

Часть 2

Документ, события, интерфейсы

Js

Илья Кантор
2015

Документ, события, интерфейсы

Сборка от 14 мая 2015 г.

Последняя версия учебника находится на сайте learn.javascript.ru.

Мы постоянно работаем над улучшением учебника. При обнаружении ошибок пишите о них на нашем баг-трекере [↗](#).

- [Документ и объекты страницы](#)
 - Окружение: DOM, BOM и JS
 - Дерево DOM
 - Работа с DOM из консоли
 - Навигация по DOM-элементам
 - Поиск: getElement* и querySelector* и не только
 - Внутреннее устройство поисковых методов
 - Свойства узлов: тип, тег и содержимое
 - Современный DOM: полифиллы
 - Атрибуты и DOM-свойства
 - Методы contains и compareDocumentPosition
 - Добавление и удаление узлов
 - Мультивставка: insertAdjacentHTML и DocumentFragment
 - Метод document.write
 - Стили, getComputedStyle
 - Размеры и прокрутка элементов
 - Размеры и прокрутка страницы
 - Координаты в окне
 - Координаты в документе
 - Итого
- [Основы работы с событиями](#)
 - Введение в браузерные события
 - Порядок обработки событий
 - Объект события
 - Всплытие и перехват
 - Делегирование событий
 - Приём проектирования «поведение»
 - Действия браузера по умолчанию
 - Генерация событий на элементах

- События в деталях

- Мышь: клики, кнопка, координаты
- Мышь: отмена выделения, невыделяемые элементы
- Мышь: движение mouseover/out, mouseenter/leave
- Мышь: Drag'n'Drop
- Мышь: Drag'n'Drop более глубоко
- Мышь: колёсико, событие wheel
- Мышь: IE8-, исправление события
- Прокрутка: событие scroll
- Клавиатура: keyup, keydown, keypress
- Загрузка документа: DOMContentLoaded, load, beforeunload, unload
- Загрузка скриптов, картинок, фреймов: onload и onerror

- Формы, элементы управления

- Навигация и свойства элементов формы
- Фокусировка: focus/blur
- Изменение: change, input, cut, copy, paste
- Формы: отправка, событие и метод submit

- Создание графических компонентов

- Введение
- Графические компоненты
- Вёрстка графических компонентов
- Шаблонизатор LoDash
- Коллбэки и события на компонентах
- Что изучать дальше

Изучаем работу со страницей — как получать элементы, манипулировать их размерами, динамически создавать интерфейсы и взаимодействовать с посетителем.

Документ и объекты страницы

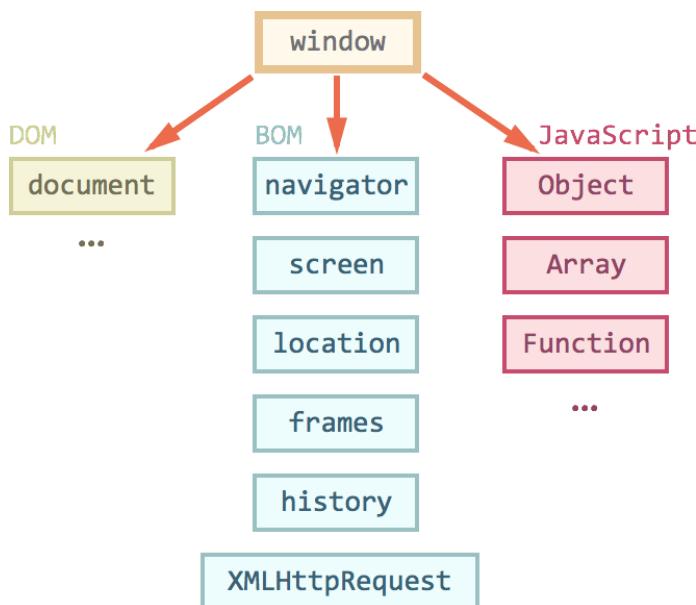
При помощи JavaScript получаем и меняем существующие элементы на странице, а также создаём новые.

Окружение: DOM, BOM и JS

Сам по себе язык JavaScript не предусматривает работы с браузером.

Он вообще не знает про HTML. Но позволяет легко расширять себя новыми функциями и объектами.

На рисунке ниже схематически отображена структура, которая получается если посмотреть на совокупность браузерных объектов с «высоты птичьего полёта».



Как видно из рисунка, на вершине стоит `window`.

У этого объекта двоякая позиция — он с одной стороны является глобальным объектом в JavaScript, с другой — содержит свойства и методы для управления окном браузера, открытия новых окон, например:

```
// открыть новое окно/вкладку с URL http://ya.ru
window.open('http://ya.ru');
```

Объектная модель документа (DOM)

Глобальный объект `document` даёт возможность взаимодействовать с содержимым страницы.

Пример использования:

```
document.body.style.background = 'red';
alert( 'Элемент BODY стал красным, а сейчас обратно вернётся' );
document.body.style.background = '';
```

Он и громадное количество его свойств и методов описаны в [стандарте W3C DOM](#).

По историческим причинам когда-то появилась первая версия стандарта DOM Level 1, затем придумали ещё свойства и методы, и появился DOM Level 2, на текущий момент поверх них добавили ещё DOM Level 3 и готовится DOM 4.

Современные браузеры также поддерживают некоторые возможности, которые не вошли в стандарты, но де-факто существуют давным-давно и отказываться от них никто не хочет. Их условно называют «DOM Level 0».

Также информацию по работе с элементами страницы можно найти в стандарте [HTML 5](#).

Мы подробно ознакомимся с DOM далее в этой части учебника.

Объектная модель браузера (BOM)

BOM — это объекты для работы с чем угодно, кроме документа.

Например:

- Объект [navigator](#) содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства: `navigator.userAgent` — содержит информацию о браузере и `navigator.platform` — содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п.
- Объект [location](#) содержит информацию о текущем URL страницы и позволяет перенаправить посетителя на новый URL.
- Функции `alert/confirm/prompt` — тоже входят в BOM.

Пример использования:

```
alert( location.href ); // выведет текущий адрес
```

Большинство возможностей BOM стандартизированы в [HTML 5](#), хотя различные браузеры и предоставляют зачастую что-то своё, в дополнение к стандарту.

Итого

Итак, у нас есть DOM, BOM и, собственно, язык JavaScript, который даёт возможность управлять всем этим.

Далее мы приступим к изучению DOM, поскольку именно документ занимает центральную роль в организации интерфейса, и работа с ним — сложнее всего.

Дерево DOM

Основным инструментом работы и динамических изменений на странице является DOM (Document Object Model) — объектная модель, используемая для XML/HTML-документов.

Согласно DOM-модели, документ является иерархией, деревом. Каждый HTML-тег образует узел дерева с типом «элемент». Вложенные в него теги становятся дочерними узлами. Для представления текста создаются узлы с типом «текст».

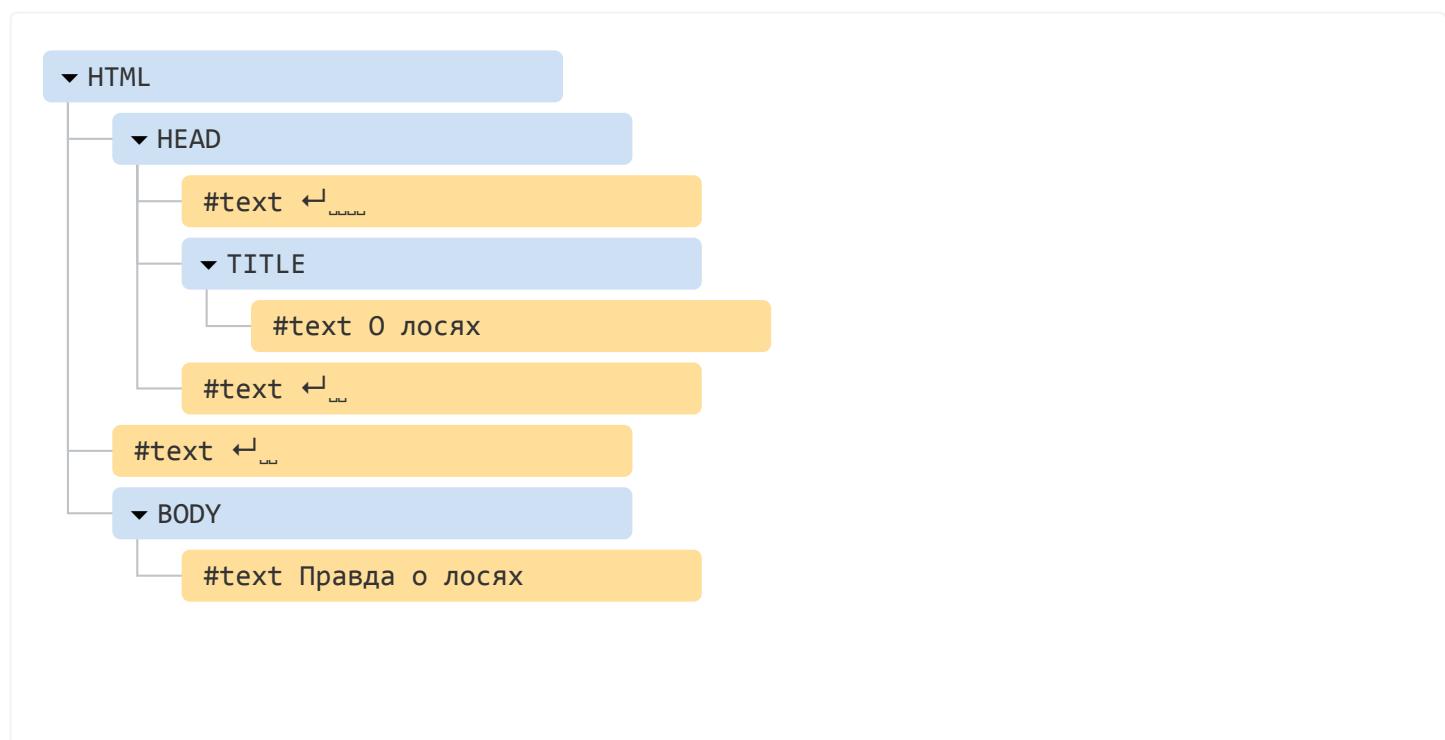
DOM — это представление документа в виде дерева объектов, доступное для изменения через JavaScript.

Пример DOM

Построим, для начала, дерево DOM для следующего документа.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>О лосях</title>
</head>
<body>
  Правда о лосях
</body>
</html>
```

Его вид:



В этом дереве выделено два типа узлов.

1. Теги образуют узлы-элементы (element node). Естественным образом одни узлы вложены в другие. Структура дерева образована исключительно за счет них.

2. Текст внутри элементов образует *текстовые узлы* (text node), обозначенные как #text.

Текстовый узел содержит исключительно строку текста и не может иметь потомков, то есть он всегда на самом нижнем уровне.

Обратите внимание на специальные символы в текстовых узлах:

- перевод строки: ↪
- пробел: ↵

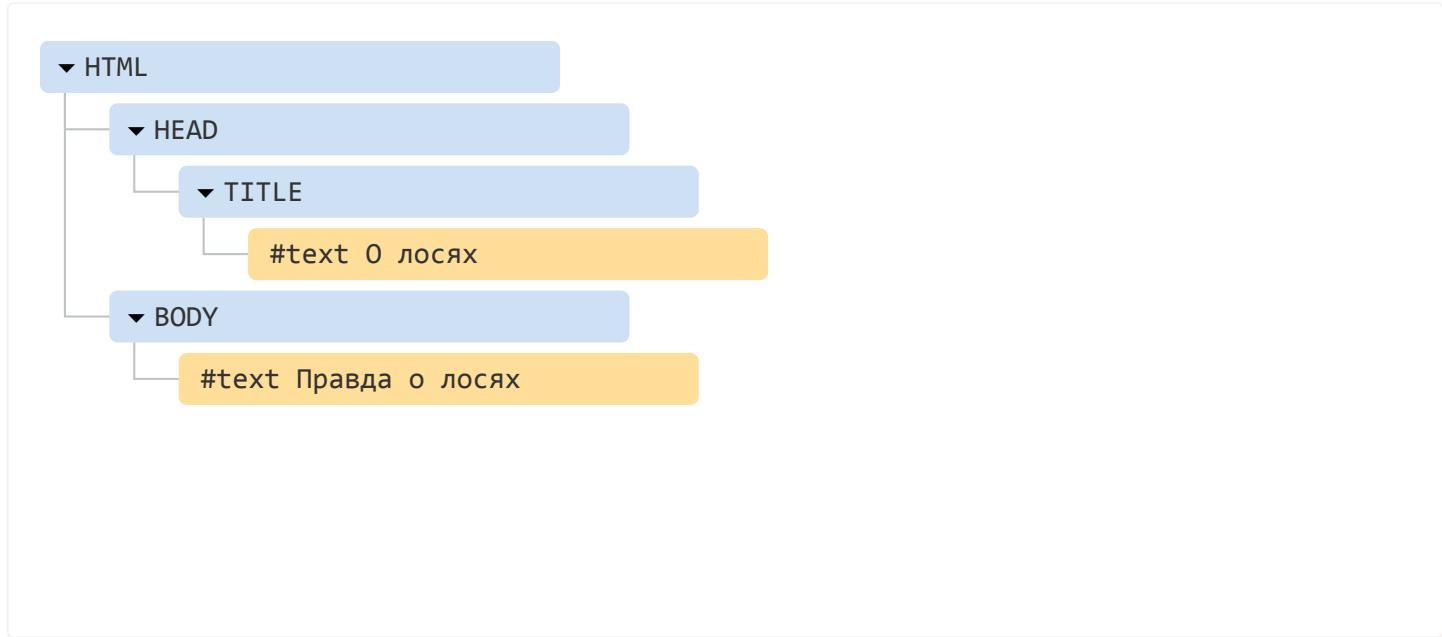
Пробелы и переводы строки — это тоже текст, полноценные символы, которые учитываются в DOM.

В частности, в примере выше тег <html> содержит не только узлы-элементы <head> и <body>, но и #text (пробелы, переводы строки) между ними.

Впрочем, как раз на самом верхнем уровне из этого правила есть исключения: пробелы до <head> по стандарту игнорируются, а любое содержимое после </body> не создаёт узла, браузер переносит его внутрь, в конец body.

В остальных случаях всё честно — если пробелы есть в документе, то они есть и в DOM, а если их убрать, то и в DOM их не будет, получится так:

```
<!DOCTYPE HTML>
<html><head><title>0 лосях</title></head><body>Правда о лосях</body></html>
```



Автоисправление

При чтении неверного HTML браузер автоматически корректирует его для показа и при построении DOM.

В частности, всегда будет верхний тег <html>. Даже если в тексте нет — в DOM он будет, браузер создаст его самостоятельно.

То же самое касается и тега <body>.

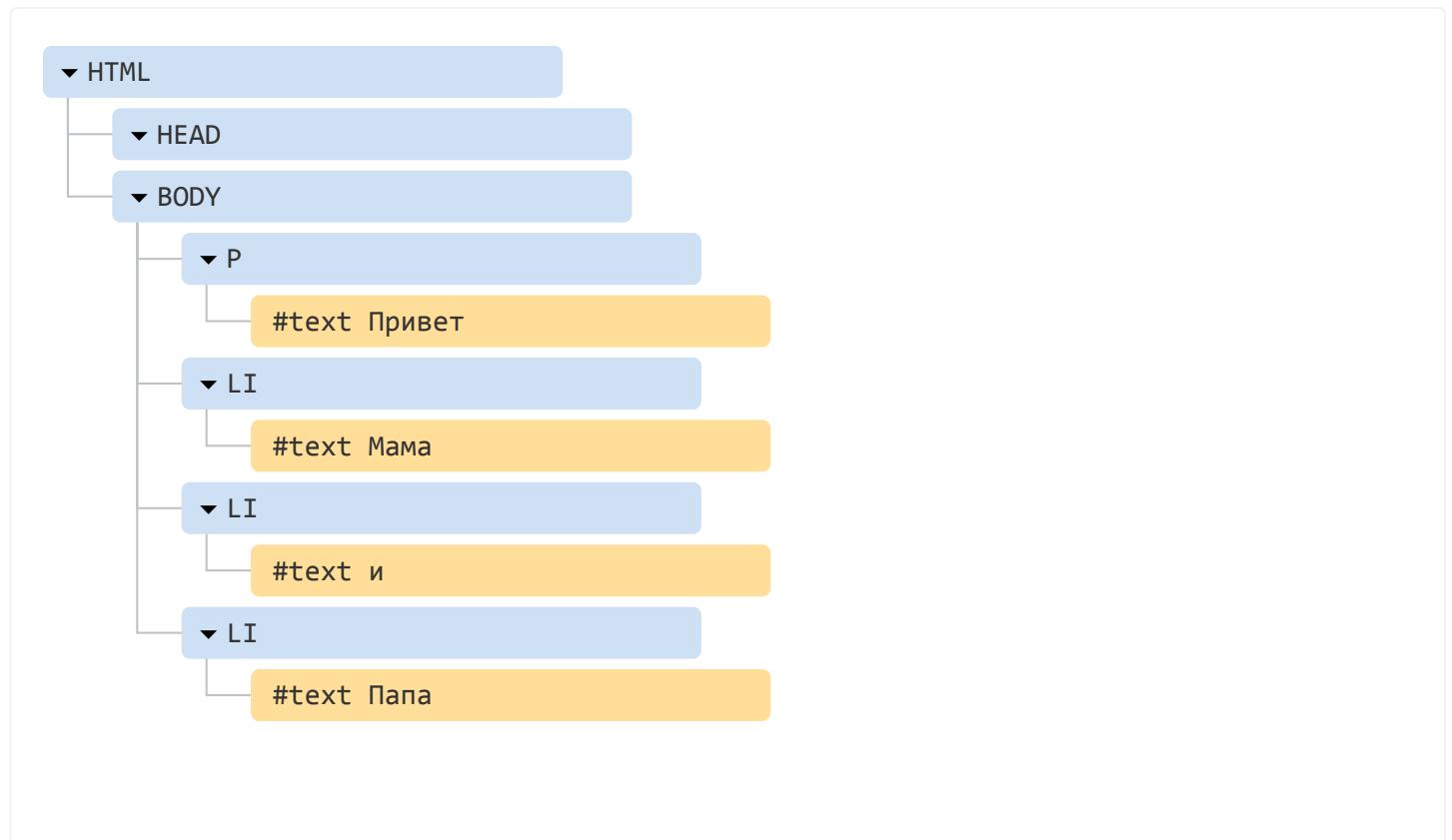
Например, если файл состоит из одного слова "Привет", то браузер автоматически обернёт его в <html> и <body>.

При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

Такой документ:

```
<p>Привет
<li>Мама
<li>и
<li>Папа
```

...Превратится вот во вполне респектабельный DOM, браузер сам закроет теги:



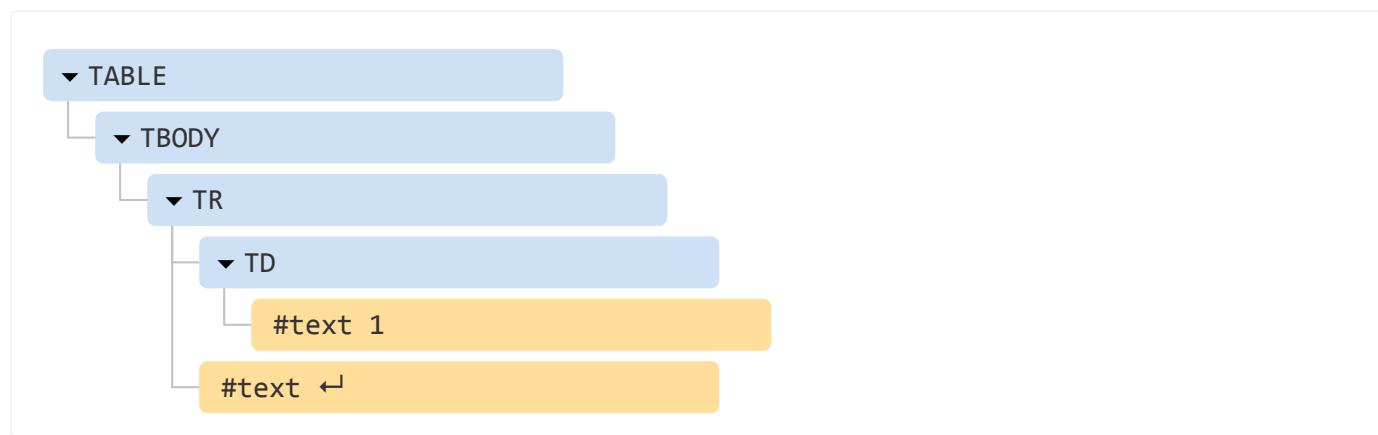
⚠ Таблицы всегда содержат `<tbody>`

Важный «особый случай» при работе с DOM — таблицы. По стандарту DOM они обязаны иметь `<tbody>`, однако в HTML их можно написать без него. В этом случае браузер добавляет `<tbody>` самостоятельно.

Например, для такого HTML:

```
<table id="table">
  <tr><td>1</td></tr>
</table>
```

DOM-структура будет такой:



Вы видите? Появился `<tbody>`, как будто документ был таким:

```
<table>
  <tbody>
    <tr><td>1</td></tr>
  </tbody>
</table>
```

Важно знать об этом, иначе при работе с таблицами возможны сюрпризы.

Другие типы узлов

Дополним страницу новыми тегами и комментарием:

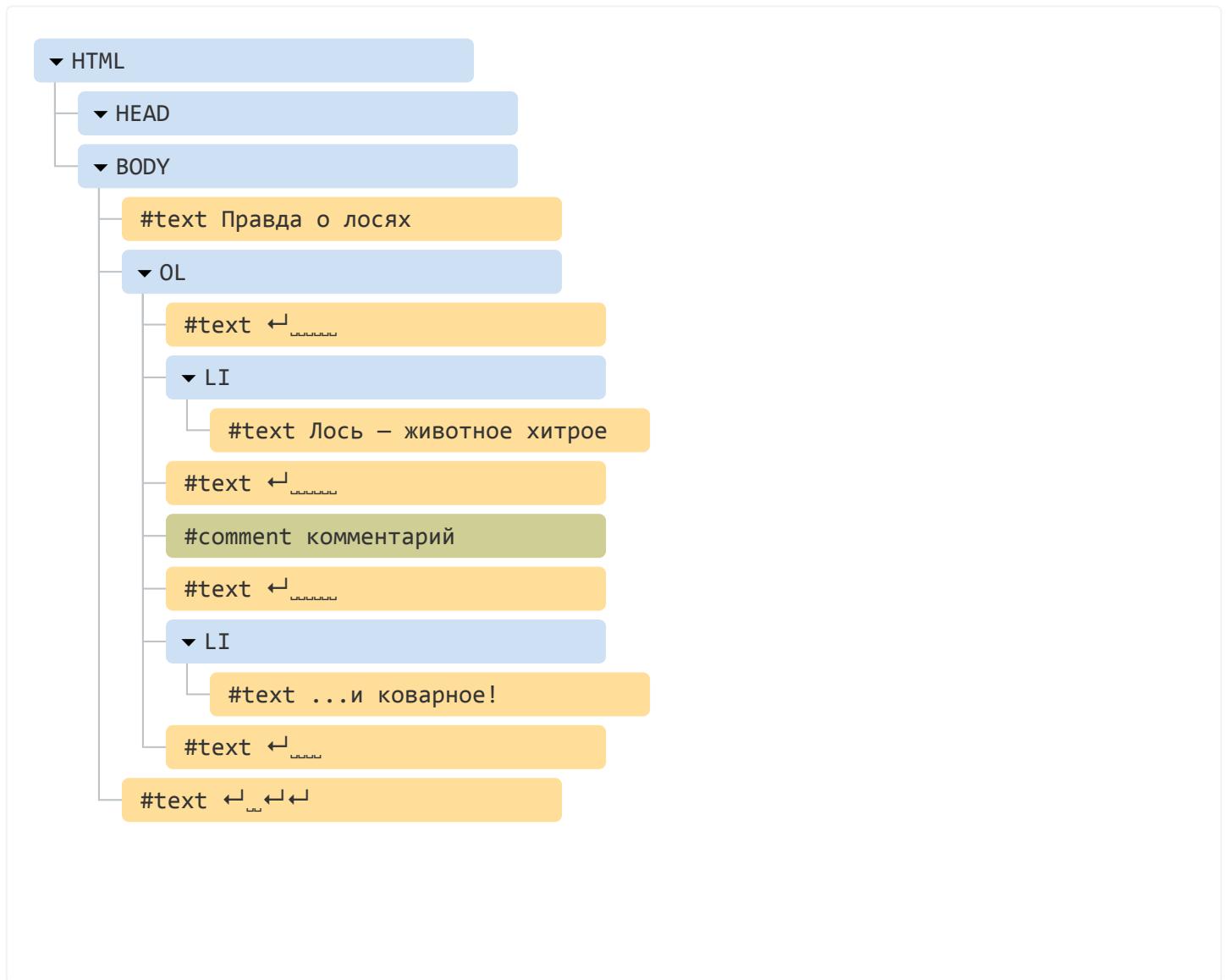
```

<!DOCTYPE HTML>
<html>

<body>
    Правда о лосях
    <ol>
        <li>Лось – животное хитрое</li>
        <!-- комментарий -->
        <li>...и коварное!</li>
    </ol>
</body>

</html>

```



В этом примере тегов уже больше, и даже появился узел нового типа — **комментарий**.

Казалось бы, зачем комментарий в DOM? На отображение-то он всё равно не влияет. Но так как он есть в HTML — обязан присутствовать в DOM-дереве.

Всё, что есть в HTML, находится и в DOM.

Даже директива `<!DOCTYPE ...>`, которую мы ставим в начале HTML, тоже является DOM-узлом, и находится в дереве DOM непосредственно перед `<html>`. На иллюстрациях выше этот факт скрыт, поскольку мы с этим узлом работать не будем, он никогда не нужен.

Даже сам объект `document`, формально, является DOM-узлом, самым-самым корневым.

Всего различают 12 типов узлов, но на практике мы работаем с четырьмя из них:

1. Документ — точка входа в DOM.
2. Элементы — основные строительные блоки.
3. Текстовые узлы — содержат, собственно, текст.
4. Комментарии — иногда в них можно включить информацию, которая не будет показана, но доступна из JS.

Возможности, которые дает DOM

Зачем, кроме красивых рисунков, нужна иерархическая модель DOM?

DOM нужен для того, чтобы манипулировать страницей — читать информацию из HTML, создавать и изменять элементы.

Узел HTML можно получить как `document.documentElement`, а BODY — как `document.body`.

Получив узел, мы можем что-то сделать с ним.

Например, можно поменять цвет BODY и вернуть обратно:

```
document.body.style.backgroundColor = 'red';
alert( 'Поменяли цвет BODY' );
document.body.style.backgroundColor = '';
alert( 'Сбросили цвет BODY' );
```

DOM предоставляет возможность делать со страницей всё, что угодно.

Позже мы более подробно рассмотрим различные свойства и методы DOM-узлов.

Особенности IE8-

IE8- не генерирует текстовые узлы, если они состоят только из пробелов.

То есть, такие два документа дадут идентичный DOM:

```
<!DOCTYPE HTML>
<html><head><title>0 лосях</title></head><body>Правда о лосях</body></html>
```

И такой:

```
<!DOCTYPE HTML>
<html>

<head>
  <title>0 лосях</title>
</head>

<body>
  Правда о лосях
</body>

</html>
```

Эта, с позволения сказать, «оптимизация» не соответствует стандарту и IE9+ уже работает как нужно, то есть как описано ранее.

Но, по большому счёту, для нас это отличие должно быть без разницы, ведь при работе с DOM/HTML мы в любом случае не должны быть завязаны на то, есть пробел между тегами или его нет. Мало ли, сегодня он есть, а завтра решили переформатировать HTML и его не стало.

К счастью, свойства и методы DOM, которые мы пройдём далее, вполне позволяют писать код, который будет работать корректно во всех версиях браузеров. Так что знать об этом отличии надо, если вы хотите поддерживать старые IE, но проблем оно нам создавать не будет.

Итого

- DOM-модель — это внутреннее представление HTML-страницы в виде дерева.
- Все элементы страницы, включая теги, текст, комментарии, являются узлами DOM.
- У элементов DOM есть свойства и методы, которые позволяют изменять их.
- IE8- не генерирует пробельные узлы.

Кстати, DOM-модель используется не только в JavaScript, это известный способ представления XML-документов.

В следующих главах мы познакомимся с DOM более плотно.

Задачи

Что выведет этот alert?

важность: 5

Что выведет alert?

