузерах. Замечательная статья про обработчик на загрузку стиля: [When is a stylesheet really loaded? \[58\]](#).

## Итого

В этой статье мы рассмотрели события onload/onerror и window.onbeforeunload.

Их можно обобщить, разделив на рецепты:

### Отловить загрузку скрипта (включая ошибку)

Ставим обработчики на onload + onerror + (для IE<9) onreadystatechange, как указано в рецепте выше

### Отловить загрузку изображения (включая ошибку)

Ставим обработчики на onload + onerror

```
1 var img = document.createElement('img');
2 img.onload = function() { alert("OK "+this.src );}
3 img.onerror = function() { alert("FAIL "+this.src );}
4 img.src = ...
```

Изображения начинают загружаться сразу при создании, не нужно их для этого вставлять в HTML.

**Чтобы работало в IE<9, src нужно ставить *после* onload/onerror.**

### Отловить загрузку IFRAME

Поддерживается только обработчик onload. Он сработает, когда IFRAME загрузится, со всеми подресурсами, а также в случае ошибки.

```
1 <iframe src="http://example.com/" style="height:60px"></iframe>
2 <script>
3   document.getElementsByTagName('iframe')[0].onload = function() {
4     alert('IFRAME загружен')
5   }
6 </script>
```

### Отловить полную загрузку страницы

Обработчик window.onload (сработает, когда окно загрузится со всеми ресурсами).

Далее мы рассмотрим событие `onDOMContentLoaded`, которое срабатывает при загрузке документа и не ждёт картинок, стилей, ифреймов.

**Отловить загрузку файла со стилями**

[When is a stylesheet really loaded? \[59\]](#).

## Событие загрузки документа "DOMContentLoaded"

---

Событие `onDOMContentLoaded` срабатывает при загрузке документа, после выполнения всех тегов `SCRIPT`.

В отличие от [window.onload \[60\]](#), оно не ждёт загрузки дополнительных ресурсов, поэтому наступает гораздо раньше.

На момент наступления `onDOMContentLoaded`, дерево DOM полностью построено и все элементы доступны. Поэтому это событие часто используют для инициализации интерфейса. Во фреймворках для него отводят специальные функции: `$(ready)`, `Ext.onReady` и т.п.

Но, как мы увидим, у него есть ряд важных особенностей, которые делают такое применение не лучшим выбором.

Событие `DOMContentLoaded` поддерживается во всех браузерах, кроме IE<9. Про поддержку аналогичного функционала в старых IE мы поговорим дальше.

Обработчик на него вешается так:

```
document.addEventListener( "DOMContentLoaded", ready, false );
```

Пример (в ифрейме):

```
1 <script>
2   function yelp() {
3       alert('DOM готов (а iframe - нет)');
4   }
5
6   document.addEventListener( "DOMContentLoaded", yelp, false );
7 </script>
8
9 <iframe src="http://example.com" height="80"></iframe>
```

## Тонкости DOMContentLoaded

В своей сути, событие `onDOMContentLoaded` — простое, как пробка. Полностью создано DOM-дерево — и вот событие.

Но с ним связан ряд существенных тонкостей.

### DOMContentLoaded и скрипты

---

Если в документе есть теги `SCRIPT`, то браузер обязан их выполнить до того, как построит DOM. Поэтому событие `DOMContentLoaded` ждёт загрузки и выполнения таких скриптов.

`DOMContentLoaded` не ждёт скриптов с атрибутами [async/defer \[61\]](#), при условии поддержки атрибута браузером, а также скриптов, добавленных через DOM.



Opera!

Браузер Opera — особый. Он ждёт любые скрипты. Это не по стандарту, но увы..

→ Побочный эффект: если на странице подключается скрипт с внешнего ресурса (реклама), и он тормозит, то инициализация интерфейсов по `DOMContentLoaded` может сильно задержаться.

## DOMContentLoaded и стили

Внешние стили никак не влияют на событие `DOMContentLoaded`. Но есть один нюанс. Если после стилия идёт скрипт, то этот скрипт обязан дожждаться, пока стиль загрузится:

```
1 <link type="text/css" rel="stylesheet" href="style.css">
2 <script>
3   // работает после загрузки style.css
4 </script>
```

Такое поведение прописано в стандарте. Его причина — скрипт может захотеть получить информацию со страницы, зависящую от стилей, например, ширину элемента.

В результате событие `DOMContentLoaded` будет также ждать стилей.



Opera!

Браузер Opera — особый. В нём скрипты не ждут стилей. Это поведение не соответствует стандарту.

## Автозаполнение

**Firefox/Chrome автозаполняют формы по `DOMContentLoaded`.**

Это означает, что если на странице есть форма для ввода логина-пароля, то браузер введёт в неё запомненные значения только по `DOMContentLoaded`.

→ Побочный эффект: если `DOMContentLoaded` ожидает множества скриптов и стилей, то посетитель не сможет воспользоваться автозаполненными значениями до полной их загрузки.

**Желательно, чтобы событие `DOMContentLoaded` происходило как можно раньше.**

Этого можно достичь, используя `defer/async`/добавление через DOM.

## IE до 9

Реализация кросс-браузерного аналога `onDOMContentLoaded` описана в статье [Кроссбраузерное событие `onDOMContentLoaded` \[62\]](#).

Альтернативой событию является вызов функции `init` из скрипта в самом конце BODY, когда основная часть DOM уже готова:

```
1 <body>
2   ...
3 </script>
4   init();
5 </script>
6 </body>
```

Это вполне кросс-браузерно. Единственная проблема — нельзя `document.body.appendChild` вызывать из такой функции в IE6.

Причина, по которой обычно предпочитают именно событие — одна: удобство. Вешается обработчик и не надо ничего писать в конец BODY.

## Проверка поддержки браузером

В этой главе мы посмотрим, как проверить поддержку браузером новых типов элементов и событий.

Современные браузеры поддерживают всё новые и новые возможности. Зачастую, они позволяют реализовать очень удобный интерфейс, но не везде поддерживаются.

Хорошее решение — проверять, есть ли нужные возможности. Если есть — использовать их, а если нет — делаем альтернативный, возможно более простой, но рабочий вариант.

### События

В современной спецификации HTML5 — гораздо больше событий, чем было когда-то. Они добавляются в браузеры постепенно. Некоторые события, например `ontouchstart`, реализованы только на платформах, поддерживающих сенсорный ввод, в частности, на планшетах и мобильных телефонах.

Большинство событий проверить очень просто. Есть рецепт, который работает в 95% случаев.

Алгоритм проверки поддержки события:

1. Получить или создать элемент, который заведомо поддерживает это событие. Например, создать `input`:
2. Проверить, существует ли атрибут для события, например `input.oninput`:

```
1 var input = document.createElement('input');
2 alert( "oninput" in input ); // true, если поддерживает
```



#### Ошибка в старых Firefox

В старых Firefox (2011 год и ранее) была ошибка, из-за которой способ выше не работал. Приходилось, в дополнение к нему, делать так:

```
1 var input = document.createElement('input');
2 input.setAttribute('oninput', 'return;');
3
4 alert( typeof input.oninput ); // "function", если событие есть
```

Для современных Firefox это уже не обязательно, способ выше работает.



**К сожалению, существуют события, поддержку которых так не проверить.**

В первую очередь это те, для которых нет соответствующего атрибута, а которые работают только через `addEventListener`.

Например, таким является событие `DOMContentLoaded`, возникающее после полной загрузки и обработке DOM документа (не ждёт картинок, ифреймов).

Такие события приходится проверять как-то иначе, и здесь универсального способа нет. Иногда можно придумать способ под конкретное событие.

Альтернатива — вообще не проверять поддержку, а просто не рассчитывать на то, что такое событие есть, дублировать функционал при помощи других, может быть не таких подходящих, но заведомо рабочих событий.

## Элементы и атрибуты

Некоторые HTML-элементы не кросс-браузерные, как и атрибуты. Например, `<input type="range">` поддерживается не везде, как и `<canvas>`.

### Проверка метода или свойства

---

Проверить поддержку элемента, например, `<canvas>` можно так:

1. Посмотреть, есть ли у элемента какие-то специальные методы или свойства. Например, можно посмотреть их в спецификации HTML5, например ввести в Google: [html 5 canvas site:w3.org](http://html5canvas.site.w3.org) [63].
2. Создать элемент и проверить наличие свойства.

```
1 // если это настоящий canvas, то у него есть метод getContext
2 var support = "getContext" in document.createElement("canvas");
3
4 alert('Поддержка: ' + support);
```

### Проверка атрибута

---

Для того, чтобы проверить, например, поддержку `input type="range"`, можно присвоить такой атрибут и посмотреть, сохранился ли он:

```
1 var input = document.createElement("input")
2 input.setAttribute("type", "range");
3
4 var support = (input.type == "range");
5
6 alert('Поддержка: ' + support);
```

## Итого

- ➡ Для проверки события — создать элемент, который его поддерживает, и проверить соответствующее свойство на `undefined`. Не проверять свойство на элементах, которые заведомо лишены этого события!
- ➡ Для проверки поддержки HTML5-элемента — создать этот элемент и проверить его специальные свойства или методы, которыми он отличается от других.
- ➡ Для проверки атрибута — присвоить этот атрибут и посмотреть, сохранился ли он в свойстве. Поддерживаемые атрибуты синхронизируются с соответствующими DOM-свойствами.

Эти способы работают примерно в 95% случаев. Это означает, что в 5% они *не* работают 😊 Проверяйте их в различных браузерах до

## Решения задач





### Решение задачи: Спрятать при клике

Решение задачи: <tutorial/browser/events/hide/hideOther.html> [64] .



### Решение задачи: Спрятаться

Решение задачи заключается в использовании `this` в обработчике.

 Смотреть  [68]

```
1 <input type="button" onclick="this.style.display='none'" value="Нажми, чтобы меня  
   спрятать"/>
```



### Решение задачи: Какие обработчики сработают?

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:

```
1 function handler() {  
2   alert("1");  
3 }  
4  
5 button.addEventListener("click", handler, false);  
6 button.removeEventListener("click", handler, false);
```

Обработчик `button.onclick` сработает независимо и в дополнение к назначенному в `addEventListener`.



### Решение задачи: Раскрывающееся меню

Решение, шаг 1

Для начала, зададим структуру HTML/CSS.

Меню является отдельным графическим компонентом, его лучше поместить в единый DOM-элемент.

Элементы меню с точки зрения семантики являются списком UL/LI. Заголовок должен быть отдельным кликабельным элементом.

Получаем структуру:

```
1 <div class="menu">
2   <span class="title">Сладости (нажми меня)!</span>
3   <ul>
4     <li>Пирог</li>
5     <li>Пончик</li>
6     <li>Мед</li>
7   </ul>
8 </div>
```

Для заголовка лучше использовать именно SPAN, а не DIV, так как DIV постарается занять 100% ширины, и мы не сможем ловить click только на тексте:

```
<div style="border: solid red 1px">[Сладости (нажми меня)!]</div>
```



...А SPAN — это элемент с display: inline, поэтому он занимает ровно столько места, сколько занимает текст внутри него:

```
<span style="border: solid red 1px">[Сладости (нажми меня)!]</span>
```



Раскрытие/закрытие делайте путём добавления/удаления класса .menu-open к меню, которые отвечает за стрелочку и отображение UL.

## Решение, шаг 2

---

CSS для меню:

```
01 .menu ul {  
02     margin: 0;  
03     list-style: none;  
04     padding-left: 20px;  
05  
06     display: none;  
07 }  
08  
09 .menu .title {  
10     padding-left: 16px;  
11     font-size: 18px;  
12     cursor: pointer;  
13  
14     background: url(arrow-right.png) left center no-repeat;  
15 }
```

Раскрытое меню перезаписывает соответствующие свойства:

```
1 .menu-open .title {  
2     background: url(arrow-down.png) left center no-repeat;  
3 }  
4  
5 .menu-open ul {  
6     display: block;  
7 }
```

Теперь сделайте JavaScript.

### Решение, шаг 3

---

Решение: [tutorial/browser/events/sliding/index.html](http://tutorial/browser/events/sliding/index.html) [81] .



### Решение задачи: Спрятать сообщение

#### 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 перекрыть 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

---

Решение показано тут: [tutorial/browser/events/messages/index.html](http://tutorial/browser/events/messages/index.html) [82] .

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`.



### Решение задачи: Карусель

#### HTML/CSS

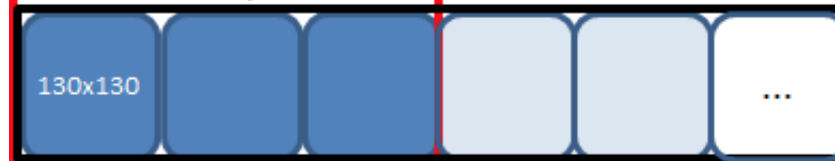
---

Лента изображений должна быть оформлена как список, согласно принципам семантической вёрстки.

Нужно стилизовать его так, чтобы он был длинной лентой, из которой внешний `DIV` вырезает нужную часть для просмотра:

div (окошко)

ul width:9999px



Чтобы список был длинный и элементы не переходили вниз, ему ставится `width: 9999px`, а элементам, соответственно, `float:left`.



Не используйте `display:inline`

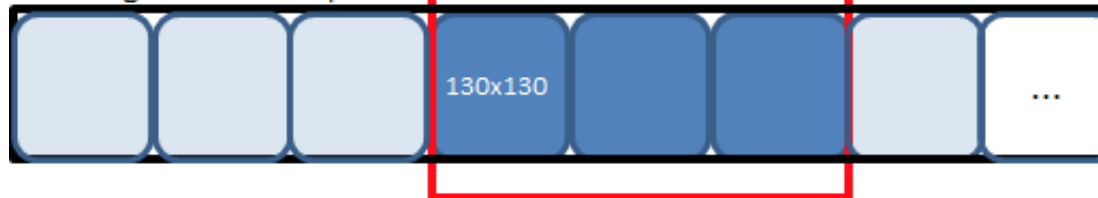
Элементы с `display:inline` имеют дополнительные отступы для возможных «хвостов букв».

В частности, для `img` нужно поставить в стилях явно `display:block`, чтобы пространства под ними не оставалось.

При прокрутке UL сдвигается назначением `margin-left`:

div (окошко)

ul marginLeft:-360px



У внешнего DIV фиксированная ширина, поэтому «лишние» изображения обрезаются.

Снаружи окошка находятся стрелки и внешний контейнер.

Реализуйте эту структуру, и к ней прикручивайте обработчики, которые меняют `ul.style.marginLeft`.

Полное решение

Решение: [tutorial/browser/events/carousel/index.html](http://tutorial/browser/events/carousel/index.html) [83]



**Решение задачи: Передвигать мяч по полю**

Мяч под курсор мыши

Основная сложность первого этапа — сдвинуть мяч под курсор, т.к. координаты клика `e.clientX/Y` — относительно

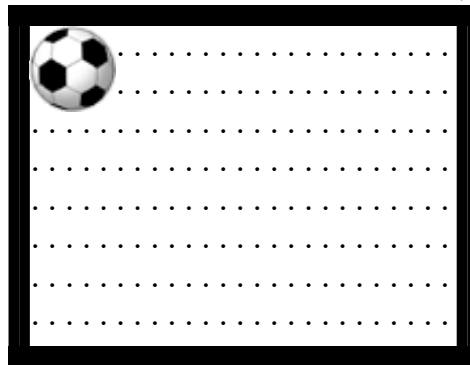
окна, а координаты мяча left/top нужно ставить относительно левого-верхнего внутреннего угла поля.

Чтобы правильно вычислить координаты мяча, нужно получить координаты угла поля и вычесть их из clientX/Y:

```
01 var field = document.getElementById('field');
02 var ball = document.getElementById('ball');
03
04 field.onclick = function(e) {
05
06     var fieldCoords = field.getBoundingClientRect();
07     var fieldInnerCoords = {
08         top: fieldCoords.top + field.clientTop,
09         left: fieldCoords.left + field.clientLeft
10     };
11
12     ball.style.left = e.clientX - fieldInnerCoords.left + 'px';
13     ball.style.top = e.clientY - fieldInnerCoords.top + 'px';
14
15 };
```

Результат:

Кликните на любое место поля, чтобы мяч перелетел туда.



[Открыть в новом окне \[87\]](#) [Открыть в песочнице \[88\]](#)

В примере выше фон мяча намеренно окрашен в серый цвет, чтобы было видно, что элемент позиционируется верно.

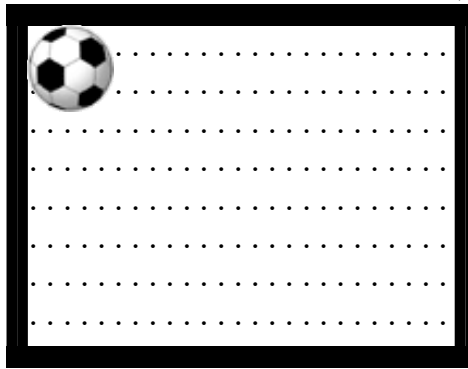
Попробуйте дальше сами.

### Дальнейшее решение

Мяч нужно сдвинуть на половину его ширины и высоты `ball.clientWidth/clientHeight`, чтобы он оказался центром под курсором. Конечно, нужно чтобы эта ширина и высота были известны, т.е. картинка либо уже была загружена, либо размеры мяча содержались в документе/CSS.

Код, который это делает и проверяет на размеры, вы найдете в полном решении:

Кликните на любое место поля, чтобы мяч перелетел туда.



[Открыть в новом окне \[89\]](#) [Открыть в песочнице \[90\]](#)

Анимация здесь добавлена всего несколькими строчками CSS, она не будет работать в IE<10.





### Решение задачи: Почему не работает return false?



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:

 Смотреть  [94]

```
1 <script>  
2   function handler() {  
3     alert("...");  
4     return false;  
5   }  
6 </script>  
7  
8 <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).

 Смотреть  [98]

```
1 <script>  
2   function handler(event) {  
3     alert("...");  
4     event.preventDefault ? event.preventDefault() : (event.returnValue=false);  
5   }  
6 </script>  
7  
8 <a href="http://w3.org" onclick="handler(event)">w3.org</a>
```



### Решение задачи: Скрытие сообщения с помощью делегирования

#### 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.

#### Решение, шаг 2

Решение задачи: [tutorial/browser/events/messages-delegate/index.html](http://tutorial/browser/events/messages-delegate/index.html) [99] .



## Решение задачи: Раскрывающееся дерево

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.

Например, ниже — участок дерева с выделенными рамкой узлами. Кликните справа от любого заголовка. Видите, клик ловится? А лучше бы такие клики (не на тексте) игнорировать.

```
01 <style>
02 li { border: 1px solid green; }
03 </style>
04
05 <ul onclick="alert(event.target || event.srcElement)">
06 <li>Млекопитающие
07   <ul>
08     <li>Коровы</li>
09     <li>Ослы</li>
10     <li>Собаки</li>
11     <li>Тигры</li>
12   </ul>
13 </li>
14 </ul>
```

- Млекопитающие
  - Коровы
  - Ослы
  - Собаки
  - Тигры

В примере выше видно, что 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.

```

01 var treeUl = document.getElementsByTagName('ul')[0];
02
03 var treeLis = treeUl.getElementsByTagName('li');
04
05 for(var i=0; i<treeLis.length; i++) {
06     var li = treeLis[i];
07
08     var span = document.createElement('span');
09     li.insertBefore(span, li.firstChild); // добавить пустой SPAN
10     span.appendChild(span.nextSibling); // переместить в него заголовок
11 }

```

Теперь можно отслеживать клики *на заголовках*.

Так выглядит дерево с обёрнутыми в SPAN заголовками и делегированием:

```

01 <style>
02 span { border: 1px solid red; }
03 </style>
04
05 <ul onclick="alert((event.target||event.srcElement).tagName)">
06 <li><span>Млекопитающие</span>
07   <ul>
08     <li><span>Коровы</span></li>
09     <li><span>Ослы</span></li>
10     <li><span>Собаки</span></li>
11     <li><span>Тигры</span></li>
12   </ul>
13 </li>
14 </ul>

```

- Млекопитающие
  - Коровы
  - Ослы
  - Собаки
  - Тигры

Так как SPAN — инлайновый элемент, он всегда такого же размера как текст. Да здравствует SPAN!

**В реальной жизни дерево должно быть сразу со SPAN, чтобы не нужно было исправлять структуру.** Если HTML-код дерева генерируется на сервере, то это несложно. Если дерево генерируется в JavaScript — тем более просто.

## Получение узла по SPAN [109]

Для делегирования нужно по клику понять, на каком узле он произошёл.

В нашем случае у SPAN нет детей-элементов, поэтому не нужно подниматься вверх по цепочке родителей. Достаточно просто проверить `event.target.tagName == 'SPAN'`, чтобы понять, где был клик, и спрятать потомков.