```
<html>
<head>
<script>
  alert( document.body ); // ?
</script>
</head>

<body>
  Привет, мир!
</body>

</html>
```

К решению

Работа с DOM из консоли

Исследовать и изменять DOM можно с помощью инструментов разработки, встроенных в браузер. Посмотрим средства для этого на примере Google Chrome.

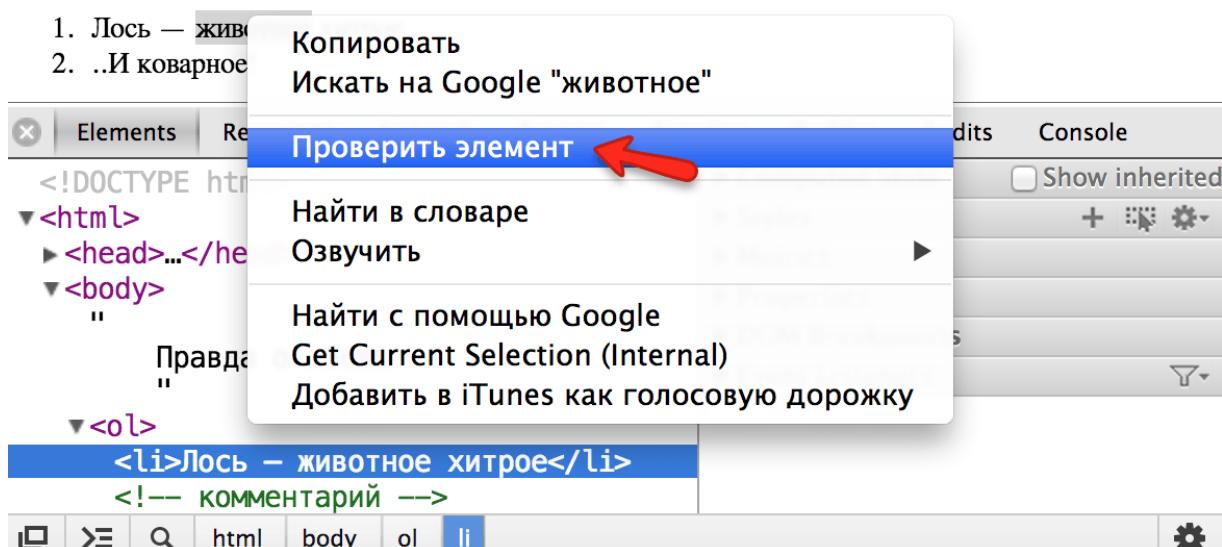
Доступ к элементу

Откройте документ [losi.html](#) и, в инструментах разработчика, перейдите во вкладку Elements.

Чтобы проанализировать любой элемент:

- Выберите его во вкладке Elements.
- ...Либо внизу вкладки Elements есть лупа, при нажатии на которую можно выбрать элемент кликом.
- ...Либо, что обычно удобнее всего, просто кликните на нужном месте на странице правой кнопкой и выберите в меню «Проверить Элемент».

Правда о лосях



Справа будет различная информация об элементе:

Computed Style

Итоговые свойства CSS элемента, которые он приобрёл в результате применения всего каскада стилей, включая внешние CSS-файлы и атрибут `style`.

Style

Каскад стилей, применённый к элементу. Каждое стилевое правило отдельно, здесь же можно менять стили кликом.

Metrics

Размеры элемента.

...

И еще некоторые реже используемые вкладки, которые станут понятны по мере изучения DOM.

DOM в Elements не совсем соответствует реальному

Отображение DOM во вкладке Elements не совсем соответствует реальному. В частности, там не отображаются пробельные узлы. Это сделано для удобства просмотра. На мы-то знаем, что они есть.

С другой стороны, в Elements можно увидеть CSS-псевдоэлементы, такие как `::before`, `::after`. Это также сделано для удобства, в DOM их не существует.

Выбранные элементы \$0 \$1...

Зачастую бывает нужно выбрать элемент DOM и сделать с ним что-то на JavaScript.

Находясь во вкладке Elements, откройте консоль нажатием Esc (или перейдите на вкладку Console).

Последний элемент, выбранный во вкладке Elements, доступен в консоли как \$0, предыдущий — \$1 и так далее.

Запустите на элементе команду, которая делает его красным:

```
$0.style.backgroundColor = 'red';
```

В браузере это может выглядеть примерно так:

Правда о лосях

1. Лось — животное хитрое
2. ..И коварное!

The screenshot shows the Chrome DevTools interface. The 'Elements' tab is active, displaying the DOM structure of a page titled 'Правда о лосях'. An 'ol' element is selected, and its child 'li' element has a red background color applied via a CSS rule. In the bottom right corner of the DevTools, there is a JavaScript console window. A red box highlights the command '\$0.style.backgroundColor = 'red'' and its result 'red', demonstrating that changes made in the console affect the live state of the element.

Мы выделили элемент, применили к нему JavaScript в консоли, тут же увидели изменения в браузере.

Есть и обратная дорожка. Любой элемент из JS-переменной можно посмотреть во вкладке Elements, для этого:

1. Выведите эту переменную в консоли, например при помощи `console.log`.
2. Кликните на выводе в консоли правой кнопкой мыши.
3. Выберите соответствующий пункт меню.

Правда о лосях

1. Лось — животное хитрое
2. li 571px x 18px

The screenshot shows the Chrome DevTools interface. The 'Elements' tab is active, displaying the DOM structure of a page titled 'Правда о лосях'. An 'li' element is selected, and its style is shown in the right-hand panel. In the bottom right corner of the DevTools, there is a JavaScript console window. A red box highlights the output of the command '\$0' (the selected 'li' element), which is then followed by a red arrow pointing to a button labeled 'Reveal in Elements Panel'. This demonstrates how to quickly switch between the Elements and Console tabs.

Таким образом, можно легко перемещаться из Elements в консоль и обратно.

Ещё методы консоли

Для поиска элементов в консоли есть два специальных метода:

- `$(“div.my”)` — ищет все элементы в DOM по данному CSS-селектору.
- `$(“div.my”)` — ищет первый элемент в DOM по данному CSS-селектору.

Более полная документация по методам консоли доступна на страницах [Console API Reference для Chrome](#) и [Command Line API для Firebug](#), а также на [firebug.ru](#).

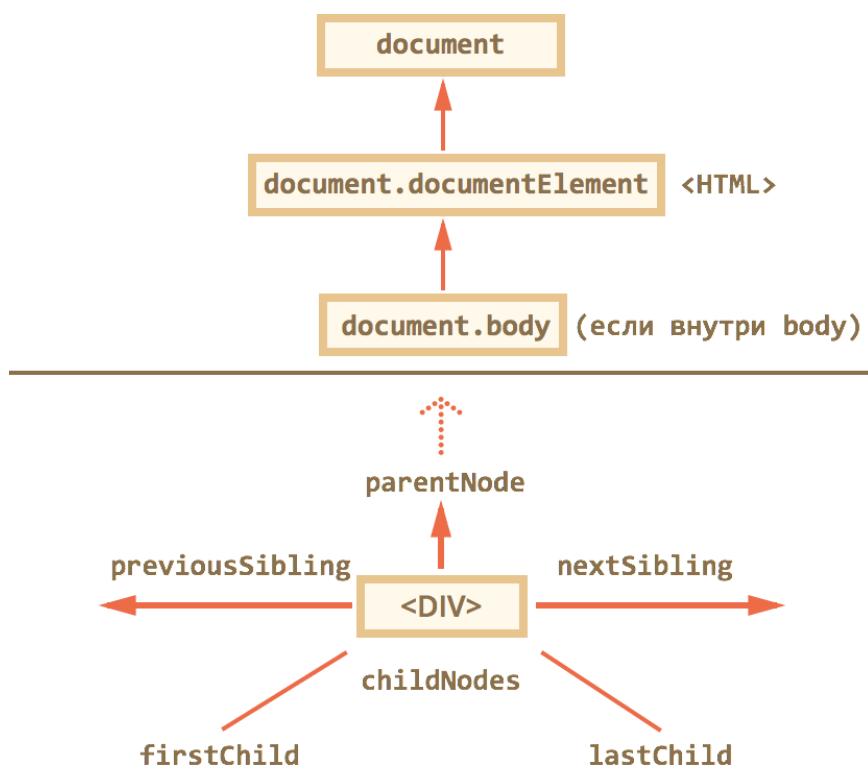
Другие браузеры реализуют похожий функционал, освоив Chrome/Firebug, вы легко с ними разберётесь.

Навигация по DOM-элементам

DOM позволяет делать что угодно с HTML-элементом и его содержимым, но для этого нужно сначала нужный элемент получить.

Доступ к DOM начинается с объекта `document`. Из него можно добраться до любых узлов.

Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



Посмотрим на них повнимательнее.

Сверху `documentElement` и `body`

Самые верхние элементы дерева доступны напрямую из `document`.

`<HTML> = document.documentElement`

Первая точка входа — `document.documentElement`. Это свойство ссылается на DOM-объект для тега `<html>`.

```
<BODY> = document.body
```

Вторая точка входа — `document.body`, который соответствует тегу `<body>`.

В современных браузерах (кроме IE8-) также есть `document.head` — прямая ссылка на `<head>`

⚠ Есть одна тонкость: `document.body` может быть равен `null`

Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.

В частности, если скрипт находится в `<head>`, то в нём недоступен `document.body`.

Поэтому в следующем примере первый `alert` выведет `null`:

```
<!DOCTYPE HTML>
<html>

<head>
  <script>
    alert( "Из HEAD: " + document.body ); // null, body ещё нет
  </script>
</head>

<body>
  <script>
    alert( "Из BODY: " + document.body ); // body есть
  </script>
</body>
</html>
```

ℹ В DOM активно используется `null`

В мире DOM в качестве значения, обозначающего «нет такого элемента» или «узел не найден», используется не `undefined`, а `null`.

Дети: `childNodes`, `firstChild`, `lastChild`

Здесь и далее мы будем использовать два принципиально разных термина.

- **Дочерние элементы (или дети)** — элементы, которые лежат *непосредственно* внутри данного. Например, внутри `<HTML>` обычно лежат `<HEAD>` и `<BODY>`.
- **Потомки** — все элементы, которые лежат внутри данного, вместе с их детьми, детьми их детей и так далее. То есть, всё поддерево DOM.

Псевдо-массив `childNodes` хранит все дочерние элементы, включая текстовые.

Пример ниже последовательно выведет дочерние элементы `document.body`:

```
<!DOCTYPE HTML>
<html>

<body>
  <div>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (var i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT
    }
  </script>
  ...
</body>

</html>
```

Обратим внимание на маленькую деталь. Если запустить пример выше, то последним будет выведен элемент `<script>`. На самом-то деле в документе есть ещё текст (обозначенный троеточием), но на момент выполнения скрипта браузер ещё до него не дошёл.

Пробельный узел будет в *итоговом документе*, но его еще нет на момент выполнения скрипта.

⚠ Список детей — только для чтения!

Скажем больше — все навигационные свойства, которые перечислены в этой главе — только для чтения. Нельзя просто заменить элемент присвоением `childNodes[i] = ...`.

Изменение DOM осуществляется другими методами, которые мы рассмотрим далее, все навигационные ссылки при этом обновляются автоматически.

Свойства `firstChild` и `lastChild` обеспечивают быстрый доступ к первому и последнему элементу.

При наличии дочерних узлов всегда верно:

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

Коллекции — не массивы

DOM-коллекции, такие как `childNodes` и другие, которые мы увидим далее, не являются JavaScript-массивами.

В них нет методов массивов, таких как `forEach`, `map`, `push`, `pop` и других.

```
var elems = document.documentElement.childNodes;  
  
elems.forEach(function(elem) { // нет такого метода!  
    /* ... */  
});
```

Именно поэтому `childNodes` и называют «коллекция» или «псевдомассив».

Можно для перебора коллекции использовать обычный цикл `for(var i=0; i<elems.length; i++)` ... Но что делать, если уж очень хочется воспользоваться методами массива?

Это возможно, основных варианта два:

1. Применить метод массива через `call/apply`:

```
var elems = document.documentElement.childNodes;  
  
[].forEach.call(elems, function(elem) {  
    alert( elem ); // HEAD, текст, BODY  
});
```

2. При помощи `Array.prototype.slice` ↗ сделать из коллекции массив.

Обычно вызов `arr.slice(a, b)` делает новый массив и копирует туда элементы `arr` с индексами от `a` до `b-1` включительно. Если же вызвать его без аргументов `arr.slice()`, то он делает новый массив и копирует туда все элементы `arr`.

Это работает и для коллекции:

```
var elems = document.documentElement.childNodes;  
elems = Array.prototype.slice.call(elems); // теперь elems - массив  
  
elems.forEach(function(elem) {  
    alert( elem.tagName ); // HEAD, текст, BODY  
});
```

Нельзя перебирать коллекцию через `for..in`

Ранее мы говорили, что не рекомендуется использовать для перебора массива цикл `for..in`.

Коллекции — наглядный пример, почему нельзя. Они похожи на массивы, но у них есть свои свойства и методы, которых в массивах нет.

К примеру, код ниже должен перебрать все дочерние элементы `<html>`. Их, естественно, два: `<head>` и `<body>`. Максимум, три, если взять ещё и текст между ними.

Но в примере ниже `alert` сработает не три, а целых 5 раз!

```
var elems = document.documentElement.childNodes;  
  
for (var key in elems) {  
    alert( key ); // 0, 1, 2, length, item  
}
```

Цикл `for..in` выведет не только ожидаемые индексы 0, 1, 2, по которым лежат узлы в коллекции, но и свойство `length` (в коллекции оно enumerable), а также функцию `item(n)` — она никогда не используется, возвращает n -й элемент коллекции, проще обратиться по индексу `[n]`.

В реальном коде нам нужны только элементы, мы же будем работать с ними, а служебные свойства — не нужны. Поэтому желательно использовать `for(var i=0; i<elems.length; i++)`.

Соседи и родитель

Доступ к элементам слева и справа данного можно получить по ссылкам `previousSibling` / `nextSibling`.

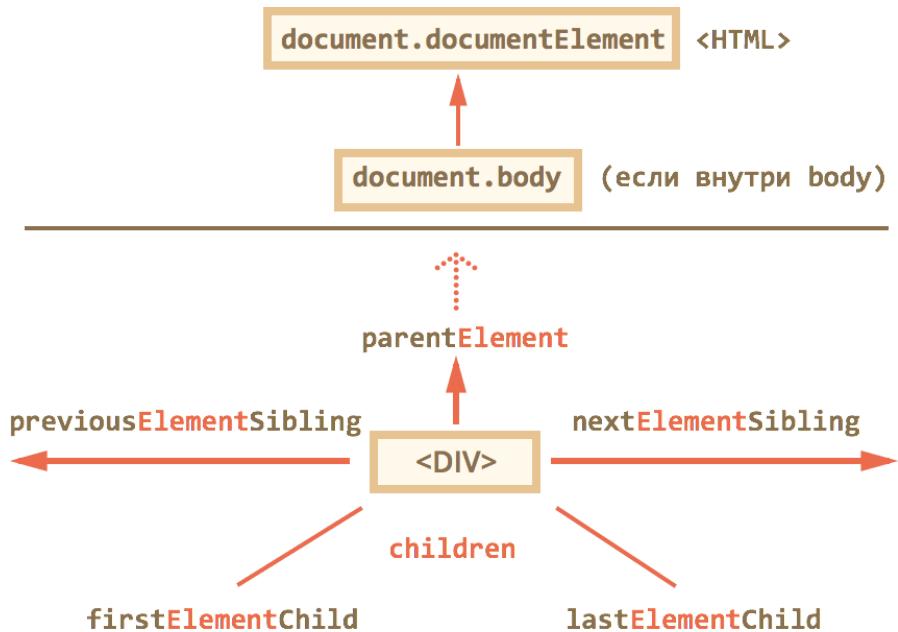
Родитель доступен через `parentNode`. Если долго идти от одного элемента к другому, то рано или поздно дойдёшь до корня DOM, то есть до `document.documentElement`, а затем и `document`.

Навигация только по элементам

Навигационные ссылки, описанные выше, равно касаются всех узлов в документе. В частности, в `childNodes` существуют и текстовые узлы и узлы-элементы и узлы-комментарии, если есть.

Но для большинства задач текстовые узлы нам не интересны.

Поэтому посмотрим на дополнительный набор ссылок, которые их не учитывают:



Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово Element:

- children — только дочерние узлы-элементы, то есть соответствующие тегам.
- firstElementChild, lastElementChild — соответственно, первый и последний дети-элементы.
- previousElementSibling, nextElementSibling — соседи-элементы.
- parentElement — родитель-элемент.

❶ Зачем parentElement? Неужели бывают родители не-элементы?

Свойство elem.parentNode возвращает родитель элемента.

Оно всегда равно parentElement, кроме одного исключения:

```
alert( document.documentElement.parentNode ); // document
alert( document.documentElement.parentElement ); // null
```

Иногда это имеет значение, если хочется перебрать всех предков и вызвать какой-то метод, а на документе его нет.

Модифицируем предыдущий пример, применив children вместо childNodes.

Теперь он будет выводить не все узлы, а только узлы-элементы:

```
<!DOCTYPE HTML>
<html>

<body>
  <div>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (var i = 0; i < document.body.children.length; i++) {
      alert( document.body.children[i] ); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>

</html>
```

Всегда верны равенства:

```
elem.firstElementChild === elem.children[0]
elem.lastElementChild === elem.children[elem.children.length - 1]
```

В IE8- поддерживается только children

Других навигационных свойств в этих браузерах нет. Впрочем, как мы увидим далее, можно легко сделать полифилл, и они, всё же, будут.

В IE8- в children присутствуют узлы-комментарии

С точки зрения стандарта это ошибка, но IE8- также включает в children узлы, соответствующие HTML-комментариям.

Это может привести к сюрпризам при использовании свойства `children`, поэтому HTML-комментарии либо убирают либо используют фреймворк, к примеру, jQuery, который даёт свои методы перебора и отфильтрует их.

Особые ссылки для таблиц

У конкретных элементов DOM могут быть свои дополнительные ссылки для большего удобства навигации.

Здесь мы рассмотрим таблицу, так как это важный частный случай и просто для примера.

В списке ниже выделены наиболее полезные:

TABLE

- **table.rows** — коллекция строк TR таблицы.
- **table.caption/tHead/tFoot** — ссылки на элементы таблицы CAPTION, THEAD, TFOOT.
- **table.tBodies** — коллекция элементов таблицы TBODY, по спецификации их может быть несколько.

THEAD/TFOOT/TBODY

- **tbody.rows** — коллекция строк TR секции.

TR

- **tr.cells** — коллекция ячеек TD/TH
- **tr.sectionRowIndex** — номер строки в текущей секции THEAD/TBODY
- **tr.rowIndex** — номер строки в таблице

TD/TH

- **td.cellIndex** — номер ячейки в строке

Пример использования:

```
<table>
  <tr>
    <td>один</td> <td>два</td>
  </tr>
  <tr>
    <td>три</td> <td>четыре</td>
  </tr>
</table>

<script>
var table = document.body.children[0];

alert( table.rows[0].cells[0].innerHTML ) // "один"
</script>
```

Спецификация: [HTML5: tabular data ↗](#).

Даже если эти свойства не нужны вам прямо сейчас, имейте их в виду на будущее, когда понадобится пройтись по таблице.

Конечно же, таблицы — не исключение.

Аналогичные полезные свойства есть у HTML-форм, они позволяют из формы получить все её элементы, а из них — в свою очередь, форму. Мы рассмотрим их позже.

Итого

В DOM доступна навигация по соседним узлам через ссылки:

- По любым узлам.

- Только по элементам.

Также некоторые виды элементов предоставляют дополнительные ссылки для большего удобства, например у таблиц есть свойства для доступа к строкам/ячейкам.

✓ Задачи

DOM children

важность: 5

Для страницы:

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <div>Пользователи:</div>
  <ul>
    <li>Маша</li>
    <li>Вовочка</li>
  </ul>

  <!-- комментарий -->

  <script>
    // ... ваш код
  </script>

</body>
</html>
```

- Напишите код, который получит элемент HEAD.
- Напишите код, который получит UL.
- Напишите код, который получит второй LI. Будет ли ваш код работать в IE8-, если комментарий переместить между элементами LI?

[К решению](#)

Проверка существования детей

важность: 5

Придумайте самый короткий код для проверки, пуст ли элемент elem.

«Пустой» — значит нет дочерних узлов, даже текстовых.

```
if /*...ваш код проверки elem...*/) { узел elem пуст }
```

Что написать в условии if ?

[К решению](#)

Вопрос по навигационным ссылкам

важность: 5

Если elem — это произвольный узел DOM...

Верно ли, что elem.lastChild.nextSibling всегда null?

Верно ли, что elem.children[0].previousSibling всегда null ?

[К решению](#)

Выделите ячейки по диагонали

важность: 5

Напишите код, который выделит все ячейки в таблице по диагонали.

Вам нужно будет получить из таблицы table все диагональные td и выделить их, используя код:

```
// в переменной td DOM-элемент для тега <td>
td.style.backgroundColor = 'red';
```

Должно получиться так:

1:1	2:1	3:1	4:1	5:1
1:2	2:2	3:2	4:2	5:2
1:3	2:3	3:3	4:3	5:3
1:4	2:4	3:4	4:4	5:4
1:5	2:5	3:5	4:5	5:5

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Поиск: getElement* и querySelector* и не только

Прямая навигация от родителя к потомку удобна, если элементы рядом. А если нет?

Как достать произвольный элемент откуда-то из глубины документа?

Для этого в DOM есть дополнительные методы поиска.

document.getElementById или просто id

Если элементу назначен специальный атрибут `id`, то можно получить его прямо по переменной с именем из значения `id`.

Например:

```
<div id="content-holder">
  <div id="content">Элемент</div>
</div>

<script>
  alert( content ); // DOM-элемент
  alert( window['content-holder'] ); // в имени дефис, поэтому через [...]
</script>
```

Это поведение соответствует [стандарту ↗](#). Оно существует, в первую очередь, для совместимости, как осколок далёкого прошлого и не очень приветствуется, поскольку использует глобальные переменные. Браузер пытается помочь нам, смешивая пространства имён JS и DOM, но при этом возможны конфликты.

Более правильной и общепринятой практикой является доступ к элементу вызовом `document.getElementById("идентификатор")`.

Например:

```
<div id="content">Выделим этот элемент</div>

<script>
  var elem = document.getElementById('content');

  elem.style.background = 'red';

  alert( elem == content ); // true

  content.style.background = ""; // один и тот же элемент
</script>
```

Должен оставаться только один

По стандарту значение `id` должно быть уникально, то есть в документе может быть только один элемент с данным `id`. И именно он будет возвращён.

Если в документе есть несколько элементов с уникальным `id`, то поведение неопределено. То есть, нет гарантии, что браузер вернёт именно первый или последний — вернёт случайным образом.

Поэтому стараются следовать правилу уникальности `id`.

Далее в примерах я часто буду использовать прямое обращение через переменную, чтобы было меньше букв и проще было понять происходящее. Но предпочтительным методом является `document.getElementById`.

getElementsByTagName

Метод `elem.getElementsByTagName(tag)` ищет все элементы с заданным тегом `tag` внутри элемента `elem` и возвращает их в виде списка.

Регистр тега не имеет значения.

Например:

```
// получить все div-элементы
var elements = document.getElementsByTagName('div');
```

Обратим внимание: в отличие от getElementById, который существует только в контексте document, метод getElementsByTagName может искать внутри любого элемента.

Например, найдём все элементы `input` внутри таблицы:

```

<table id="age-table">
  <tr>
    <td>Ваш возраст:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> младше 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> от 18 до 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> старше 60
      </label>
    </td>
  </tr>

</table>

<script>
  var tableElem = document.getElementById('age-table');
  var elements = tableElem.getElementsByTagName('input');

  for (var i = 0; i < elements.length; i++) {
    var input = elements[i];
    alert( input.value + ': ' + input.checked );
  }
</script>

```

Можно получить всех потомков, передав звездочку '*' вместо тега:

```

// получить все элементы документа
document.getElementsByTagName('*');

// получить всех потомков элемента elem:
elem.getElementsByTagName('*');

```

Не забываем про букву "s"!

Одна из самых частых ошибок начинающих (впрочем, иногда и не только) — это забыть букву "s", то есть пробовать вызывать метод `getElementByTagName` вместо `getElementsByTagName`.

Буква "s" не нужна там, где элемент только один, то есть в `getElementById`, в остальных методах она обязательна.

Возвращается коллекция, а не элемент

Другая частая ошибка — это код вида:

```
// не работает
document.getElementsByTagName('input').value = 5;
```

То есть, вместо элемента присваивают значение коллекции. Работать такое не будет.

Коллекцию нужно или перебрать в цикле или получить элемент по номеру и уже ему присваивать `value`, например так:

```
// работает
document.getElementsByTagName('input')[0].value = 5;
```

getElementsByName

Вызов `document.getElementsByName(name)` позволяет получить все элементы с данным атрибутом `name`.

Например, все элементы с именем `age`:

```
var elems = document.getElementsByName('age');
```

До появления стандарта HTML5 этот метод возвращал только те элементы, в которых предусмотрена поддержка атрибута `name`, в частности: `iframe`, `a`, `input` и другими. В современных браузерах (IE10+) тег не имеет значения.

Используется этот метод весьма редко.

getElementsByClassName

Вызов `elem.getElementsByClassName(className)` возвращает коллекцию элементов с классом `className`. Находит элемент и в том случае, если у него несколько классов, а искомый — один из них.

Поддерживается всеми современными браузерами, кроме IE8-.

Например:

```
<div class="article">Статья</div>
<div class="long article">Длинная статья</div>

<script>
  var articles = document.getElementsByClassName('article');
  alert(articles.length); // 2, найдёт оба элемента
</script>
```

Как и `getElementsByName`, этот метод может быть вызван и в контексте DOM-элемента и в контексте документа.

querySelectorAll

Вызов `elem.querySelectorAll(css)` возвращает все элементы внутри `elem`, удовлетворяющие CSS-селектору `css`.

Это один из самых часто используемых и полезных методов при работе с DOM.

Он есть во всех современных браузерах, включая IE8+ (в режиме соответствия стандарту).

Следующий запрос получает все элементы `LI`, которые являются последними потомками в `UL`:

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  var elements = document.querySelectorAll('ul > li:last-child');

  for (var i = 0; i < elements.length; i++) {
    alert(elements[i].innerHTML); // "тест", "пройден"
  }
</script>
```

querySelector

Вызов `elem.querySelector(css)` возвращает не все, а только первый элемент, соответствующий CSS-селектору `css`.

Иначе говоря, результат — такой же, как и при `elem.querySelectorAll(css)[0]`, но в последнем вызове сначала ищутся все элементы, а потом берётся первый, а в `elem.querySelector(css)` ищется только первый, то есть он эффективнее.

matches

Предыдущие методы искали по DOM.

Метод `elem.matches(css)` ничего не ищет, а проверяет, удовлетворяет ли `elem` селектору `css`. Он

возвращает true либо false.

Не поддерживается в IE8-.

Этот метод бывает полезным, когда мы перебираем элементы (в массиве или по обычным навигационным ссылкам) и пытаемся отфильтровать те из них, которые нам интересны.

Ранее в спецификации он назывался matchesSelector, и большинство браузеров поддерживают его под этим старым именем, либо с префиксами ms/moz/webkit.

Например:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  var elems = document.body.children;

  for (var i = 0; i < elems.length; i++) {
    if (elems[i].matches('a[href$="zip"]')) {
      alert( "Ссылка на архив: " + elems[i].href );
    }
  }
</script>
```

closest

Метод elem.closest(css) ищет ближайший элемент выше по иерархии DOM, подходящий под CSS-селектор css. Сам элемент тоже включается в поиск.

Иначе говоря, метод closest бежит от текущего элемента вверх по цепочке родителей и проверяет, подходит ли элемент под указанный CSS-селектор. Если подходит — останавливается и возвращает его.

Он самый новый из методов, рассмотренных в этой главе, поэтому старые браузеры его слабо поддерживают. Это, конечно, легко поправимо, как мы увидим позже в главе [Современный DOM: полифиллы](#).

Пример использования (браузер должен поддерживать closest):

```

<ul>
  <li class="chapter">Глава I
    <ul>
      <li class="subchapter">Глава <span class="num">1.1</span></li>
      <li class="subchapter">Глава <span class="num">1.2</span></li>
    </ul>
  </li>
</ul>

<script>
  var numberSpan = document.querySelector('.num');

  // ближайший элемент сверху подходящий под селектор li
  alert(numberSpan.closest('li').className) // subchapter

  // ближайший элемент сверху подходящий под селектор .chapter
  alert(numberSpan.closest('.chapter').tagName) // LI

  // ближайший элемент сверху, подходящий под селектор span
  // это сам numberSpan, так как поиск включает в себя сам элемент
  alert(numberSpan.closest('span') === numberSpan) // true
</script>

```

XPath в современных браузерах

Для полноты картины рассмотрим ещё один способ поиска, который обычно используется в XML. Это [язык запросов XPath ↗](#).

Он очень мощный, во многом мощнее CSS, но сложнее. Например, запрос для поиска элементов H2, содержащих текст "XPath", будет выглядеть так: `//h2[contains(., "XPath")]`.

Все современные браузеры, кроме IE, поддерживают XPath с синтаксисом, близким к [описанному в MDN ↗](#).

Найдем заголовки с текстом XPath в текущем документе:

```

var result = document.evaluate("//h2[contains(., 'XPath')]", document.documentElement, null,
  XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);

for (var i = 0; i < result.snapshotLength; i++) {
  alert(result.snapshotItem(i).outerHTML);
}

```

IE тоже поддерживает XPath, но эта поддержка не соответствует стандарту и работает только для XML-документов, например, полученных с помощью XMLHttpRequest (AJAX). Для обычных же HTML-документов XPath в IE не поддерживается.

Так как XPath сложнее и длиннее CSS, то используют его очень редко.

Итого

Есть 6 основных методов поиска элементов DOM:

Метод	Ищет по...	Ищет внутри элемента?	Поддержка
-------	------------	-----------------------	-----------

getElementById	id	-	везде
getElementsByName	name	-	везде
getElementsByTagName	тег или '*'	✓	везде
getElementsByClassName	классу	✓	кроме IE8-
querySelector	CSS-селектор	✓	везде
querySelectorAll	CSS-селектор	✓	везде

Практика показывает, что в 95% ситуаций достаточно querySelector/querySelectorAll. Хотя более специализированные методы getElement* работают чуть быстрее, но разница в миллисекунду-другую редко играет роль.

Кроме того:

- Есть метод elem.matches(css), который проверяет, удовлетворяет ли элемент CSS-селектору. Он поддерживается большинством браузеров в префиксной форме (ms, moz, webkit).
- Язык запросов XPath поддерживается большинством браузеров, кроме IE, даже 9й версии, но querySelector удобнее. Поэтому XPath используется редко.

✓ Задачи

Поиск элементов

важность: 4

Ниже находится документ с таблицей и формой.

Найдите (получите в переменную) в нём:

1. Все элементы `label` внутри таблицы. Должно быть 3 элемента.
2. Первую ячейку таблицы (со словом "Возраст").
3. Вторую форму в документе.
4. Форму с именем `search`, без использования её позиции в документе.
5. Элемент `input` в форме с именем `search`. Если их несколько, то нужен первый.
6. Элемент с именем `info[0]`, без точного знания его позиции в документе.
7. Элемент с именем `info[0]`, внутри формы с именем `search-person`.