```
01 var tree = document.getElementsByTagName('ul')[0];
02
03 tree.onclick = function(e) {
04     e = e || event;
05     var target = e.target || e.srcElement;
06
07     if (target.tagName != 'SPAN') {
08         return; // клик был не на заголовке
09     }
10
11     var li = target.parentNode; // получить родительский LI
12
13     // получить UL с потомками -- это первый UL внутри LI
14     var node = li.getElementsByTagName('ul')[0];
15
16     if (!node) return; // потомков нет -- ничего не надо делать
17
18     // спрятать/показать (можно и через CSS-класс)
19     node.style.display = node.style.display ? '' : 'none';
20 }
```

## Жирные узлы при наведении [113]

Узел выделяется при наведении при помощи CSS-селектора `:hover`.

## Невыделяемость при клике [114]

На всё дерево можно поставить обработчик, отменяющий выделение при клике

```
tree.onselectstart = tree.onmousedown = function() {
    return false; // делаем узлы невыделяемыми
}
```

Полное решение вы можете увидеть здесь: [tutorial/browser/events/tree/index.html](http://tutorial/browser/events/tree/index.html) [115] .



### Решение задачи: Галерея изображений

Решение состоит в том, чтобы добавить обработчик на контейнер #thumbs и отслеживать клики на ссылках.

Когда происходит событие, обработчик должен изменять src #largeImg на href ссылки и заменять alt на ее title.

Код решения:

```
01 var largeImg = document.getElementById('largeImg');
02
03 document.getElementById('thumbs').onclick = function(e) {
04     e = e || window.event;
05     var target = e.target || e.srcElement;
06
07     while(target != this) {
08
09         if (target.nodeName == 'A') {
10             showThumbnail(target.href, target.title);
11             return false;
12         }
13
14         target = target.parentNode;
15     }
16 }
17
18
19 function showThumbnail(href, title) {
20     largeImg.src = href;
21     largeImg.alt = title;
22 }
```

Рабочий пример [tutorial/browser/events/gallery/index.html](http://tutorial/browser/events/gallery/index.html) [119] .

### Предзагрузка картинок

Для того, чтобы картинка загрузилась, достаточно создать новый элемент IMG и указать ему src, вот так:

```
1 var imgs = thumbs.getElementsByTagName('img');
2 for(var i=0; i<imgs.length; i++) {
3     var url = imgs[i].parentNode.href;
4
5     var img = document.createElement('img');
6     img.src = url;
7 }
```

Как только элемент создан и ему назначен src, браузер сам начинает скачивать файл картинки.

При правильных настройках сервера как-то использовать этот элемент не обязательно — картинка уже кэширована.

### Семантическая верстка

Для списка картинок используется DIV. С точки зрения семантики более верный вариант — список UL/LI.

Полное решение: [tutorial/browser/events/gallery/index.html](http://tutorial/browser/events/gallery/index.html) [123] .



### Решение задачи: Поймайте переход по ссылке

Это - классическая задача на тему делегирования.

В реальной жизни, мы можем перехватить событие и создать AJAX-запрос к серверу, который сохранит информацию о том, по какой ссылке ушел посетитель.

Мы перехватываем событие на contents и поднимаемся до parentNode пока не получим A или не упрямся в контейнер.

```
01 document.getElementById('contents').onclick = function(evt) {
02     var evt = evt || event;
03     var target = evt.target || evt.srcElement;
04
05     function handleLink(href) {
06         var isLeaving = confirm('Уйти на '+href+'?');
07         if (!isLeaving) return false;
08     }
09
10     while(target != this) {
11         if (target.nodeName == 'A') {
12             return handleLink(target.getAttribute('href')); // (*)
13         }
14         target = target.parentNode;
15     }
16 }
```

В строке (\*) используется атрибут, а не свойство href, т.к. свойство обязано содержать полный валидный адрес, а атрибут — в точности что указано в HTML.

Полное решение: [tutorial/browser/events/links.html](http://tutorial/browser/events/links.html) [127] .



## Решение задачи: Сортировка таблицы

### Подсказка (обработчик)

---

1. Обработчик `onclick` можно повесить один, на всю таблицу или `THEAD`. Он будет игнорировать клики не на `TH`.
2. При клике на `TH` обработчик будет получать номер из `TH`, на котором кликнули (`TH.cellIndex`) и вызывать функцию `sortColumn`, передавая ей номер колонки и тип.
3. Функция `sortColumn(colNum, type)` будет сортировать.

### Подсказка (сортировка)

---

Функция сортировки:

1. Переносит все `TR` из `TBODY` в массив `rowsArr`
2. Сортирует массив, используя `rowsArr.sort(compare)`, функция `compare` зависит от типа столбца.
3. Добавляет `TR` из массива обратно в `TBODY`

### Решение

---

<tutorial/browser/events/grid-sort/index.html> [128] .



### Решение задачи: Функция делегирования

Данное решение поддерживает только один обработчик на элемент, т.к. реализовано через *onсобытие*:

```
01 function delegate(elem, eventName, selectorFunc, handler) {  
02   elem['on'+eventName] = function(e) {  
03  
04     var target = e && e.target || e.srcElement;  
05  
06     while(target != this) {  
07       if (selectorFunc(target)) {  
08         return handler.call(target, e); // (*)  
09       }  
10       target = target.parentNode;  
11     }  
12  
13   }  
14 }
```

#### Важно:

- Обработчик в строке (\*) вызывается в контексте `target`, ему передаётся объект события `e`, содержащий информацию о произошедшем.
- Возвращаем результат обработчика, так что `return false` из него сработает.

Итоговый документ с ним: [tutorial/browser/events/delegate.html](http://tutorial/browser/events/delegate.html) [132] .

Если нужно более одного события одного типа на элемент, то `delegate` можно переписать с использованием `addEventListener/attachEvent`, например так: [tutorial/browser/events/delegate2.html](http://tutorial/browser/events/delegate2.html) [133]



### Решение задачи: Список с выделением

#### Решение, шаг 1

Выделение одного элемента: [tutorial/browser/events/selectable-list-1.html](http://tutorial/browser/events/selectable-list-1.html) [134]

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: [tutorial/browser/events/selectable-list-2.html](http://tutorial/browser/events/selectable-list-2.html) [135]

#### Решение, шаг 3

Выделение с Shift: [tutorial/browser/events/selectable-list-3.html](http://tutorial/browser/events/selectable-list-3.html) [136]





### Решение задачи: Дерево: проверка клика на заголовке

#### Подсказка

У события клика есть координаты. Проверьте по ним, попал ли клик на заголовок.

Самый глубокий узел на координатах можно получить вызовом `document.elementFromPoint(clientX, clientY)` [137] .

..Но заголовок является текстовым узлом, поэтому эта функция для него работать не будет. Однако это, всё же, можно обойти. Как?

#### Подсказка 2

Можно при клике на LI сделать временный SPAN и переместить в него текстовый узел-заголовок.

После этого проверить, попал ли клик в него и вернуть всё как было.

```
1 // 1) заворачиваем текстовый узел в SPAN
2
3 // 2) проверяем
4 var elem = document.elementFromPoint(e.clientX, e.clientY);
5 var isClickOnTitle = (elem == span);
6
7 // 3) возвращаем текстовый узел обратно из SPAN
```

На шаге 3 текстовый узел вынимается обратно из SPAN, всё возвращается в исходное состояние.

#### Решение

Решение: <tutorial/browser/events/tree-coords/index.html> [141] .



### Решение задачи: Как отследить прекращение движения курсора?

Самый простой способ решения этой задачи — замерять скорость движения курсора. То есть, при `mousemove` вычислять расстояние между текущими координатами и предыдущими, а затем делить на разницу во времени.

Когда скорость будет очень маленькой, например 5 пикселей за 100 миллисекунд — можно считать, что курсор остановился (некоторое дрожание может присутствовать) и обработать этот факт.

Обработчик `mousemove` может стоять на всём документе, либо на контейнере, который включает в себя интересующие нас элементы.



### Решение задачи: Выделение кнопки при заходе и клике

#### Решение, шаг 1

Для HTML можно использовать элементы INPUT, BUTTON, либо просто DIV. На последнем и остановимся:

```
<div class="button"></div>
```

Для отслеживания действий посетителя в случае с JavaScript — нужны события `mouseover`, `mouseout`, которые будут отслеживать состояние «курсор над кнопкой» (`hover`), а также `mousedown` и `mouseup` для состояния «кнопка нажата».

Для CSS нужны псевдо-селекторы `:hover` и `:active`. К сожалению, в IE<8 `:active` поддерживается слабо.

## Решение, шаг 2

---

Первым будем делать вариант с JavaScript.

Кнопку можно оформить как DIV, со стилизацией состояния через CSS:

```
01 .button {
02   width: 186px;
03   height: 52px;
04   background: url(button_static.png);
05 }
06
07 .button-hover {
08   background: url(button_hover.png);
09 }
10
11 .button-click {
12   background: url(button_click.png);
13 }
```

Изначально кнопка имеет класс `.button`, затем при проведении мышью над ней добавляется класс `.button-hover`, а при клике вместо него добавляется `.button-click`. То есть, кнопка будет иметь одно из трех сочетаний классов: `.button`, `.button .button-hover`, `.button .button-click`.

При этом так как классы состояний находятся под `.button`, то их `background` будет перекрывать свойство `background` у `.button`.

Код обработки событий:

```
01 button.onmouseover = function() {
02   addClass(this, 'button-hover');
03 }
04
05 button.onmouseout = function() {
06   removeClass(this, 'button-hover');
07 }
08
09 button.onmousedown = function() {
10   addClass(this, 'button-click');
11 }
12
13 button.onmouseup = function() {
14   removeClass(this, 'button-click');
15 }
```

Полное решение: [tutorial/browser/events/rollover/index.html](http://tutorial/browser/events/rollover/index.html) [148] .

## CSS-спрайты [149]

В решении, описанном выше, есть важный недостаток. Когда курсор мыши первый раз заходит на элемент, его фон

меняется.. Но подгрузка изображения с сервера требует времени, поэтому состояние визуально изменится с задержкой.

Такое будет только в первый раз, потом картинка уже в кеше.

Чтобы обойти эту проблему, все состояния кнопки объединяют в один [CSS-спрайт \[150\]](#):



Это изображение ставится в background. Оно ровно в три раза выше, чем обычная высота кнопки, поэтому будет видна только часть — текущее состояние. Изменение состояние реализуется через сдвиг фона при помощи background-position:

```
1 .button {  
2   width: 186px;  
3   height: 52px;  
4   background: url(button.png) no-repeat;  
5 }  
6  
7 .button-hover { background-position: 0 -52px; }  
8  
9 .button-click { background-position: 0 -104px; }
```

Полный пример: [tutorial/browser/events/rollover-sprite/index.html \[154\]](#) .

## CSS-решение [155]

Решение при помощи CSS находится здесь: [tutorial/browser/events/rollover-css/index.html \[156\]](#) .

В IE6 состояние :hover поддерживается только для элементов A, поэтому кнопку нужно сделать этим элементом.

Такое решение некорректно работает в IE<8, так как старые IE плохо поддерживают псевдоселектор :active.



**Решение задачи: Поведение "подсказка"**

[tutorial/browser/events/behavior-tooltip.html \[157\]](#)



### Решение задачи: Поведение "вложенная подсказка"

<tutorial/browser/events/behavior-tooltip-nested.html> [158]



### Решение задачи: Запретите прокрутку страницы

#### Подсказка

Источниками прокрутки могут быть: клавиши-стрелки, клавиши pageUp, pageDown, Home, End, а также колёсико мыши.

#### Решение



Запустить



[162]

```
01  /*
02  Коды клавиш, которые вызывают прокрутку:
03      33, // pageUp
04      34, // pageDown
05      35, // end
06      36, // home
07      37, // left
08      38, // up
09      39, // right
10      40  // down
11  */
12
13  document.onmousewheel = document.onwheel = function() {
14      return false;
15  };
16
17  document.addEventListener ("MozMousePixelScroll",
18      function() { return false }, false);
19
20  document.onkeydown = function(e) {
21      if (e.keyCode >= 33 && e.keyCode <= 40) return false;
22  }
```



## Решение задачи: Слайдер



### HTML/CSS, подсказка

Слайдер — это DIV, покрашенный фоном/градиентом, внутри которого находится другой DIV, оформленный как бегунок, с `position: relative`.

Бегунок немного поднят, и вылезает по высоте из родителя.

### HTML/CSS, решение

Например, вот так:

 Смотреть  [166]

```
01 <style>
02 .slider {
03     border-radius: 5px;
04     background: #E0E0E0;
05     background: -moz-linear-gradient(left top , #E0E0E0, #EEEEEE) repeat scroll 0 0
    transparent;
06     background: -webkit-gradient(linear, left top, right bottom, from(#E0E0E0), to(#EEEEEE));
07     background: linear-gradient(left top, #E0E0E0, #EEEEEE);
08     width: 310px;
09     height: 15px;
10     margin: 5px;
11 }
12 .thumb {
13     width: 10px;
14     height: 25px;
15     border-radius: 3px;
16     position: relative;
17     left: 10px;
18     top: -5px;
19     background: blue;
20     cursor: pointer;
21 }
22 </style>
23
24 <div class="slider">
25     <div class="thumb"></div>
26 </div>
```

Теперь на этом реализуйте перенос бегунка.

### Полное решение

Полное решение: [tutorial/browser/events/slider-simple/index.html](http://tutorial/browser/events/slider-simple/index.html) [167] .

Это горизонтальный Drag'n'Drop, ограниченный по ширине. Его особенность — в `position: relative` у переносимого элемента, т.е. координата ставится не абсолютная, а относительно родителя.



### Решение задачи: Расставить супергероев по полю

Решение: <tutorial/browser/drag-heroes/index.html> [168] .



### Решение задачи: Поле только для цифр

#### Подсказка: событие

Нам нужно событие `keypress`, так как по скан-коду мы не отличим, например, клавишу '2' обычную и в верхнем регистре (символ '@').

Нужно отменять действие по умолчанию (т.е. ввод), если введена не цифра.

#### Решение

Нам нужно проверять *символы* при вводе, поэтому, будем использовать событие `keypress`.

Алгоритм такой: получаем символ и проверяем, является ли он цифрой. Если не является, то отменяем действие по умолчанию.

Кроме того, игнорируем специальные символы и нажатия со включенным `Ctrl/Alt`.

Итак, вот решение:

```
01 input.onkeypress = function(e) {
02     e = e || event;
03
04     if (e.ctrlKey || e.altKey || e.metaKey) return;
05
06     var chr = getChar(e);
07
08     // с null надо осторожно в неравенствах,
09     // т.к. например null >= '0' => true
10     // на всякий случай лучше вынести проверку chr == null отдельно
11     if (chr == null) return;
12
13     if (chr < '0' || chr > '9') {
14         return false;
15     }
16 }
```

Полное решение тут: <tutorial/browser/events/numeric-input/index.html> [172] .



### Решение задачи: Отследить одновременное нажатие

#### Ход решения

- Функция `runOnKeys` — с переменным числом аргументов. Для их получения используйте `arguments`.
- Используйте два обработчика: `document.onkeydown` и `document.onkeyup`. Первый отмечает нажатие клавиши в объекте `pressed = {}`, устанавливая `pressed[keyCode] = true`, а второй — удаляет это свойство. Если все клавиши с кодами из `arguments` нажаты — запускайте `func`.
- Возникнет проблема с повторным нажатием сочетания клавиш после `alert`, решите её.



#### Решение

<tutorial/browser/events/multikeys.html> [173]



### Решение задачи: Добавьте опцию к селекту

#### Решение:

 Смотреть  [177]

```
01 <select>
02   <option value="Rock">Рок</option>
03   <option value="Blues" selected>Блюз</option>
04 </select>
05
06 <script>
07   var select = document.body.children[0];
08
09   // 1)
10   var selectedOption = select.options[select.selectedIndex];
11   alert(selectedOption.value);
12
13   // 2)
14   var newOption = new Option("Classic", "Классика");
15   select.appendChild(newOption);
16
17   // 3)
18   newOption.selected = true;
19 </script>
```



### Решение задачи: Валидация формы

Решение: <tutorial/form/validate.html> [178] .



## Решение задачи: Автовычисление процентов по вкладу

### Решение, шаг 1

---

Алгоритм решения такой.

Только численный ввод в поле с суммой разрешаем, повесив обработчик на `keypress`.

Отслеживаем события изменения для перевычисления результатов:

- На `input`: событие `input` и дополнительно `propertychange/keyup` для совместимости со старыми IE.
- На `checkbox`: событие `click` вместо `change` для совместимости с IE<9.
- На `select`: событие `change`.

### Решение, шаг 2

---

Решение: [tutorial/form/percent.html](http://tutorial/form/percent.html) [179] .





## Решение задачи: Модальное диалоговое окно

### Решение, шаг 1

---

HTML/CSS для формы: [tutorial/form/prompt-1/index.html](https://tutorialedge.net/html5/tutorial/form/prompt-1/index.html) [180] .

### Решение, шаг 2

---

Модальное окно делается путём добавления к документу DIV, полностью перекрывающего документ и имеющего больший z-index.

В результате все клики будут доставаться этому DIV'у:

Стиль:

```
01 #cover-div {  
02   position: fixed;  
03   top: 0;  
04   left: 0;  
05   z-index: 9000;  
06   width: 100%;  
07   height: 100%;  
08   background-color: gray;  
09   opacity: 0.3;  
10   filter: alpha(opacity=30);  
11 }
```

Самой форме нужно, естественно, дать еще больший z-index, чтобы она была над DIV'ом. Мы не помещаем форму в контейнер, чтобы она не унаследовала полупрозрачность.

### Решение, шаг 3

---

Решение: [tutorial/form/prompt/index.html](https://tutorialedge.net/html5/tutorial/form/prompt/index.html) [184] .



### Решение задачи: Плейсхолдер

Состояние элемента определяется наличием класса `placeholder`. Для простоты будем считать, что это — единственный возможный класс у `INPUT`'а

При фокусировке, если в элементе находится плейсхолдер — он должен исчезать:

```
01 input.onfocus = function() {  
02   if (this.className == 'placeholder') {  
03     prepareInput(this);  
04   }  
05 }  
06  
07 function prepareInput(input) { // превратить элемент в простой пустой input  
08   input.className = '';  
09   input.value = '';  
10 }
```

... Затем элемент потеряет фокус. При этом нужно вернуть плейсхолдер, но только в том случае, если элемент пустой:

```
01 input.onblur = function() {  
02   if (this.value == '') { // если пустой  
03     resetInput(this); // заполнить плейсхолдером  
04   }  
05 }  
06  
07 function resetInput(input) {  
08   input.className = 'placeholder';  
09   input.value = 'E-mail';  
10 }
```

Это решение можно сделать удобнее в поддержке, если при выполнении `prepareInput` копировать значение в специальное свойство, а в `resetInput` — восстанавливать его:

```
01 function prepareInput(input) {  
02   input.className = '';  
03   input.oldValue = input.value;  
04   input.value = '';  
05 }  
06  
07 function resetInput(input) {  
08   input.className = 'placeholder';  
09   input.value = input.oldValue;  
10 }
```

Теперь, если понадобится изменить значение плейсхолдера — это достаточно сделать в HTML, и не надо трогать JavaScript-код.

Полное решение: [tutorial/browser/events/placeholder/index.html](http://tutorial/browser/events/placeholder/index.html) [194]



### Решение задачи: Мышонок на "клавиатурном" приводе



## Алгоритм

Самый естественный алгоритм решения:

1. При клике мышонка получает фокус. Для этого нужно либо заменить DIV на другой тег, либо добавить ему `tabindex="-1"`.
2. Когда на элементе фокус, то клавиатурные события будут срабатывать прямо на нём. То есть ловим `mousie.onkeydown`.

Мы выбираем `keydown`, потому что он позволяет во-первых отлавливать нажатия на спец. клавиши (стрелки), а во-вторых, отменить действие браузера, которым по умолчанию является прокрутка страницы.

3. При нажатии на стрелки двигаем мышонка через `position: absolute` и `top/left`.

Дальше решение — попробуйте сделать сами. Возможны подводные камни 😊

## Решение

1. При получении фокуса — готовим мышонка к перемещению:

```
1 document.getElementById('mousie').onfocus = function() {  
2   this.style.position = 'relative';  
3   this.style.left = '0px';  
4   this.style.top = '0px';  
5 }
```

2. Коды для клавиш стрелок можно узнать, нажимая на них на [тестовом стенде \[198\]](#). Вот они: 37-38-39-40 (влево-вверх-вправо-вниз).

При нажатии стрелки — двигаем мышонка:

```
01 document.getElementById('mousie').onkeydown = function(e) {  
02   e = e || event;  
03   switch(e.keyCode) {  
04     case 37: // влево  
05       this.style.left = parseInt(this.style.left)-this.offsetWidth+'px';  
06       return false;  
07     case 38: // вверх  
08       this.style.top = parseInt(this.style.top)-this.offsetHeight+'px';  
09       return false;  
10     case 39: // вправо  
11       this.style.left = parseInt(this.style.left)+this.offsetWidth+'px';  
12       return false;  
13     case 40: // вниз  
14       this.style.top = parseInt(this.style.top)+this.offsetHeight+'px';  
15       return false;  
16   }  
17 }
```

Обратите внимание, что действием по умолчанию для стрелок является прокрутка страницы. Поэтому, чтобы её отменить, нужно использовать `return false`.

Когда пользователь убирает фокус с мышки, то она перестает реагировать на клавиши. Нет нужды удалять обработчики на `blur`, потому что браузер перестанет вызывать `keydown`.

В решении выше есть проблема. Мышонки находятся в DIV с `position: relative`. Это означает, что его `left/top` являются координатами не относительно документа, а относительно позиционированного предка.

Что делать? Решений три.

1. Первое — учесть этого позиционированного предка при вычислении `left/top`, вычитать его координаты из координат относительно документа.
2. Второе — сделать `position: fixed`. При этом координаты мышонка можно взять напрямую из `mouseie.getBoundingClientRect() [202]`, т.е. все вычисления выполнять относительно окна. Это больше компьютерно-игровой подход, чем работа с документом.
3. Третье — переместить мышонка под `document.body` в начале движения. Тогда и с координатами всё будет в порядке. Но при этом могут «слететь» стили.

Хочется верить, что первое и второе решения понятны. А вот третье более интересно, так как скрывает новые тонкости.

Если пойти этим путём, то в обработчик `onfocus` следует добавить перемещение мышонка под BODY:

```
1 document.getElementById('mouseie').onfocus = function() {
2   var coords = getCoords(this);
3
4   document.body.appendChild(this);
5
6   this.style.position = 'absolute';
7   this.style.left = coords.left + 'px';
8   this.style.top = coords.top + 'px';
9 };
```

...Но вот беда! При `document.body.appendChild(this)` с элемента слетает фокус!

Фокус нужно восстановить, чтобы ловить `keydown`. Однако некоторые браузеры, например FF и IE, не дают вызвать метод `focus()` элемента из его обработчика `onfocus`. То есть сделать это нельзя.

Чтобы это обойти, можно поставить обработчик не `onfocus`, а `onclick`:

```
01 document.getElementById('mouseie').onclick = function() {
02   var coords = getCoords(this);
03   this.style.position = 'absolute';
04   this.style.left = coords.left + 'px';
05   this.style.top = coords.top + 'px';
06
07   if (this.parentNode !== document.body) {
08     document.body.appendChild(this);
09     this.focus();
10   }
11 };
```

Обычно событие `focus` всё равно происходит *после* `click`, но здесь элемент перемещается, поэтому оно «съедается» и мы иницилируем его сами вызовом `focus()`.

Окончательное решение: [tutorial/browser/events/mouseie/index.html \[209\]](http://tutorial/browser/events/mouseie/index.html).



Решение задачи: Поле, предупреждающее о включенном CapsLock



## Алгоритм

JavaScript не имеет доступа к CapsLock. При загрузке страницы не известно, включён он или нет.

Но мы можем догадаться о его состоянии из событий:

1. Проверив символ, полученный по `keypress`. Символ в верхнем регистре без нажатого Shift означает, что включён CapsLock. Аналогично, символ в нижнем регистре, но с Shift говорят о включенном CapsLock. Свойство `event.shiftKey` показывает, нажат ли Shift. Так мы можем точно узнать, нажат ли Caps Lock.
2. Проверять `keydown`. Если нажат CapsLock (скан-код равен 20), то переключить состояние, но лишь в том случае, когда оно уже известно.  
Под Mac так делать не получится, поскольку клавиатурные события с CapsLock [работают некорректно \[210\]](#).

Имея состояние CapsLock в переменной, можно при фокусировке на INPUT выдавать предупреждение.

Отслеживать оба события: `keydown` и `keypress` хорошо бы на уровне документа, чтобы уже на момент входа в поле ввода мы знали состояние CapsLock.

Но при вводе сразу в нужный `input` событие `keypress` событие доплывёт до `document` и поставит состояние CapsLock *после того, как сработает на input*. Как это обойти — подумайте сами.

## Решение

При загрузке страницы, когда еще ничего не набрано, мы ничего не знаем о состоянии CapsLock, поэтому оно равно `null`:

```
var capsLockEnabled = null;
```

Когда нажата клавиша, мы можем попытаться проверить, совпадает ли регистр символа и Shift:

```
01 document.onkeypress = function(e) {  
02     e = e || event;  
03  
04     var chr = getChar(e);  
05     if (!chr) return; // специальная клавиша  
06  
07     if (chr.toLowerCase() == chr.toUpperCase()) {  
08         // символ, который не имеет регистра, такой как пробел,  
09         // мы не можем использовать для определения состояния CapsLock  
10         return;  
11     }  
12  
13     capsLockEnabled = (chr.toLowerCase() == chr && e.shiftKey) || (chr.toUpperCase() == chr  
14     && !e.shiftKey);  
15 }
```

Когда пользователь нажимает CapsLock, мы должны изменить его текущее состояние. Но мы можем сделать это только если знаем, что был нажат CapsLock.

Например, когда пользователь открыл страницу, мы не знаем, включен ли CapsLock. Затем, мы получаем событие `keydown` для CapsLock. Но мы все равно не знаем его состояния, был ли CapsLock *выключен* или, наоборот, включен.

```

1  if (navigator.platform.substr(0,3) != 'Mac') { // событие для CapsLock глючит под Mac
2      document.onkeydown = function(e) {
3          e = e || event;
4
5          if (e.keyCode == 20 && capsLockEnabled !== null) {
6              capsLockEnabled = !capsLockEnabled;
7          }
8      }
9  }

```

Теперь поле. Задание состоит в том, чтобы предупредить пользователя о включенном CapsLock, чтобы уберечь его от неправильного ввода.

1. Для начала, когда пользователь сфокусировался на поле, мы должны вывести предупреждение о CapsLock, если он включен.
2. Пользователь начинает ввод. Каждое событие `keypress` всплывает до обработчика `document.keypress`, который обновляет состояние `capsLockEnabled`.

Мы не можем использовать событие `input.onkeypress`, для отображения состояния пользователю, потому что оно сработает до `document.onkeypress` (из-за всплытия) и, следовательно, до того, как мы узнаем состояние CapsLock.

Есть много способов решить эту проблему. Можно, например, назначить обработчик состояния CapsLock на событие `input.onkeyup`. То есть, индикация будет с задержкой, но это несущественно.

Альтернативное решение — добавить на `input` такой же обработчик, как и на `document.onkeypress`.

3. ..И наконец, пользователь убирает фокус с поля. Предупреждение может быть видно, если CapsLock включен, но так как пользователь уже ушел с поля, то нам нужно спрятать предупреждение.

Код проверки поля:

```

01  <input type="text" onkeyup="checkCapsWarning(event)" onfocus="checkCapsWarning(event)"
02      onblur="removeCapsWarning()"/>
03  <div style="display:none;color:red" id="caps">Внимание: нажат CapsLock!</div>
04
05  <script>
06      function checkCapsWarning() {
07          document.getElementById('caps').style.display = capsLockEnabled ? 'block' : 'none';
08      }
09
10      function removeCapsWarning() {
11          document.getElementById('caps').style.display = 'none';
12      }
13  </script>

```

Полный код решения: [tutorial/browser/events/capslock/index.html](http://tutorial/browser/events/capslock/index.html) [220] .



#### Решение задачи: Горячие клавиши

Как видно из исходного кода, `#view` - это DIV, который будет содержать результат, а `#area` - это редактируемое текстовое поле.

## Внешний вид [221]

Так как мы преобразуем DIV в TEXTAREA и обратно, нам нужно сделать их практически одинаковыми с виду:

```
1 #view, #area {  
2   height:150px;  
3   width:400px;  
4   font-family:arial;  
5 }
```

Текстовое поле нужно как-то выделить. Можно добавить границу, но тогда изменится блок: он увеличится в размерах и немного съедет текст.

Для того, чтобы сделать размер #area таким же, как и #view, добавим поля(padding):

```
1 #view {  
2   /* padding + border = 3px */  
3   padding: 2px;  
4   border:1px solid black;  
5 }
```

#area заменяет поля границами:

```
1 #area {  
2   border: 3px groove blue;  
3   padding: 0px;  
4  
5   display:none;  
6 }
```

По умолчанию, текстовое поле скрыто. Кстати, этот код убирает дополнительную рамку в Chrome/Safari, которая появляется вокруг поля, когда на него попадает фокус:

```
#area:focus {  
  outline: none; /* убирает рамку в Safari при фокусе */  
}
```

## Коды клавиш [231]

Чтобы отследить клавиши, нам нужны их скан-коды, а не символы. Это важно, потому что горячие клавиши должны работать независимо от языковой раскладки. Поэтому, мы будем использовать keydown:

```

01 document.onkeydown = function(e) {
02     e = e || event;
03     if (e.keyCode == 27) { // escape
04         cancel();
05         return false;
06     }
07
08     if ((e.ctrlKey && e.keyCode == 'E'.charCodeAt(0)) && !area.offsetHeight) {
09         edit();
10         return false;
11     }
12
13     if ((e.ctrlKey && e.keyCode == 'S'.charCodeAt(0)) && area.offsetHeight) {
14         save();
15         return false;
16     }
17 }

```

В примере выше, `offsetHeight` используется для того, чтобы проверить, отображается элемент или нет. Это очень надежный способ для всех элементов, кроме `TR` в некоторых старых браузерах.

В отличие от простой проверки `display=='none'`, этот способ работает с элементом, скрытым с помощью стилей, а так же для элементов, у которых скрыты родители.

## Редактирование [235]

Следующие функции переключают режимы. HTML-код разрешен, поэтому возможна прямая трансформация в `TEXTAREA` и обратно.