Используйте для этого консоль браузера, открыв страницу [table.html](#) в отдельном окне.

[К решению](#)

Дерево

важность: 5

Есть дерево из тегов `/`.

Напишите код, который для каждого элемента `` выведет:

1. Текст непосредственно в нём (без подразделов).
2. Количество вложенных в него элементов `` — всех, с учётом вложенных.

[Демо в новом окне ↗](#)

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Внутреннее устройство поисковых методов

Эта глава не обязательна при первом чтении учебника.

Если вы хотите действительно глубоко понимать, что происходит при поиске, то посмотрите эту главу. Если нет — её можно пропустить.

Несмотря на схожесть в синтаксисе, поисковые методы `get*` и `querySelector*` внутри устроены очень по-разному.

document.getElementById(id)

Браузер поддерживает у себя внутреннее соответствие `id -> элемент`. Поэтому нужный элемент возвращается сразу. Это очень быстро.

elem.querySelector(query), elem.querySelectorAll(query)

Чтобы найти элементы, удовлетворяющие поисковому запросу, браузер не использует никаких сложных структур данных.

Он просто перебирает все подэлементы внутри элемента `elem`(или по всему документу, если вызов в контексте документа) и проверяет каждый элемент на соответствие запросу `query`.

Вызов `querySelector` прекращает перебор после первого же найденного элемента, а `querySelectorAll` собирает найденные элементы в «псевдомассив»: внутреннюю структуру данных, по сути аналогичную массиву JavaScript.

Этот перебор происходит очень быстро, так как осуществляется непосредственно движком браузера, а не JavaScript-кодом.

Оптимизации:

- В случае поиска по ID: `elem.querySelector('#id')`, большинство браузеров оптимизируют
- поиск, используя вызов `getElementById`.
 - Последние результаты поиска сохраняются в кеше. Но это до тех пор, пока документ как-нибудь не изменится.

elem.getElementsByTagName(...)

Результаты поиска `getElementsByTagName` — живые! При изменении документа — изменяется и результат запроса.

Например, найдём все `div` при помощи `querySelectorAll` и `getElementsByTagName`, а потом изменим документ:

```
<div></div>
<script>
  var resultGet = document.getElementsByTagName('div');
  var resultQuery = document.querySelectorAll('div');

  alert( resultQuery.length + ', ' + resultGet.length ); // 1, 1

  document.body.innerHTML = ''; // удалить всё содержимое BODY

  alert( resultQuery.length + ', ' + resultGet.length ); // 1, 0
</script>
```

Как видно, длина коллекции, найденной через `querySelectorAll`, осталась прежней. А длина коллекции, возвращённой `getElementsByTagName`, изменилась.

Дело в том, что результат запросов `getElementsByTagName` — это не массив, а специальный объект, имеющий тип [NodeList](#) или [HTMLCollection](#). Он похож на массив, так как имеет нумерованные элементы и длину, но внутри это не готовая коллекция, а «живой поисковой запрос».

Собственно поиск выполняется только при обращении к элементам коллекции или к её длине.

Алгоритмы `getElementsByTagName`

Поиск `getElementsByTagName` наиболее сложно сделать эффективно, так как его результат — «живая» коллекция, она должна быть всегда актуальной для текущего состояния документа.

```
var elems = document.getElementsByTagName('div');
alert( elems[0] );
// изменили документ
alert( elems[0] ); // результат может быть уже другой
```

Можно искать заново при каждой попытке получить элемент из `elems`. Тогда результат будет всегда актуален, но поиск будет работать уж слишком медленно. Да и зачем? Ведь, скорее всего, документ не поменялся.

Чтобы производительность `getElementsByTagName` была достаточно хорошей, активно используется кеширование результатов поиска.

Для этого есть два основных способа: назовём их условно «Способ Firefox» (Firefox, IE) и «Способ WebKit» (Chrome, Safari, Opera).

Для примера, рассмотрим поиск в произвольном документе, в котором есть 1000 элементов div.

Посмотрим, как будут работать браузеры, если нужно выполнить следующий код:

```
// вместо document может быть любой элемент
var elems = document.getElementsByTagName('div');
alert( elems[0] );
alert( elems[995] );
alert( elems[500] );
alert( elems.length );
```

Способ Firefox

Перебрать подэлементы document.body в порядке их появления в поддереве. Запоминать все найденные элементы во внутренней структуре данных, чтобы при повторном обращении обойтись без поиска.

Разбор действий браузера при выполнении кода выше:

1. Браузер создаёт пустую «живую коллекцию» elems. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый div. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает все найденные.
4. Возвращает ранее запомненный элемент с индексом 500, без дополнительного поиска!
5. Продолжает обход поддерева с элемента, на котором остановился (995) и до конца. Запоминает найденные элементы и возвращает их количество.

Способ WebKit

Перебирать подэлементы document.body. Запоминать только один, последний найденный, элемент, а также, по окончании перебора — длину коллекции.

Здесь кеширование используется меньше.

Разбор действий браузера по строкам:

1. Браузер создаёт пустую «живую коллекцию» elems. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый div. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает его и возвращает.
4. Браузер запоминает только последний найденный, поэтому не помнит об элементе 500. Нужно найти его перебором поддерева. Этот перебор можно начать либо с начала — вперед по поддереву, 500й по счету) либо с элемента 995 — назад по поддереву, 495й по счету. Так как назад разница в индексах меньше, то браузер выбирает второй путь и идет от 995го назад 495 раз. Запоминает теперь уже 500й элемент и возвращает его.
5. Продолжает обход поддерева с 500го (не 995го!) элемента и до конца. Запоминает число найденных элементов и возвращает его.

Основное различие — в том, что Firefox запоминает все найденные, а Webkit — только последний. Таким образом, «метод Firefox» требует больше памяти, но гораздо эффективнее при повторном доступе к предыдущим элементам.

А «метод Webkit» ест меньше памяти и при этом работает не хуже в самом важном и частом случае — последовательном переборе коллекции, без возврата к ранее выбранным.

Запомненные элементы сбрасываются при изменениях DOM.

Документ может меняться. При этом, если изменение может повлиять на результаты поиска, то запомненные элементы необходимо сбросить. Например, добавление нового узла `div` сбросит запомненные элементы коллекции `elem.getElementsByTagName('div')`.

Сбрасывание запомненных элементов при изменении документа выполняется интеллектуально.

1. Во-первых, при добавлении элемента будут сброшены только те коллекции, которые могли быть затронуты обновлением. Например, если в документе есть два независимых раздела `<section>`, и поисковая коллекция привязана к первому из них, то при добавлении во второй — она сброшена не будет.

Если точнее — будут сброшены все коллекции, привязанные к элементам вверх по иерархии от непосредственного родителя нового `div` и выше, то есть такие, которые потенциально могли измениться. И только они.

2. Во-вторых, если добавлен только `div`, то не будут сброшены запомненные элементы для поиска по другим тегам, например `elem.getElementsByTagName('a')`.
3. ...И, конечно же, не любые изменения DOM приводят к сбросу кешей, а только те, которые могут повлиять на коллекцию. Если где-то добавлен новый атрибут элементу — с кешем для `getElementsByTagName` ничего не произойдёт, так как атрибут никак не может повлиять на результат поиска по тегу.

Прочие поисковые методы, такие как `getElementsByClassName` тоже сбрасывают кеш при изменениях интеллектуально.

Разницу в алгоритмах поиска легко «пощупать». Посмотрите сами:

```
<script>
  for (var i = 0; i < 10000; i++) document.write('<span> </span>');
  var elements = document.body.getElementsByTagName('span');
  var len = elements.length;

  var d = new Date;
  for (var i = 0; i < len; i++) elements[i];
  alert("Последовательно: " + (new Date - d) + "мс"); // (1)

  var d = new Date;
  for (var i = 0; i < len; i += 2) elements[i], elements[len - i - 1];
  alert("Вразнобой: " + (new Date - d) + "мс"); // (2)
</script>
```

В примере выше первый цикл проходит элементы последовательно. А второй — идет по шагам: один с начала, потом один с конца, потом ещё один с начала, ещё один — с конца, и так далее.

Количество обращений к элементам одинаково.

- В браузерах, которые запоминают все найденные (Firefox, IE) — скорость будет одинаковой.
- В браузерах, которые запоминают только последний (Webkit) — разница будет порядка 100 и более раз, так как браузер вынужден бегать по дереву при каждом запросе заново.

✓ Задачи

Длина коллекции после удаления элементов

важность: 5

Вот небольшой документ:

```
<ul id="menu">
  <li>Главная страница</li>
  <li>Форум</li>
  <li>Магазин</li>
</ul>
```

1. Что выведет следующий код (простой вопрос)?

```
var lis = document.body.getElementsByTagName('li');

document.body.innerHTML = "";

alert( lis.length );
```

2. А такой код (вопрос посложнее)?

```
var menu = document.getElementById('menu');
var lis = menu.getElementsByTagName('li');

document.body.innerHTML = "";

alert( lis.length );
```

[К решению](#)

Сравнение количества элементов

важность: 5

Для любого документа сделаем следующее:

```
var aList1 = document.getElementsByTagName('a');
var aList2 = document.querySelectorAll('a');
```

Что произойдёт со значениями `aList1.length`, `aList2.length`, если в документе вдруг появится ещё одна ссылка `...`?

[К решению](#)

Бенчмаркинг методов поиска в DOM

важность: 2

Какой метод поиска элементов работает быстрее: `getElementsByName(tag)` или `querySelectorAll(tag)`?

Напишите код, который измеряет разницу между ними.

P.S. В задаче есть подвох, все не так просто. Если разница больше 10 раз — вы решили ее неверно. Тогда подумайте, почему такое может быть.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Получить второй LI

важность: 5

Есть длинный список ul:

```
<ul>
  <li>...</li>
  <li>...</li>
  <li>...</li>
  ...
</ul>
```

Как наиболее эффективно получить второй LI?

[К решению](#)

Свойства узлов: тип, тег и содержимое

В этой главе мы познакомимся с основными, самыми важными свойствами, которые отвечают за тип DOM-узла, тег и содержимое.

Классы, иерархия DOM

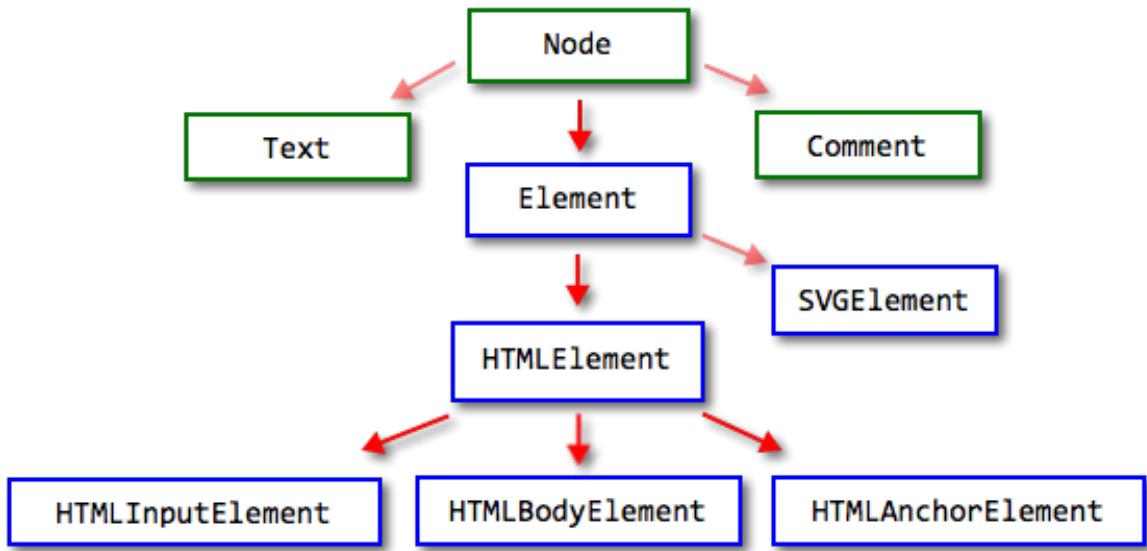
Самое главное различие между DOM-узлами — разные узлы являются объектами различных классов.

Поэтому, к примеру, у узла, соответствующего тегу `<td>` — одни свойства, у `<form>` — другие, у `<a>` — третьи.

Есть и кое-что общее, за счёт наследования.

Классы DOM образуют иерархию.

Основной объект в ней: [Node ↗](#), от которого наследуют остальные:



На рисунке выше изображены основные классы:

- Прямо от `Node` наследуют текстовые узлы `Text`, комментарии `Comment` и элементы `Element`.
- Элементы `Element` — это ещё не HTML-элементы, а более общий тип, который используется в том числе в XML. От него наследует `SVGElement` для SVG-графики и, конечно, `HTMLElement`.
- От `HTMLElement` уже наследуют разнообразные узлы HTML:
 - Для `<input>` — `HTMLInputElement`
 - Для `<body>` — `HTMLBodyElement`
 - Для `<a>` — `HTMLAnchorElement`... и так далее.