```

01 function edit() {
02     view.style.display = 'none';
03     area.value = view.innerHTML;
04     area.style.display = 'block';
05     area.focus();
06 }
07
08 function save() {
09     area.style.display = 'none';
10     view.innerHTML = area.value;
11     view.style.display = 'block';
12 }
13
14 function cancel() {
15     area.style.display = 'none';
16     view.style.display = 'block';
17 }

```

Полное решение: [tutorial/browser/events/hotfield/index.html](http://tutorial/browser/events/hotfield/index.html) [239] .  
Чтобы проверить его, сфокусируйтесь на правом `iframe`, пожалуйста.





### Решение задачи: Красивый плейсхолдер для INPUT

#### Вёрстка

Для вёрстки можно использовать отрицательный margin у текста с подсказкой.

Решение в плане вёрстка есть в решении задачи [Расположить текст внутри INPUT \[240\]](#).

#### Решение

```
01 placeholder.onclick = function() {  
02     input.focus();  
03 }  
04  
05 // onfocus работает и вызове input.focus() и при клике на input  
06 input.onfocus = function() {  
07     if (placeholder.parentNode) {  
08         placeholder.parentNode.removeChild(placeholder);  
09     }  
10 }
```

Полный код: [tutorial/browser/events/placeholder-html.html \[244\]](#) .



### Решение задачи: Аватар наверху при прокрутке

[tutorial/browser/events/scroll-position/index.html \[245\]](#) .



### Решение задачи: Кнопка вверх-вниз

Добавим в документ DIV с кнопкой:

```
<div id="updown"></div>
```

Сама кнопка должна иметь «position:fixed».

```
1 #updown {  
2     position: fixed;  
3     top: 30px;  
4     left: 10px;  
5     cursor: pointer;  
6 }
```

Кнопка является CSS-спрайтом, поэтому мы дополнительно добавляем ей размер и два состояния:

```

01 #updown {
02   height: 9px;
03   width: 14px;
04   position: fixed;
05   top: 30px;
06   left: 10px;
07   cursor: pointer;
08 }
09
10 #updown.up {
11   background: url(updown.gif) left top;
12 }
13
14 #updown.down {
15   background: url(updown.gif) left -9px;
16 }

```

Для решения необходимо аккуратно разобрать все возможные состояния кнопки и указать, что делать при каждом.

Состояние — это просто класс элемента: up/down или пустая строка, если кнопка не видна.

При прокрутке состояния меняются следующим образом:

```

01 window.onscroll = function() {
02   var pageY = window.pageYOffset || document.documentElement.scrollTop;
03   var innerHeight = document.documentElement.clientHeight;
04
05   switch(updownElem.className) {
06     case '':
07       if (pageY > innerHeight) {
08         updownElem.className = 'up';
09       }
10       break;
11
12     case 'up':
13       if (pageY < innerHeight) {
14         updownElem.className = '';
15       }
16       break;
17
18     case 'down':
19       if (pageY > innerHeight) {
20         updownElem.className = 'up';
21       }
22       break;
23   }
24 }

```

При клике:

```

01 var pageYLabel = 0;
02
03 updownElem.onclick = function() {
04     var pageY = window.pageYOffset || document.documentElement.scrollTop;
05
06     switch(this.className) {
07         case 'up':
08             pageYLabel = pageY;
09             window.scrollTo(0, 0);
10             this.className = 'down';
11             break;
12
13         case 'down':
14             window.scrollTo(0, pageYLabel);
15             this.className = 'up';
16     }
17 }
18 }

```

Полное решение: [tutorial/browser/dom/updown/index.html](http://tutorial/browser/dom/updown/index.html) [258] .



### Решение задачи: Загрузка видимых изображений

Функция должна по текущей прокрутке определять, какие изображения видимы, и загружать их.

Она должна срабатывать не только при прокрутке, но и при загрузке. Вполне достаточно для этого — указать ее вызов в скрипте под страницей, вот так:

```

01 ... страница ...
02
03 function showVisible() {
04     var imgs = document.getElementsByTagName('img');
05     for(var i=0; i<imgs.length; i++) {
06
07         var img = imgs[i];
08
09         var realsrc = img.getAttribute('realsrc');
10         if (!realsrc) continue;
11
12         if (isVisible(img)) {
13             img.src = realsrc;
14             img.setAttribute('realsrc', '');
15         }
16     }
17 }
18
19 showVisible();
20 window.onload = showVisible;

```

При запуске функция ищет все видимые картинки с realsrc и перемещает значение realsrc в src. Обратите внимание, т.к. атрибут realsrc нестандартный, то для доступа к нему мы используем get/setAttribute. А src — стандартный, поэтому можно обратиться по DOM-свойству.

Функция проверки видимости `isVisible(elem)` получает координаты текущей видимой области и сравнивает их с элементом.

Для видимости достаточно, чтобы координаты верхней(или нижней) границы элемента находились между границами видимой области.

Как получить метрики элемента и страницы, описано в статье [Размеры и прокрутка элементов \[262\]](#).

Итоговая функция:

```
01 function isVisible(elem) {  
02  
03     var coords = getOffsetRect(elem);  
04  
05     var windowTop = window.pageYOffset ||  
06         document.documentElement.scrollTop;  
07     var windowBottom = windowTop +  
08         document.documentElement.clientHeight;  
09  
10     coords.bottom = coords.top + elem.offsetHeight;  
11  
12     // верхняя граница элем в пределах видимости  
13     // ИЛИ нижняя граница видима  
14     var topVisible = coords.top > windowTop  
15         && coords.top < windowBottom;  
16     var bottomVisible = coords.bottom < windowBottom  
17         && coords.bottom > windowTop;  
18  
19     return topVisible || bottomVisible;  
20 }
```

В решении также указан вариант с `isVisible`, который расширяет область видимости на +-1 страницу (высота страницы — `document.documentElement.clientHeight`).

Решение: [tutorial/browser/dom/lazyimg/index.html](http://tutorial/browser/dom/lazyimg/index.html) [266] .



### Решение задачи: Красивый "ALT"

#### Решение, шаг 1

Текст на странице пусть будет изначально DIV, с классом `img-replace` и атрибутом `data-src` для картинки.

Функция `replaceImg()` должна искать такие DIV и загружать изображение с указанным `src`. По `onload` осуществляется замена DIV на картинку.

#### Решение, шаг 2

Решение: [tutorial/browser/events/img-onload/index.html](http://tutorial/browser/events/img-onload/index.html) [267] .



### Решение задачи: Загрузить изображения с коллбэком

#### Подсказка

---

Создайте переменную-счетчик для подсчёта количества загруженных картинок, и увеличивайте при каждом `onload/onerror`.

Когда счетчик станет равен количеству картинок — вызывайте `callback`.

#### Решение

---

<tutorial/browser/events/images-load/index.html> [268]



### Решение задачи: Скрипт с коллбэком

#### Решение, шаг 1

---

Добавляйте `SCRIPT` при помощи методов DOM:

```
1 var script = document.createElement('script');
2 script.src = src;
3
4 // в документе может не быть HEAD или BODY,
5 // но хотя бы один (текущий) SCRIPT в документе есть
6 var s = document.getElementsByTagName('script')[0];
7 s.parentNode.insertBefore(script, s); // перед ним и вставим
```

На скрипт повесьте обработчики `onload/onreadystatechange`.

#### Решение, шаг 2

---

Решение: <tutorial/browser/events/script-load/index.html> [272] .



### Решение задачи: Скрипты с коллбэком

#### Решение, шаг 1

---

Создайте переменную-счетчик для подсчёта количества загруженных скриптов.

Чтобы один скрипт не учитывался два раза (например, `onreadystatechange` запустился при `loaded` и `complete`), учитывайте его состояние в объекте `loaded`. Свойство `loaded[i] = true` означает что *i*-й скрипт уже учтён.

#### Решение, шаг 2

---

Решение: <tutorial/browser/events/scripts-load/index.html> [273] .

# Ссылки

1. Всплытие и перехват <http://learn.javascript.ru/bubbling-and-capturing>
2. Событие фокусировки на элементе <http://learn.javascript.ru/focus-blur>
3. Selectstart [http://msdn.microsoft.com/en-us/library/ms536969\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms536969(VS.85).aspx)
4. Выделение: Range, TextRange и Selection <http://learn.javascript.ru/vydelenie-range-textrange-i-selection>
5. Controlling Selection with CSS user-select <http://blogs.msdn.com/b/ie/archive/2012/01/11/controlling-selection-with-css-user-select.aspx>
6. Unselectable [http://msdn.microsoft.com/en-us/library/ms534706\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms534706(v=vs.85).aspx)
7. Выделение: Range, TextRange и Selection <http://learn.javascript.ru/vydelenie-range-textrange-i-selection>
8. Логических операторов <http://learn.javascript.ru/logical-ops>
9. Диаграмму «Ба Гуа» [http://en.wikipedia.org/wiki/Ba\\_gua](http://en.wikipedia.org/wiki/Ba_gua)
10. SetTimeout и setInterval <http://learn.javascript.ru/settimeout-setinterval>
11. Web Workers <http://www.w3.org/TR/workers/>
12. События и методы «focus/blur» <http://learn.javascript.ru/focus-blur>
13. SetImmediate <http://learn.javascript.ru/setimmediate>
14. Web Workers <http://www.w3.org/TR/workers/>
15. SetImmediate <http://learn.javascript.ru/setimmediate>
16. Побитовая операция <http://learn.javascript.ru/bitwise-operators>
17. Прокрутка страницы <http://learn.javascript.ru/metrics-window#page-scroll>
18. DOM Level 3 <http://www.w3.org/TR/DOM-Level-3-Events/#event-type-mouseenter>
19. Onscroll <http://learn.javascript.ru/event-onscroll>
20. Стандарте <http://www.w3.org/TR/DOM-Level-3-Events/#event-type-wheel>
21. Mousewheel <http://msdn.microsoft.com/en-us/library/ie/ms536951.aspx>
22. DOMMouseScroll [https://developer.mozilla.org/en-US/docs/DOM/DOM\\_event\\_reference/DOMMouseScroll](https://developer.mozilla.org/en-US/docs/DOM/DOM_event_reference/DOMMouseScroll)
23. MozMousePixelScroll [https://developer.mozilla.org/en-US/docs/DOM/DOM\\_event\\_reference/MozMousePixelScroll](https://developer.mozilla.org/en-US/docs/DOM/DOM_event_reference/MozMousePixelScroll)
24. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
25. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
26. Специальных событий <http://www.html5rocks.com/en/tutorials/dnd/basics/>
27. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
28. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
29. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
30. Drag'n'Drop объектов <http://learn.javascript.ru/drag-and-drop-objects>
31. Предыдущей статье <http://learn.javascript.ru/drag-and-drop>
32. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
33. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
34. Основы Drag'n'Drop <http://learn.javascript.ru/drag-and-drop>
35. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
36. FixEvent <http://learn.javascript.ru/mouse-events#fixEvent>
37. Document.elementFromPoint(clientX, clientY) <https://developer.mozilla.org/en/DOM/document.elementFromPoint>
38. Координаты <http://learn.javascript.ru/coordinates#elementFromPoint>
39. Применяем ООП: Drag'n'Drop++ <http://learn.javascript.ru/drag-and-drop-plus>
40. JavaScript Madness: Keyboard Events <http://unixpapa.com/js/key.html>
41. Тестовом стенде <http://learn.javascript.ru/keyboard-events#keyboard-test-stand>
42. События для элементов форм <http://learn.javascript.ru/events-change>
43. Repeat=true <http://www.w3.org/TR/DOM-Level-3-Events/#events-Keyboardevent-repeat>
44. SetTimeout(handler, 0) <http://learn.javascript.ru/settimeout-setinterval#settimeout0>
45. События формы <http://learn.javascript.ru/events-change>
46. HTMLSelectElement <http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-94282980>
47. The Option Element <http://dev.w3.org/html5/spec/the-option-element.html#the-option-element>
48. The Option Element <http://dev.w3.org/html5/spec/the-option-element.html#the-option-element>
49. Стандарту <http://www.w3.org/TR/2006/WD-DOM-Level-3-Events-20060413/events.html#event-change>
50. DOMAttrModified <http://help.dottoro.com/ljdchxcl.php>

51. Mutation Events <http://www.w3.org/TR/DOM-Level-3-Events/#events-mutationevents>
52. SetTimeout(..., 0) <http://learn.javascript.ru/settimeout-setinterval#setTimeout0>
53. DOMAttrModified <http://help.dottoro.com/ljdchxcl.php>
54. В MSDN <http://msdn.microsoft.com/en-us/library/ms536934.aspx>
55. HideFocus <http://msdn.microsoft.com/en-us/library/ie/ms533783.aspx>
56. Размеры и прокрутка элементов <http://learn.javascript.ru/metrics>
57. Window.onerror <http://dev.w3.org/html5/spec/webappapis.html#runtime-script-errors>
58. When is a stylesheet really loaded? <http://www.phpied.com/when-is-a-stylesheet-really-loaded/>
59. When is a stylesheet really loaded? <http://www.phpied.com/when-is-a-stylesheet-really-loaded/>
60. Window.onload <http://learn.javascript.ru/onload-onerror#window.onload>
61. Async/defer <http://learn.javascript.ru/script-place-optimize#script.async>
62. Кроссбраузерное событие onDOMContentLoaded <http://javascript.ru/tutorial/events/ondomcontentloaded>
63. Html 5 canvas site:w3.org <https://www.google.ru/search?q=html+5+canvas+site:w3.org>
64. Tutorial/browser/events/hide/hideOther.html <http://learn.javascript.ru/play/tutorial/browser/events/hide/hideOther.html>
65. Показать чистый исходник в новом окне [#viewSource](#)
66. Скрыть/показать номера строк [#noList](#)
67. Печать кода с сохранением подсветки [#printSource](#)
- 68.
69. Показать чистый исходник в новом окне [#viewSource](#)
70. Скрыть/показать номера строк [#noList](#)
71. Печать кода с сохранением подсветки [#printSource](#)
72. Показать чистый исходник в новом окне [#viewSource](#)
73. Скрыть/показать номера строк [#noList](#)
74. Печать кода с сохранением подсветки [#printSource](#)
75. Показать чистый исходник в новом окне [#viewSource](#)
76. Скрыть/показать номера строк [#noList](#)
77. Печать кода с сохранением подсветки [#printSource](#)
78. Показать чистый исходник в новом окне [#viewSource](#)
79. Скрыть/показать номера строк [#noList](#)
80. Печать кода с сохранением подсветки [#printSource](#)
81. Tutorial/browser/events/sliding/index.html <http://learn.javascript.ru/play/tutorial/browser/events/sliding/index.html>
82. Tutorial/browser/events/messages/index.html <http://learn.javascript.ru/play/tutorial/browser/events/messages/index.html>
83. Tutorial/browser/events/carousel/index.html <http://learn.javascript.ru/play/tutorial/browser/events/carousel/index.html>
84. Показать чистый исходник в новом окне [#viewSource](#)
85. Скрыть/показать номера строк [#noList](#)
86. Печать кода с сохранением подсветки [#printSource](#)
87. Открыть в новом окне <http://learn.javascript.ru/files/tutorial/browser/dom/ball-move-1/index.html>
88. Открыть в песочнице <http://learn.javascript.ru/play/tutorial/browser/dom/ball-move-1/index.html>
89. Открыть в новом окне <http://learn.javascript.ru/files/tutorial/browser/dom/ball-move/index.html>
90. Открыть в песочнице <http://learn.javascript.ru/play/tutorial/browser/dom/ball-move/index.html>
91. Показать чистый исходник в новом окне [#viewSource](#)
92. Скрыть/показать номера строк [#noList](#)
93. Печать кода с сохранением подсветки [#printSource](#)
- 94.
95. Показать чистый исходник в новом окне [#viewSource](#)
96. Скрыть/показать номера строк [#noList](#)
97. Печать кода с сохранением подсветки [#printSource](#)
- 98.
99. Tutorial/browser/events/messages-delegate/index.html <http://learn.javascript.ru/play/tutorial/browser/events/messages-delegate/index.html>
100. Показать чистый исходник в новом окне [#viewSource](#)
101. Скрыть/показать номера строк [#noList](#)
102. Печать кода с сохранением подсветки [#printSource](#)
103. Показать чистый исходник в новом окне [#viewSource](#)

104. Скрыть/показать номера строк [#noList](#)
105. Печать кода с сохранением подсветки [#printSource](#)
106. Показать чистый исходник в новом окне [#viewSource](#)
107. Скрыть/показать номера строк [#noList](#)
108. Печать кода с сохранением подсветки [#printSource](#)
109. Получение узла по SPAN [#получение-узла-по-span](#)
110. Показать чистый исходник в новом окне [#viewSource](#)
111. Скрыть/показать номера строк [#noList](#)
112. Печать кода с сохранением подсветки [#printSource](#)
113. Жирные узлы при наведении [#жирные-узлы-при-наведении](#)
114. Невыделяемость при клике [#невыделяемость-при-клике](#)
115. Tutorial/browser/events/tree/index.html <http://learn.javascript.ru/play/tutorial/browser/events/tree/index.html>
116. Показать чистый исходник в новом окне [#viewSource](#)
117. Скрыть/показать номера строк [#noList](#)
118. Печать кода с сохранением подсветки [#printSource](#)
119. Tutorial/browser/events/gallery/index.html <http://learn.javascript.ru/play/tutorial/browser/events/gallery/index.html>
120. Показать чистый исходник в новом окне [#viewSource](#)
121. Скрыть/показать номера строк [#noList](#)
122. Печать кода с сохранением подсветки [#printSource](#)
123. Tutorial/browser/events/gallery/index.html <http://learn.javascript.ru/play/tutorial/browser/events/gallery/index.html>
124. Показать чистый исходник в новом окне [#viewSource](#)
125. Скрыть/показать номера строк [#noList](#)
126. Печать кода с сохранением подсветки [#printSource](#)
127. Tutorial/browser/events/links.html <http://learn.javascript.ru/play/tutorial/browser/events/links.html>
128. Tutorial/browser/events/grid-sort/index.html <http://learn.javascript.ru/play/tutorial/browser/events/grid-sort/index.html>