Узнать класс узла очень просто — достаточно привести его к строке, к примеру, вывести:

```
alert( document.body ); // [object HTMLBodyElement]
```

Можно и проверить при помощи `instanceof`:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
```

Как видно, DOM-узлы — обычные JavaScript-объекты. Их классы заданы в прототипном стиле. В этом легко убедиться, если вывести в консoli любой элемент через `console.dir(elem)`. Или даже можно напрямую обратиться к методам, которые хранятся в `Node.prototype`, `Element.prototype` и так далее.

`console.dir(elem)` против `console.log(elem)`

Вывод `console.log(elem)` и `console.dir(elem)` различен.

- `console.log` выводит элемент в виде, удобном для исследования HTML-структуры.
- `console.dir` выводит элемент в виде JavaScript-объекта, удобно для анализа его свойств.

Попробуйте сами на `document.body`.

Детальное описание свойств и методов каждого DOM-класса дано в [спецификации](#).

Например, [The input element](#) описывает класс, соответствующий `<input>`, включая [interface HTMLInputElement](#), который нас как раз интересует.

При описании свойств и методов используется не JavaScript, а специальный язык [IDL](#) (Interface Definition Language), который достаточно легко понять «с ходу».

Вот из него выдержка, с комментариями:

```
// Объявлен HTMLInputElement
// двоеточие означает, что он наследует от HTMLElement
interface HTMLInputElement: HTMLElement {

    // у всех таких элементов есть строковые свойства
    // accept, alt, autocomplete, value
    attribute DOMString accept;
    attribute DOMString alt;
    attribute DOMString autocomplete;
    attribute DOMString value;

    // и логическое свойство autofocus
    attribute boolean autofocus;
    ...
    // а также метод select, который значение не возвращает (void)
    void select();
    ...
}
```

Далее в этом разделе мы поговорим о самых главных свойствах узлов DOM, которые используются наиболее часто.

Тип: nodeType

Тип узла содержится в его свойстве `nodeType`.

Как правило, мы работаем всего с двумя типами узлов:

- Элемент.
- Текстовый узел.

На самом деле типов узлов гораздо больше. Строго говоря, их 12, и они описаны в спецификации с древнейших времён, см. [DOM Уровень 1](#):

```
interface Node {  
    // Всевозможные значения nodeType  
    const unsigned short ELEMENT_NODE = 1;  
    const unsigned short ATTRIBUTE_NODE = 2;  
    const unsigned short TEXT_NODE = 3;  
    const unsigned short CDATA_SECTION_NODE = 4;  
    const unsigned short ENTITY_REFERENCE_NODE = 5;  
    const unsigned short ENTITY_NODE = 6;  
    const unsigned short PROCESSING_INSTRUCTION_NODE = 7;  
    const unsigned short COMMENT_NODE = 8;  
    const unsigned short DOCUMENT_NODE = 9;  
    const unsigned short DOCUMENT_TYPE_NODE = 10;  
    const unsigned short DOCUMENT_FRAGMENT_NODE = 11;  
    const unsigned short NOTATION_NODE = 12;  
    ...  
}
```

В частности, тип «Элемент» `ELEMENT_NODE` имеет номер 1, а «Текст» `TEXT_NODE` — номер 3.

Например, выведем все узлы-потомки `document.body`, являющиеся элементами:

```
<body>  
    <div>Читатели:</div>  
    <ul>  
        <li>Вася</li>  
        <li>Петя</li>  
    </ul>  
  
    <!-- комментарий -->  
  
    <script>  
        var childNodes = document.body.childNodes;  
  
        for (var i = 0; i < childNodes.length; i++) {  
  
            // отфильтровать не-элементы  
            if (childNodes[i].nodeType != 1) continue;  
  
            alert( childNodes[i] );  
        }  
    </script>  
</body>
```

Тип узла можно только читать, изменить его невозможно.

Тег: nodeName и tagName

Существует целых два свойства: `nodeName` и `tagName`, которые содержат название(тег) элемента узла.

Название HTML-тега всегда находится в верхнем регистре.

Например, для `document.body`:

```
alert( document.body.nodeName ); // BODY  
alert( document.body.tagName ); // BODY
```

В XHTML nodeName может быть не в верхнем регистре

У браузера есть два режима обработки документа: HTML и XML-режим. Обычно используется режим HTML.

XML-режим включается, когда браузер получает XML-документ через `XMLHttpRequest`(технология AJAX) или при наличии заголовка `Content-Type: application/xml+xhtml`.

В XML-режиме сохраняется регистр и `nodeName` может выдать «body» или даже «bOdY» — в точности как указано в документе. XML-режим используют очень редко.

Какая разница между tagName и nodeName ?

Разница отражена в названиях свойств, но неочевидна.

- Свойство `tagName` есть только у элементов `Element` (в IE8- также у комментариев, но это ошибка в браузере).
- Свойство `nodeName` определено для любых узлов `Node`, для элементов оно равно `tagName`, а для не-элементов обычно содержит строку с типом узла.

Таким образом, при помощи `tagName` мы можем работать только с элементами, а `nodeName` может что-то сказать и о других типах узлов.

Например, сравним `tagName` и `nodeName` на примере узла-комментария и объекта `document`:

```
<body>
  <!-- комментарий -->

  <script>
    // для комментария
    alert( document.body.firstChild.nodeName ); // #comment
    alert( document.body.firstChild.tagName ); // undefined (в IE8- воскл. знак "!" )

    // для документа
    alert( document.nodeName ); // #document, т.к. корень DOM -- не элемент
    alert( document.tagName ); // undefined
  </script>
</body>
```

При работе с элементами, как это обычно бывает, имеет смысл использовать свойство `tagName` — оно короче.

innerHTML: содержимое элемента

Свойство `innerHTML` описано в спецификации HTML 5 — [embedded content ↗](#).

Оно позволяет получить HTML-содержимое элемента в виде строки. В `innerHTML` можно и читать и писать.

Пример выведет на экран все содержимое `document.body`, а затем заменит его на другое:

```
<body>
  <p>Параграф</p>
  <div>Div</div>

  <script>
    alert( document.body.innerHTML ); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
  </script>

</body>
```

Значение, возвращаемое `innerHTML` — всегда валидный HTML-код. При записи можно попробовать записать что угодно, но браузер исправит ошибки:

```
<body>
  <script>
    document.body.innerHTML = '<b>тест' // незакрытый тег
    alert( document.body.innerHTML ); // <b>тест</b> (исправлено)
  </script>

</body>
```

Свойство `innerHTML` — одно из самых часто используемых.

Тонкости `innerHTML`

`innerHTML` не так прост, как может показаться, и таит в себе некоторые тонкости, которые могут сбить с толку новичка, а иногда и опытного программиста.

Ознакомьтесь с ними. Даже если этих сложностей у вас пока нет, эта информация отложится где-то в голове и поможет, когда проблема появится.

⚠ Для таблиц в IE9- — innerHTML только для чтения

В Internet Explorer версии 9 и ранее, [innerHTML](#) доступно только для чтения для элементов COL, COLGROUP, FRAMESET, HEAD, HTML, STYLE, TABLE, TBODY, TFOOT, THEAD, TITLE, TR.

В частности, в IE9- запрещена запись в innerHTML для любых табличных элементов, кроме ячеек (TD/TH).

⚠ Добавление innerHTML+= осуществляет перезапись

Синтаксически, можно добавить текст к innerHTML через +=:

```
chatDiv.innerHTML += "<div>Привет<img src='smile.gif' /> !</div>";  
chatDiv.innerHTML += "Как дела?";
```

На практике этим следует пользоваться с большой осторожностью, так как фактически происходит не добавление, а перезапись:

1. Удаляется старое содержание
2. На его место становится новое значение innerHTML.

Так как новое значение записывается с нуля, то **все изображения и другие ресурсы будут перезагружены**. В примере выше вторая строчка перезагрузит smile.gif, который был до неё. Если в chatDiv много текста, то эта перезагрузка будет очень заметна.

Есть и другие побочные эффекты, например если существующий текст был выделен мышкой, то в большинстве браузеров это выделение пропадёт. Если в HTML был <input>, в который посетитель что-то ввёл, то введённое значение пропадёт. И тому подобное.

К счастью, есть и другие способы добавить содержимое, не использующие innerHTML.

Скрипты не выполняются

Если в `innerHTML` есть тег `script` — он не будет выполнен.

К примеру:

```
<div id="my"></div>

<script>
  var elem = document.getElementById('my');
  elem.innerHTML = 'TECT<script>alert( 1 );</scr' + 'ipt>';
</script>
```

В примере закрывающий тег `</scr' + 'ipt>` разбит на две строки, т.к. иначе браузер подумает, что это конец скрипта. Вставленный скрипт не выполнится.

Иключение — IE9-, в нем вставляемый скрипт выполняется, если у него есть атрибут `defer`. Но это нестандартная возможность, которой не следует пользоваться.

IE8- обрезает style и script в начале innerHTML

Если в начале `innerHTML` находятся стили `<style>`, то старый IE проигнорирует их. То есть, иными словами, они не применяются.

Смотрите также [innerHTML на MSDN](#) ↗ на эту тему.

outerHTML: HTML элемента целиком

Свойство `outerHTML` содержит HTML элемента целиком.

Пример чтения `outerHTML`:

```
<div>Привет <b>Мир</b></div>

<script>
  var div = document.body.children[0];

  alert( div.outerHTML ); // <div>Привет <b>Мир</b></div>
</script>
```

Изменить `outerHTML` элемента невозможно.

Здесь мы остановимся чуть подробнее. Дело в том, что технически свойство `outerHTML` доступно на запись. Но при этом элемент не меняется, а заменяется на новый, который тут же создаётся из нового `outerHTML`.

При этом переменная, в которой изначально был старый элемент, и в которой мы «перезаписали» `outerHTML`, остаётся со старым элементом.

Это легко может привести к ошибкам, что видно на примере:

```
<div>Привет, Мир!</div>

<script>
  var div = document.body.children[0];

  // заменяем div.outerHTML на <p>...</p>
  div.outerHTML = '<p>Новый элемент!</p>';

  // ... но содержимое div.outerHTML осталось тем же, несмотря на "перезапись"
  alert( div.outerHTML ); // <div>Привет, Мир!</div>
</script>
```

То, что произошло в примере выше — так это замена div в документе на новый узел `<p>...</p>`. При этом переменная div не получила этот новый узел! Она сохранила старое значение, чтение из неё это отлично показывает.

⚠ Записал outerHTML? Понимай последствия!

Иногда начинающие делают здесь ошибку: сначала заменяют `div.outerHTML`, а потом продолжают работать с div, как будто это изменившийся элемент. Такое возможно с `innerHTML`, но не с `outerHTML`.

Записать новый HTML в `outerHTML` можно, но нужно понимать, что это никакое не изменение свойств узла, а создание нового.

Новосозданный узел не доступен сразу в переменной, хотя его, конечно, можно получить из DOM.

nodeValue/data: содержимое текстового узла

Свойство `innerHTML` есть только у узлов-элементов.

Содержимое других узлов, например, текстовых или комментариев, доступно на чтение и запись через свойство data.

Его тоже можно читать и обновлять. Следующий пример демонстрирует это:

```
<body>
  Привет
  <!-- Комментарий -->
  <script>
    for (var i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i].data );
    }
  </script>
  Пока
</body>
```

Если вы запустите этот пример, то увидите, как выводятся последовательно:

1. Привет — это содержимое первого узла (текстового).
2. Комментарий — это содержимое второго узла (комментария).
3. Пробелы — это содержимое небольшого пробельного узла после комментария до скрипта.
4. undefined — далее цикл дошёл до `<script>`, но это узел-элемент, у него нет `data`.

Вообще говоря, после `<script>...</script>` и до закрытия `</body>` в документе есть еще один текстовый узел. Однако, на момент работы скрипта браузер ещё не знает о нём, поэтому не выведет.

Свойство `nodeValue` мы использовать не будем.

Оно работает так же, как `data`, но на некоторых узлах, где `data` нет, `nodeValue` есть и имеет значение `null`. Как-то использовать это тонкое отличие обычно нет причин.

Два свойства существуют по историческим причинам, мы будем использовать лишь `data`, поскольку оно короче.

Текст: `textContent`

Свойство `textContent` содержит только текст внутри элемента, за вычетом всех `<тегов>`.

Оно поддерживается везде, кроме IE8-.

Например:

```
<div>
  <h1>Срочно в номер!</h1>
  <p>Марсиане атакуют людей!</p>
</div>

<script>
  var news = document.body.children[0];

  // \n Срочно в номер!\n Марсиане атакуют людей!\n
  alert( news.textContent );
</script>
```

Как видно из примера выше, возвращается в точности весь текст, включая переводы строк и пробелы, но без тегов.

Иными словами, `elem.textContent` возвращает конкатенацию всех текстовых узлов внутри `elem`.

Не сказать, чтобы эта информация была часто востребована.

Гораздо полезнее возможность записать текст в элемент, причём именно как текст!

В этом примере имя посетителя попадёт в первый `div` как `innerHTML`, а во второй — как текст:

```
<div></div>
<div></div>

<script>
var name = prompt("Введите имя?", "<b>Винни-пух</b>");

document.body.children[0].innerHTML = name;
document.body.children[1].textContent = name;
</script>
```

При запуске примера мы увидим, что в первый DIV текст от посетителя вставилялся именно как HTML, то есть теги стали именно тегами, а во второй — как обычный текст.

Вряд ли мы *действительно* хотим, чтобы посетители вставляли в наш сайт произвольный HTML-код. Присваивание через `textContent` — один из способов от этого защититься.

⚠ Нестандартное свойство `innerText`

Всеми браузерами, кроме Firefox, поддерживается нестандартное свойство [innerText](#).

У него, в некотором роде, преимущество перед `textContent` в том, что оно по названию напоминает `innerHTML`, его проще запомнить.

Однако, свойство `innerText` не следует использовать, так как оно не стандартное и не будет стандартным.

Это свойство возвращает текст не в том виде, в котором он в DOM, а в том, в котором он виден — как если бы мы выбрали содержимое элемента мышкой и скопировали его. В частности, если элемент невидим, то его текст возвращён не будет. Это довольно странная особенность существует по историческим причинам и скорее мешает, чем помогает.

Впрочем, при записи значения `innerText` работает так же, как и `textContent`.

Свойство `hidden`

Как правило, видим или невидим узел, определяется через CSS, свойствами `display` или `visibility`.

В стандарте HTML5 предусмотрен специальный атрибут и свойство для этого: `hidden`.

Его поддерживают все современные браузеры, кроме IE10-.

В примере ниже второй и третий `<div>` скрыты:

```
<div>Текст</div>
<div hidden>С атрибутом hidden</div>
<div>Со свойством hidden</div>

<script>
  var lastDiv = document.body.children[2];
  lastDiv.hidden = true;
</script>
```

Технически, атрибут `hidden` работает так же, как `style="display:none"`. Но его проще поставить через JavaScript (меньше букв), и могут быть преимущества для скринридеров и прочих нестандартных браузеров.

Для старых IE тоже можно сделать, чтобы свойство поддерживалось, мы ещё вернёмся к этому далее в учебнике.

Исследование элементов

У DOM-узлов есть и другие свойства, зависящие от типа, например:

- `value` — значение для `INPUT`, `SELECT` или `TEXTAREA`
- `id` — идентификатор
- `href` — адрес ссылки
- ...многие другие...

Например:

```
<input type="text" value="значение">

<script>
  var input = document.body.children[0];

  alert( input.type ); // "text"
  alert( input.id ); // "input"
  alert( input.value ); // значение
</script>
```

Как узнать, какие свойства есть у данного типа элементов?

Это просто. Нужно либо посмотреть [СПИСОК ЭЛЕМЕНТОВ HTML5 ↗](#) и найти в нём интересующий вас элемент и прочитать секцию с `interface`.

Если же недосуг или интересуют особенности конкретного браузера — элемент всегда можно вывести в консоль вызовом `console.dir(элемент)`.

Метод `console.dir` выводит аргумент не в «красивом» виде, а как объект, который можно развернуть и исследовать.

Например:

```
// в консоли можно будет увидеть все свойства DOM-объекта document  
console.dir(document);
```

Итого

Основные свойства DOM-узлов:

nodeType

Тип узла. Самые популярные типы: "1" – для элементов и "3" – для текстовых узлов. Только для чтения.

nodeName/tagName

Название тега заглавными буквами. nodeName имеет специальные значения для узлов-неэлементов. Только для чтения.

innerHTML

Внутреннее содержимое узла-элемента в виде HTML. Можно изменять.

outerHTML

Полный HTML узла-элемента. При записи в elem.outerHTML переменная elem сохраняет старый узел.

nodeValue/data

Содержимое текстового узла или комментария. Свойство nodeValue также определено и для других типов узлов. Можно изменять.

Узлы DOM также имеют другие свойства, в зависимости от тега. Например, у INPUT есть свойства value и checked, а у A есть href и т.д. Мы рассмотрим их далее.

✓ Задачи

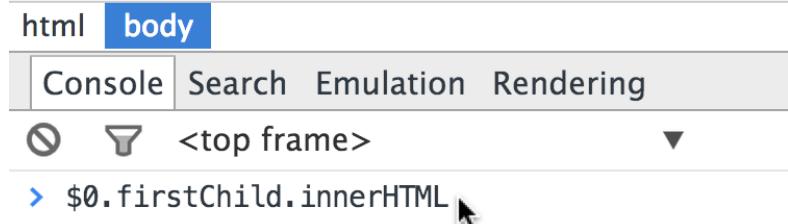
Что выведет код в консоли?

важность: 5

В браузере Chrome открыт HTML-документ.

Вы зашли во вкладку Elements и видите такую картинку:

```
▼ <html>
  <head></head>
  ▼ <body>
    <h1>Привет, мир!</h1>
  </body>
</html>
```



В настоящий момент выбран элемент `<body>`.

Что выведет код `$0.firstChild.innerHTML` в консоли?

[К решению](#)

В инлайн скрипте `lastChild.nodeType`

важность: 5

Что выведет скрипт на этой странице?

```
<!DOCTYPE HTML>
<html>

  <body>
    <script>
      alert( document.body.lastChild.nodeType );
    </script>
  </body>

</html>
```

[К решению](#)

Тег в комментарии

важность: 3

Что выведет этот код?

```
<script>
  var body = document.body;

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // что выведет?
</script>
```

[К решению](#)

Где в DOM-иерархии `document`?

важность: 4

Объектом какого класса является `document`?

Какое место он занимает в DOM-иерархии?

Наследует ли он `Node` или `Element`?

Воспользуйтесь для решения тем фактом, что DOM-узлы образуют стандартную прототипную иерархию классов.

[К решению](#)

Современный DOM: полифиллы

В старых IE, особенно в IE8 и ниже, ряд стандартных DOM-свойств не поддерживаются или поддерживаются плохо.

Если говорить о современных браузерах, то они тоже не все идут «в ногу», всегда какие-то современные возможности реализуются сначала в одном, потом в другом.

Но это не значит, что нужно ориентироваться на самый старый браузер из поддерживаемых!

Для того, чтобы не думать об устаревших браузерах, а писать современный код, который при этом работает везде, используют полифиллы.

Полифиллы

«Полифилл» (англ. polyfill) — это библиотека, которая добавляет в старые браузеры поддержку возможностей, которые в современных браузерах являются встроенными.

Один полифилл мы уже видели, когда изучали собственно JavaScript — это библиотека [ES5 shim ↗](#). Если её подключить, то в IE8- начинают работать многие возможности ES5. Работает она через модификацию стандартных объектов и их прототипов. Это типично для полифиллов.

В работе с DOM несовместимостей гораздо больше, как и способов их обхода.

Что делает полифилл?

Для примера добавим в DOM поддержку свойства `firstElementChild`, если её нет. Здесь речь, конечно, об IE8, в других браузерах оно и так поддерживается, но пример типовой.

Вот код для такого полифилла:

```
if (document.documentElement.firstElementChild === undefined) { // (1)

Object.defineProperty(Element.prototype, 'firstElementChild', { // (2)
  get: function() {
    var el = this.firstChild;
    do {
      if (el.nodeType === 1) {
        return el;
      }
      el = el.nextSibling;
    } while (el);

    return null;
  }
});
```

Если этот код запустить, то `firstElementChild` появится у всех элементов в IE8.

Общий вид этого полифилла довольно типичен. Обычно полифилл состоит из двух частей:

1. Проверка, есть ли встроенная возможность.
2. Эмуляция, если её нет.

Проверка встроенного свойства

Для проверки встроенной поддержки `firstElementChild` мы можем просто обратиться к `document.documentElement.firstElementChild`.

Если DOM-свойство `firstElementChild` поддерживается, то его значение не может быть `undefined`. Если детей нет — свойство равно `null`, но не `undefined`.

Сравним:

```
alert( document.head.previousSibling ); // null, поддержка есть
alert( document.head.blabla ); // undefined, поддержки нет
```

За счёт этого работает проверка в первой строке полифилла.

Важная тонкость — элемент, который мы тестируем, должен по стандарту поддерживать такое свойство.

Попытаемся, к примеру, проверить «поддержку» свойства `value`. У `input` оно есть, у `div` такого свойства нет:

```
var div = document.createElement('div');
var input = document.createElement('input');

alert( input.value ); // пустая строка, поддержка есть
alert( div.value ); // undefined, поддержки нет
```

Поддержка значений свойств

Если мы хотим проверить поддержку не свойства целиком, а некоторых его значений, то ситуация сложнее.

Например, нам интересно, поддерживает ли браузер `<input type="range">`. То есть, понятно, что свойство `type` у `input`, в целом, поддерживается, а вот конкретный тип `<input>`?

Для этого можно создать `<input>` с таким `type` и посмотреть, подействовал ли он.

Например:

```
<input type="radio">
<input type="no-such-type">

<script>
  alert( document.body.children[0].type ); // radio, поддерживается
  alert( document.body.children[1].type ); // text, не поддерживается
</script>
```

1. Первый `input` имеет `type="radio"`. Этот тип точно поддерживается, поэтому `input.type` имеет значение "radio", как и указано.
2. Второй `input` имеет `type="no-such-type"`. В качестве типа, для примера, специально указано заведомо неподдерживаемое значение. При этом `input.type` равен "text", таково значение по умолчанию. Мы можем прочитать его и увидеть, что поддержки нет.

Эта проверка работает, так как хоть в HTML-атрибут `type` и можно присвоить любую строку, но DOM-свойство `type` [по стандарту ↗](#) хранит реальный тип `input`а`.

Добавляем поддержку свойства

Если мы осуществили проверку и видим, что встроенной поддержки нет — полифилл должен её добавить.

Для этого вспомним, что DOM элементы описываются соответствующими JS-классами.

Например:

- `` — [HTMLLiElement ↗](#)

- <a> — [HTMLAnchorElement ↗](#)
- <body> — [HTMLBodyElement ↗](#)

Они наследуют, как мы видели ранее, от [HTMLElement ↗](#), который является общим родительским классом для HTML-элементов.

А [HTMLElement](#), в свою очередь, наследует от [Element ↗](#), который является общим родителем не только для HTML, но и для других DOM-структур, например для XML и SVG.

Для добавления нужной возможности берётся правильный класс и модифицируется его prototype.

Например, можно добавить всем элементам в прототип функцию:

```
Element.prototype.sayHi = function() {  
    alert( "Привет от " + this );  
}  
  
document.body.sayHi(); // Привет от [object HTMLBodyElement]
```

Сложнее — добавить свойство, но это тоже возможно, через `Object.defineProperty`:

```
Object.defineProperty(Element.prototype, 'lowerTag', {  
    get: function() {  
        return this.tagName.toLowerCase();  
    }  
});  
  
alert( document.body.lowerTag ); // body
```

⚠ Геттер-сеттер и IE8

В IE8 современные методы для работы со свойствами, такие как [Object.defineProperty ↗](#), [Object.getOwnPropertyDescriptor ↗](#) и другие не поддерживаются для произвольных объектов, но отлично работают для DOM-элементов.

Чем полифиллы и пользуются, «добавляя» в IE8 многие из современных методов DOM.

Какова поддержка свойства?

А нужен ли вообще полифилл? Какие браузеры поддерживают интересное нам свойство или метод?

Зачастую такая информация есть в справочнике MDN, например для метода `remove()`:
[https://developer.mozilla.org/en-US/docs/Web/API/ChildNode.remove ↗](https://developer.mozilla.org/en-US/docs/Web/API/ChildNode.remove) — табличка совместимости внизу.

Также бывает полезен сервис [http://caniuse.com ↗](http://caniuse.com), например для `elem.matches(css)`:
[http://caniuse.com/#feat=matchesselector ↗](http://caniuse.com/#feat=matchesselector).

Итого

Если вы поддерживаете устаревшие браузеры — и здесь речь идёт не только про старые IE, другие браузеры тоже обновляются не у всех мгновенно — не обязательно ограничивать себя в использовании современных возможностей.

Многие из них легко полифиллятся добавлением на страницу соответствующих библиотек.

Для поиска полифилла обычно достаточно ввести в поисковике "polyfill", и нужное свойство либо метод. Как правило, полифиллы идут в виде коллекций скриптов.

Полифиллы хороши тем, что мы просто подключаем их и используем везде современный DOM/JS, а когда старые браузеры окончательно отомрут — просто выкинем полифилл, без изменения кода.

Типичная схема работы полифилла DOM-свойства или метода:

- Создаётся элемент, который его, в теории, должен поддерживать.
- Соответствующее свойство сравнивается с `undefined`.
- Если его нет — модифицируется прототип, обычно это `Element.prototype` — в него дописываются новые геттеры и функции.

Другие полифиллы сделать сложнее. Например, полифилл, который хочет добавить в браузер поддержку элементов вида `<input type="range">`, может найти все такие элементы на странице и обработать их, меняя внешний вид и работу через JavaScript. Это возможно. Но если уже существующему `<input>` поменять `type` на `range` — полифилл не «подхватит» его автоматически.

Описанная ситуация нормальна. Не всегда полифилл обеспечивает идеальную поддержку наравне с родными свойствами. Но если мы не собираемся так делать, то подобный полифилл вполне подойдёт.

Один из лучших сервисов для полифиллов: [polyfill.io](https://cdn.polyfill.io/v1/polyfill.js?features=es6). Он даёт возможность вставлять на свою страницу скрипт с запросом к сервису, например:

```
<script src="//cdn.polyfill.io/v1/polyfill.js?features=es6"></script>
```

При запросе сервис анализирует заголовки, понимает, какая версия какого браузера к нему обратилась и возвращает скрипт-полифилл, добавляющий в браузер возможности, которых там нет. В параметре `features` можно указать, какие именно возможности нужны, в примере выше это функции стандарта ES6. Подробнее — см. [примеры](#) и [список возможностей](#).

Также есть и другие коллекции, как правило, полифиллы организованы в виде коллекции, из которой можно как выбрать отдельные свойства и функции, так и подключить всё вместе, пачкой.

Примеры полифиллов:

- <https://github.com/jonathantneal/polyfill> — ES5 вместе с DOM
- <https://github.com/termi/ES5-DOM-SHIM> — ES5 вместе с DOM
- <https://github.com/inexorabletash/polyfill> — ES5+ вместе с DOM

Более мелкие библиотеки, а также коллекции ссылок на них:

- [http://compatibility.shwups-cms.ch/en/polyfills/ ↗](http://compatibility.shwups-cms.ch/en/polyfills/)
- <http://html5please.com/#polyfill ↗>
- <https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills ↗>

Конечно, можно собрать и свою библиотеку полифиллов самостоятельно из различных коллекций, которые перечислены выше, а при необходимости и написать самому. В этой части учебника мы изучим ещё много методов работы с DOM, которые в этом помогут.

✓ Задачи

Полифилл для matches

важность: 5

Метод `elem.matches(css)` в некоторых старых браузерах поддерживается под старым именем `matchesSelector` или с префиксами, то есть: `webkitMatchesSelector` (старый Chrome, Safari), `mozMatchesSelector` (старый Firefox) или `Element.prototype.msMatchesSelector` (старый IE).

Создайте полифилл, который гарантирует стандартный синтаксис `elem.matches(css)` для всех браузеров.

[К решению](#)

Полифилл для closest

важность: 5

Метод `elem.closest(css)` для поиска ближайшего родителя, удовлетворяющего селектору `css`, не поддерживается некоторыми браузерами, например IE11-.

Создайте для него полифилл.

[К решению](#)

Полифилл для textContent

важность: 3

Свойство `textContent` не поддерживается IE8. Однако, там есть свойство `innerText`.

Создаёте полифилл, который проверяет поддержку свойства `textContent`, и если её нет — создаёт

его, используя `innerText`. Получится, что в IE8 «новое» свойство `textContent` будет «псевдонимом» для `innerText`.

Хотя свойство `innerText` и работает по-иному, нежели `textContent`, но в некоторых ситуациях они могут быть взаимозаменимы. Именно на них направлен полифилл.

[К решению](#)

Атрибуты и DOM-свойства

При чтении HTML браузер генерирует DOM-модель. При этом большинство стандартных HTML-атрибутов становятся свойствами соответствующих объектов.

Например, если тег выглядит как `<body id="page">`, то у объекта будет свойство `body.id = "page"`.

Но это преобразование — не один-в-один. Бывают ситуации, когда атрибут имеет одно значение, а свойство — другое. Бывает и так, что атрибут есть, а свойства с таким названием не создаются.

Если коротко — HTML-атрибуты и DOM-свойства обычно, но не всегда соответствуют друг другу, нужно понимать, что такое свойство и что такое атрибут, чтобы работать с ними правильно.

Свои DOM-свойства

Ранее мы видели некоторые встроенные свойства DOM-узлов. Но, технически, никто нас ими не ограничивает.

Узел DOM — это объект, поэтому, как и любой объект в JavaScript, он может содержать пользовательские свойства и методы.

Например, создадим в `document.body` новое свойство и запишем в него объект:

```
document.body.myData = {  
    name: 'Петр',  
    familyName: 'Петрович'  
};  
  
alert( document.body.myData.name ); // Петр
```

Можно добавить и новую функцию:

```
document.body.sayHi = function() {  
    alert( this.nodeName );  
}  
  
document.body.sayHi(); // BODY, выполнилась с правильным this
```

Нестандартные свойства и методы видны только в JavaScript и никак не влияют на отображение соответствующего тега.

Обратим внимание, пользовательские DOM-свойства:

- Могут иметь любое значение.
- Названия свойств *чувствительны* к регистру.
- Работают за счет того, что DOM-узлы являются объектами JavaScript.

Атрибуты

Элементам DOM, с другой стороны, соответствуют HTML-теги, у которых есть текстовые атрибуты.

Конечно, здесь речь именно об узлах-элементах, не о текстовых узлах или комментариях.

Доступ к атрибутам осуществляется при помощи стандартных методов:

- `elem.hasAttribute(name)` – проверяет наличие атрибута
- `elem.getAttribute(name)` – получает значение атрибута
- `elem.setAttribute(name, value)` – устанавливает атрибут
- `elem.removeAttribute(name)` – удаляет атрибут

Эти методы возвращают именно то значение, которое находится в HTML.

Также все атрибуты элемента можно получить с помощью свойства `elem.attributes`, которое содержит псевдо-массив объектов типа [Attr ↗](#).

В отличие от свойств, атрибуты:

- Всегда являются строками.
- Их имя *нечувствительно* к регистру (ведь это HTML)
- Видны в `innerHTML` (за исключением старых IE)

Рассмотрим различия между DOM-свойствами и атрибутами на примере HTML-кода:

```
<body>
  <div id="elem" about="Elephant" class="smiling"></div>
</body>
```

Пример ниже устанавливает атрибуты и демонстрирует их особенности.