129. Показать чистый исходник в новом окне [#viewSource](#)
130. Скрыть/показать номера строк [#noList](#)
131. Печать кода с сохранением подсветки [#printSource](#)
132. Tutorial/browser/events/delegate.html <http://learn.javascript.ru/play/tutorial/browser/events/delegate.html>
133. Tutorial/browser/events/delegate2.html <http://learn.javascript.ru/play/tutorial/browser/events/delegate2.html>
134. Tutorial/browser/events/selectable-list-1.html <http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-1.html>
135. Tutorial/browser/events/selectable-list-2.html <http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-2.html>
136. Tutorial/browser/events/selectable-list-3.html <http://learn.javascript.ru/play/tutorial/browser/events/selectable-list-3.html>
137. Document.elementFromPoint(clientX, clientY) <https://developer.mozilla.org/en/DOM/document.elementFromPoint>
138. Показать чистый исходник в новом окне [#viewSource](#)
139. Скрыть/показать номера строк [#noList](#)
140. Печать кода с сохранением подсветки [#printSource](#)
141. Tutorial/browser/events/tree-coords/index.html <http://learn.javascript.ru/play/tutorial/browser/events/tree-coords/index.html>
142. Показать чистый исходник в новом окне [#viewSource](#)
143. Скрыть/показать номера строк [#noList](#)
144. Печать кода с сохранением подсветки [#printSource](#)
145. Показать чистый исходник в новом окне [#viewSource](#)
146. Скрыть/показать номера строк [#noList](#)
147. Печать кода с сохранением подсветки [#printSource](#)
148. Tutorial/browser/events/rollover/index.html <http://learn.javascript.ru/play/tutorial/browser/events/rollover/index.html>
149. CSS-спрайты [#css-спрайты](#)
150. CSS-спрайт <http://www.webremeslo.ru/faq/faq4.html>
151. Показать чистый исходник в новом окне [#viewSource](#)
152. Скрыть/показать номера строк [#noList](#)
153. Печать кода с сохранением подсветки [#printSource](#)
154. Tutorial/browser/events/rollover-sprite/index.html <http://learn.javascript.ru/play/tutorial/browser/events/rollover-sprite/index.html>
155. CSS-решение [#css-решение](#)
156. Tutorial/browser/events/rollover-css/index.html <http://learn.javascript.ru/play/tutorial/browser/events/rollover-css/index.html>



157. Tutorial/events/behavior-tooltip.html <http://learn.javascript.ru/play/tutorial/events/behavior-tooltip.html>  
158. Tutorial/browser/events/behavior-tooltip-nested.html <http://learn.javascript.ru/play/tutorial/browser/events/behavior-tooltip-nested.html>  
159. Показать чистый исходник в новом окне [#viewSource](#)  
160. Скрыть/показать номера строк [#noList](#)  
161. Печать кода с сохранением подсветки [#printSource](#)  
162.  
163. Показать чистый исходник в новом окне [#viewSource](#)  
164. Скрыть/показать номера строк [#noList](#)  
165. Печать кода с сохранением подсветки [#printSource](#)  
166.  
167. Tutorial/browser/events/slider-simple/index.html <http://learn.javascript.ru/play/tutorial/browser/events/slider-simple/index.html>  
168. Tutorial/browser/drag-heroes/index.html <http://learn.javascript.ru/play/tutorial/browser/drag-heroes/index.html>  
169. Показать чистый исходник в новом окне [#viewSource](#)  
170. Скрыть/показать номера строк [#noList](#)  
171. Печать кода с сохранением подсветки [#printSource](#)  
172. Tutorial/browser/events/numeric-input/index.html <http://learn.javascript.ru/play/tutorial/browser/events/numeric-input/index.html>  
173. Tutorial/browser/events/multikeys.html <http://learn.javascript.ru/play/tutorial/browser/events/multikeys.html>  
174. Показать чистый исходник в новом окне [#viewSource](#)  
175. Скрыть/показать номера строк [#noList](#)  
176. Печать кода с сохранением подсветки [#printSource](#)  
177.  
178. Tutorial/form/validate.html <http://learn.javascript.ru/play/tutorial/form/validate.html>  
179. Tutorial/form/percent.html <http://learn.javascript.ru/play/tutorial/form/percent.html>  
180. Tutorial/form/prompt-1/index.html <http://learn.javascript.ru/play/tutorial/form/prompt-1/index.html>  
181. Показать чистый исходник в новом окне [#viewSource](#)  
182. Скрыть/показать номера строк [#noList](#)  
183. Печать кода с сохранением подсветки [#printSource](#)  
184. Tutorial/form/prompt/index.html <http://learn.javascript.ru/play/tutorial/form/prompt/index.html>  
185. Показать чистый исходник в новом окне [#viewSource](#)  
186. Скрыть/показать номера строк [#noList](#)  
187. Печать кода с сохранением подсветки [#printSource](#)  
188. Показать чистый исходник в новом окне [#viewSource](#)  
189. Скрыть/показать номера строк [#noList](#)  
190. Печать кода с сохранением подсветки [#printSource](#)  
191. Показать чистый исходник в новом окне [#viewSource](#)  
192. Скрыть/показать номера строк [#noList](#)  
193. Печать кода с сохранением подсветки [#printSource](#)  
194. Tutorial/browser/events/placeholder/index.html <http://learn.javascript.ru/play/tutorial/browser/events/placeholder/index.html>  
195. Показать чистый исходник в новом окне [#viewSource](#)  
196. Скрыть/показать номера строк [#noList](#)  
197. Печать кода с сохранением подсветки [#printSource](#)  
198. Тестовом стенде <http://learn.javascript.ru/keyboard-events#keyboard-test-stand>  
199. Показать чистый исходник в новом окне [#viewSource](#)  
200. Скрыть/показать номера строк [#noList](#)  
201. Печать кода с сохранением подсветки [#printSource](#)  
202. Mousie.getBoudingClientRect() <https://developer.mozilla.org/en/DOM/element.getBoudingClientRect>  
203. Показать чистый исходник в новом окне [#viewSource](#)  
204. Скрыть/показать номера строк [#noList](#)  
205. Печать кода с сохранением подсветки [#printSource](#)  
206. Показать чистый исходник в новом окне [#viewSource](#)  
207. Скрыть/показать номера строк [#noList](#)  
208. Печать кода с сохранением подсветки [#printSource](#)  
209. Tutorial/browser/events/mousie/index.html <http://learn.javascript.ru/play/tutorial/browser/events/mousie/index.html>

- 210. Работают ли некорректно <http://learn.javascript.ru/keyboard-events#keyboard-events-order>
- 211. Показать чистый исходник в новом окне [#viewSource](#)
- 212. Скрыть/показать номера строк [#noList](#)
- 213. Печать кода с сохранением подсветки [#printSource](#)
- 214. Показать чистый исходник в новом окне [#viewSource](#)
- 215. Скрыть/показать номера строк [#noList](#)
- 216. Печать кода с сохранением подсветки [#printSource](#)
- 217. Показать чистый исходник в новом окне [#viewSource](#)
- 218. Скрыть/показать номера строк [#noList](#)
- 219. Печать кода с сохранением подсветки [#printSource](#)
- 220. Tutorial/browser/events/capslock/index.html <http://learn.javascript.ru/play/tutorial/browser/events/capslock/index.html>
- 221. Внешний вид [#внешний-вид](#)
- 222. Показать чистый исходник в новом окне [#viewSource](#)
- 223. Скрыть/показать номера строк [#noList](#)
- 224. Печать кода с сохранением подсветки [#printSource](#)
- 225. Показать чистый исходник в новом окне [#viewSource](#)
- 226. Скрыть/показать номера строк [#noList](#)
- 227. Печать кода с сохранением подсветки [#printSource](#)
- 228. Показать чистый исходник в новом окне [#viewSource](#)
- 229. Скрыть/показать номера строк [#noList](#)
- 230. Печать кода с сохранением подсветки [#printSource](#)
- 231. Коды клавиш [#коды-клавиш](#)
- 232. Показать чистый исходник в новом окне [#viewSource](#)
- 233. Скрыть/показать номера строк [#noList](#)
- 234. Печать кода с сохранением подсветки [#printSource](#)
- 235. Редактирование [#редактирование](#)
- 236. Показать чистый исходник в новом окне [#viewSource](#)
- 237. Скрыть/показать номера строк [#noList](#)
- 238. Печать кода с сохранением подсветки [#printSource](#)
- 239. Tutorial/browser/events/hotfield/index.html <http://learn.javascript.ru/play/tutorial/browser/events/hotfield/index.html>
- 240. Расположить текст внутри INPUT <http://learn.javascript.ru/task/raspolozhit-tekst-vnutri-input>
- 241. Показать чистый исходник в новом окне [#viewSource](#)
- 242. Скрыть/показать номера строк [#noList](#)
- 243. Печать кода с сохранением подсветки [#printSource](#)
- 244. Tutorial/browser/events/placeholder-html.html <http://learn.javascript.ru/play/tutorial/browser/events/placeholder-html.html>
- 245. Tutorial/browser/events/scroll-position/index.html <http://learn.javascript.ru/play/tutorial/browser/events/scroll-position/index.html>
- 246. Показать чистый исходник в новом окне [#viewSource](#)
- 247. Скрыть/показать номера строк [#noList](#)
- 248. Печать кода с сохранением подсветки [#printSource](#)
- 249. Показать чистый исходник в новом окне [#viewSource](#)
- 250. Скрыть/показать номера строк [#noList](#)
- 251. Печать кода с сохранением подсветки [#printSource](#)
- 252. Показать чистый исходник в новом окне [#viewSource](#)
- 253. Скрыть/показать номера строк [#noList](#)
- 254. Печать кода с сохранением подсветки [#printSource](#)
- 255. Показать чистый исходник в новом окне [#viewSource](#)
- 256. Скрыть/показать номера строк [#noList](#)
- 257. Печать кода с сохранением подсветки [#printSource](#)
- 258. Tutorial/browser/dom/updown/index.html <http://learn.javascript.ru/play/tutorial/browser/dom/updown/index.html>
- 259. Показать чистый исходник в новом окне [#viewSource](#)
- 260. Скрыть/показать номера строк [#noList](#)
- 261. Печать кода с сохранением подсветки [#printSource](#)
- 262. Размеры и прокрутка элементов <http://learn.javascript.ru/metrics>

- 263. Показать чистый исходник в новом окне [#viewSource](#)
- 264. Скрыть/показать номера строк [#noList](#)
- 265. Печать кода с сохранением подсветки [#printSource](#)
- 266. Tutorial/browser/dom/lazyimg/index.html <http://learn.javascript.ru/play/tutorial/browser/dom/lazyimg/index.html>
- 267. Tutorial/browser/events/img-onload/index.html <http://learn.javascript.ru/play/tutorial/browser/events/img-onload/index.html>
- 268. Tutorial/browser/events/images-load/index.html <http://learn.javascript.ru/play/tutorial/browser/events/images-load/index.html>
- 269. Показать чистый исходник в новом окне [#viewSource](#)
- 270. Скрыть/показать номера строк [#noList](#)
- 271. Печать кода с сохранением подсветки [#printSource](#)
- 272. Tutorial/browser/events/script-load/index.html <http://learn.javascript.ru/play/tutorial/browser/events/script-load/index.html>
- 273. Tutorial/browser/events/scripts-load/index.html <http://learn.javascript.ru/play/tutorial/browser/events/scripts-load/index.html>