```
<body>
  <div id="elem" about="Elephant"></div>

  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant', атрибут получен
    elem.setAttribute('Test', 123); // (2) атрибут Test установлен
    alert( document.body.innerHTML ); // (3) в HTML видны все атрибуты!

    var attrs = elem.attributes; // (4) можно получить коллекцию атрибутов
    for (var i = 0; i < attrs.length; i++) {
      alert( attrs[i].name + " = " + attrs[i].value );
    }
  </script>
</body>
```

При запуске кода выше обратите внимание:

1. `getAttribute('About')` — первая буква имени атрибута `About` написана в верхнем регистре, а в HTML — в нижнем, но это не имеет значения, так как имена нечувствительны к регистру.
2. Мы можем записать в атрибут любое значение, но оно будет превращено в строку. Объекты также будут автоматически преобразованы.
3. После добавления атрибута его можно увидеть в `innerHTML` элемента.
4. Коллекция `attributes` содержит все атрибуты в виде объектов со свойствами `name` и `value`.

Когда полезен доступ к атрибутам?

Когда браузер читает HTML и создаёт DOM-модель, то он создаёт свойства для всех *стандартных* атрибутов.

Например, свойства тега 'A' описаны в спецификации DOM: [HTMLAnchorElement ↗](#).

Например, у него есть свойство "href". Кроме того, он имеет "id" и другие свойства, общие для всех элементов, которые описаны в спецификации в [HTMLElement ↗](#).

Все стандартные свойства DOM синхронизируются с атрибутами, однако не всегда такая синхронизация происходит 1-в-1, поэтому иногда нам нужно значение именно из HTML, то есть атрибут.

Рассмотрим несколько примеров.

Ссылка «как есть» из атрибута href

Синхронизация не гарантирует одинакового значения в атрибуте и свойстве.

Для примера, посмотрим, что произойдет с атрибутом "href" при изменении свойства:

```
<a id="a" href="#"></a>
<script>
  a.href = '/';
  alert( 'атрибут:' + a.getAttribute('href') ); // '/'
  alert( 'свойство:' + a.href ); // полный URL
</script>
```

Это происходит потому, что атрибут может быть любым, а свойство `href`, [в соответствии со спецификацией W3C](#), должно быть полной ссылкой.

Стало быть, если мы хотим именно то, что в HTML, то нужно обращаться через атрибут.

Есть и другие подобные атрибуты

Кстати, есть и другие атрибуты, которые не копируются в точности. Например, DOM-свойство `input.checked` имеет логическое значение `true/false`, а HTML-атрибут `checked` — любое строковое, важно лишь его наличие.

Работа с `checked` через атрибут и свойство:

```
<input id="input" type="checkbox" checked>

<script>
  // работа с checked через атрибут
  alert( input.getAttribute('checked') ); // пустая строка
  input.removeAttribute('checked'); // снять галочку

  // работа с checked через свойство
  alert( input.checked ); // false <-- может быть только true/false
  input.checked = true; // поставить галочку
</script>
```

Исходное значение `value`

Изменение некоторых свойств обновляет атрибут. Но это скорее исключение, чем правило.

Чаще синхронизация — односторонняя: свойство зависит от атрибута, но не наоборот.

Например, при изменении свойства `input.value` атрибут `input.getAttribute('value')` не меняется:

```
<body>
  <input id="input" type="text" value="markup">
  <script>
    input.value = 'new'; // поменяли свойство
    alert( input.getAttribute('value') ); // 'markup', не изменилось!
  </script>
</body>
```

То есть, изменение DOM-свойства value на атрибут не влияет, он остаётся таким же.

А вот изменение атрибута обновляет свойство:

```
<body>
  <input id="input" type="text" value="markup">
  <script>
    input.setAttribute('value', 'new'); // поменяли атрибут
    alert( input.value ); // 'new', input.value изменилось!
  </script>
</body>
```

Эту особенность можно красиво использовать.

Получается, что атрибут `input.getAttribute('value')` хранит оригинальное (исходное) значение даже после того, как пользователь заполнил поле и свойство изменилось.

Например, можно взять изначальное значение из атрибута и сравнить со свойством, чтобы узнать, изменилось ли значение. А при необходимости и перезаписать свойство атрибутом, отменив изменения.

Классы в виде строки: `className`

Атрибуту "class" соответствует свойство `className`.

Так как слово "class" является зарезервированным словом в Javascript, то при проектировании DOM решили, что соответствующее свойство будет называться `className`.

Например:

```
<body class="main page">
  <script>
    // прочитать класс элемента
    alert( document.body.className ); // main page

    // поменять класс элемента
    document.body.className = "class1 class2";
  </script>
</body>
```

Кстати, есть и другие атрибуты, которые называются иначе, чем свойство. Например, атрибуту `for` (`<label for="...">`) соответствует свойство с названием `htmlFor`.

Классы в виде объекта: `classList`

Атрибут `class` — уникален. Ему соответствует аж целых два свойства!

Работать с классами как со строкой неудобно. Поэтому, кроме `className`, в современных браузерах есть свойство `classList`.

Свойство classList — это объект для работы с классами.

Оно поддерживается в IE начиная с IE10, но его можно эмулировать в IE8+, подключив мини-библиотеку [classList.js](#).

Методы classList:

- `elem.classList.contains("class")` — возвращает `true/false`, в зависимости от того, есть ли у элемента класс `class`.
- `elem.classList.add/remove("class")` — добавляет/удаляет класс `class`
- `elem.classList.toggle("class")` — если класса `class` нет, добавляет его, если есть — удаляет.

Кроме того, можно перебрать классы через `for`, так как `classList` — это псевдо-массив.

Например:

```
<body class="main page">
<script>
  var classList = document.body.classList;

  classList.remove('page'); // удалить класс
  classList.add('post'); // добавить класс

  for (var i = 0; i < classList.length; i++) { // перечислить классы
    alert( classList[i] ); // main, затем post
  }

  alert( classList.contains('post') ); // проверить наличие класса

  alert( document.body.className ); // main post, тоже работает
</script>
</body>
```

Нестандартные атрибуты

У каждого элемента есть некоторый набор стандартных свойств, например для `<a>` это будут `href`, `name`, `title`, а для `` это будут `src`, `alt`, и так далее.

Точный набор свойств описан в стандарте, обычно мы более-менее представляем, если пользуемся HTML, какие свойства могут быть, а какие — нет.

Для нестандартных атрибутов DOM-свойство не создаётся.

Например:

```
<div id="elem" href="http://ya.ru" about="Elephant"></div>

<script>
  alert( elem.id ); // elem
  alert( elem.about ); // undefined
</script>
```

Стандартным свойство является, лишь если оно описано в стандарте именно для этого элемента.

То есть, если назначить элементу `` атрибут `href`, то свойство `img.href` от этого не появится. Как, впрочем, и если назначить ссылке `<a>` атрибут `alt`:

```
<img id="img" href="test">
<a id="link" alt="test"></a>

<script>
  alert( img.href ); // undefined
  alert( link.alt ); // undefined
</script>
```

Нестандартные атрибуты иногда используют для CSS.

В примере ниже для показа «состояния заказа» используется атрибут `order-state`:

```
<style>
  .order[order-state="new"] {
    color: green;
  }

  .order[order-state="pending"] {
    color: blue;
  }

  .order[order-state="canceled"] {
    color: red;
  }
</style>

<div class="order" order-state="new">
  Новый заказ.
</div>

<div class="order" order-state="pending">
  Ожидаящий заказ.
</div>

<div class="order" order-state="canceled">
  Заказ отменён.
</div>
```

Почему именно атрибут? Разве нельзя было сделать классы `.order-state-new`, `.order-state-pending`, `.order-state-canceled`?

Конечно можно, но манипулировать атрибутом из JavaScript гораздо проще.

Например, если нужно отменить заказ, неважно в каком он состоянии сейчас — это сделает код:

```
div.setAttribute('order-state', 'canceled');
```

Для классов — нужно знать, какой класс у заказа сейчас. И тогда мы можем снять старый класс, и поставить новый:

```
div.classList.remove('order-state-new');
div.classList.add('order-state-canceled');
```

...то есть, требуется больше исходной информации и надо написать больше букв. Это менее удобно.

Проще говоря, значение атрибута — произвольная строка, значение класса — это «есть» или «нет», поэтому естественно, что атрибуты «мощнее» и бывают удобнее классов как в JS так и в CSS.

Свойство dataset, data-атрибуты

С помощью нестандартных атрибутов можно привязать к элементу данные, которые будут доступны в JavaScript.

Как правило, это делается при помощи атрибутов с названиями, начинающимися на data-, например:

```
<div id="elem" data-about="Elephant" data-user-location="street">
    По улице прошёлся слон. Весьма красив и толст был он.
</div>
<script>
    alert( elem.getAttribute('data-about') ); // Elephant
    alert( elem.getAttribute('data-user-location') ); // street
</script>
```

Стандарт HTML5 ↗ специально разрешает атрибуты data-* и резервирует их для пользовательских данных.

При этом во всех браузерах, кроме IE10-, к таким атрибутам можно обратиться не только как к атрибутам, но и как к свойствам, при помощи специального свойства dataset:

```
<div id="elem" data-about="Elephant" data-user-location="street">
    По улице прошёлся слон. Весьма красив и толст был он.
</div>
<script>
    alert( elem.dataset.about ); // Elephant
    alert( elem.dataset.userLocation ); // street
</script>
```

Обратим внимание — название data-user-location трансформировалось в dataset.userLocation. Дефис превращается в большую букву.

Полифилл для атрибута hidden

Для старых браузеров современные атрибуты иногда нуждаются в полифилле. Как правило, такой полифилл включает в себя не только JavaScript, но и CSS.

Например, свойство/атрибут hidden не поддерживается в IE11.

Этот атрибут должен прятать элемент, действие весьма простое, для его поддержки в HTML достаточно такого CSS:

```
<style>
  [hidden] { display: none }
</style>

<div>Текст</div>
<div hidden>С атрибутом hidden</div>
<div id="last">Со свойством hidden</div>

<script>
  last.hidden = true;
</script>
```

Если запустить в IE11- пример выше, то `<div hidden>` будет скрыт, а вот последний `div`, которому поставили свойство `hidden` в JavaScript — по-прежнему виден.

Это потому что CSS «не видит» присвоенное свойство, нужно синхронизировать его в атрибут.

Вот так — уже работает:

```
<style>
  [hidden] { display: none }
</style>

<script>
  if (document.documentElement.hidden === undefined) {
    Object.defineProperty(Element.prototype, "hidden", {
      set: function(value) {
        this.setAttribute('hidden', value);
      },
      get: function() {
        return this.getAttribute('hidden');
      }
    });
  }
</script>

<div>Текст</div>
<div hidden>С атрибутом hidden</div>
<div id="last">Со свойством hidden</div>

<script>
  last.hidden = true;
</script>
```

«Особенности» IE8

Если вам нужна поддержка этих версий IE — есть пара нюансов.

1. Во-первых, версии IE8- синхронизируют все свойства и атрибуты, а не только стандартные:

```
document.body.setAttribute('my', 123);
alert( document.body.my ); // 123 в IE8-
```

При этом даже тип данных не меняется. Атрибут не становится строкой, как ему положено.

2. Ещё одна некорректность IE8-: для изменения класса нужно использовать именно свойство

`className`, вызов `setAttribute('class', ...)` не сработает.

Вывод из этого довольно прост — чтобы не иметь проблем в IE8, нужно использовать всегда только свойства, кроме тех ситуаций, когда нужны именно атрибуты. Впрочем, это в любом случае хорошая практика.

Итого

- Атрибуты — это то, что написано в HTML.
- Свойство — это то, что находится в свойстве DOM-объекта.

Таблица сравнений для атрибутов и свойств:

Свойства	Атрибуты
Любое значение	Строка
Названия регистрозависимы	Не чувствительны к регистру
Не видны в <code>innerHTML</code>	Видны в <code>innerHTML</code>

Синхронизация между атрибутами и свойствами:

- Стандартные свойства и атрибуты синхронизируются: установка атрибута автоматически ставит свойство DOM. Некоторые свойства синхронизируются в обе стороны.
- Бывает так, что свойство не совсем соответствует атрибуту. Например, «логические» свойства вроде `checked`, `selected` всегда имеют значение `true/false`, а в атрибут можно записать произвольную строку. Выше мы видели другие примеры на эту тему, например `href`.

Нестандартные атрибуты:

- Нестандартный атрибут (если забыть глюки старых IE) никогда не попадёт в свойство, так что для кросс-браузерного доступа к нему нужно обязательно использовать `getAttribute`.
- Атрибуты, название которых начинается с `data-`, можно прочитать через `dataset`. Эта возможность не поддерживается IE10-.

Для того, чтобы избежать проблем со старыми IE, а также для более короткого и понятного кода старайтесь везде использовать свойства, а атрибуты — только там, где это *действительно* нужно.

А *действительно* нужны атрибуты очень редко — лишь в следующих трёх случаях:

1. Когда нужно кросс-браузерно получить нестандартный HTML-атрибут.
2. Когда нужно получить «оригинальное значение» стандартного HTML-атрибута, например, `<input value="...">`.
3. Когда нужно получить список всех атрибутов, включая пользовательские. Для этого используется коллекция `attributes`.

Если вы хотите использовать собственные атрибуты в HTML, то помните, что атрибуты с именем, начинающимся на `data-` валидны в HTML5 и современные браузеры поддерживают доступ к ним через свойство `dataset`.

✓ Задачи

Получите пользовательский атрибут

важность: 5

1. Получите `div` в переменную.
2. Получите значение атрибута "data-widget-name" в переменную.
3. Выведите его.

Документ:

```
<body>

<div id="widget" data-widget-name="menu">Выберите жанр</div>

<script>
/* ... */
</script>
</body>
```

[Открыть в песочнице ↗](#)

[К решению](#)

Поставьте класс ссылкам

важность: 3

Сделайте желтыми внешние ссылки, добавив им класс `external`.

Все ссылки без `href`, без протокола и начинающиеся с `http://internal.com` считаются внутренними.

```
<style>
.external {
    background-color: yellow
}
</style>

<a name="list">список</a>
<ul>
<li><a href="http://google.com">http://google.com</a></li>
<li><a href="/tutorial">/tutorial.html</a></li>
<li><a href="local/path">local/path</a></li>
<li><a href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
<li><a href="http://nodejs.org">http://nodejs.org</a></li>
<li><a href="http://internal.com/test">http://internal.com/test</a></li>
</ul>
```

Результат:

список

- <http://google.com>
- </tutorial.html>
- <local/path>
- <ftp://ftp.com/my.zip>
- <http://nodejs.org>
- <http://internal.com/test>

[К решению](#)

Методы `contains` и `compareDocumentPosition`

Если есть два элемента, то иногда бывает нужно понять, лежит ли один из них выше другого, то есть является ли его предком.

Обычные поисковые методы здесь не дают ответа, но есть два специальных. Они используются редко, но когда подобная задача встаёт, то знание метода может сэкономить много строк кода.

Метод `contains` для проверки на вложенность

Синтаксис:

```
var result = parent.contains(child);
```

Возвращает `true`, если `parent` содержит `child` или `parent == child`.

Метод `compareDocumentPosition` для порядка узлов

Бывает, что у нас есть два элемента, к примеру, `` в списке, и нужно понять, какой из них выше другого.

Метод `compareDocumentPosition` — более мощный, чем `contains`, он предоставляет одновременно информацию и о содержании и об относительном порядке элементов.

Синтаксис:

```
var result = nodeA.compareDocumentPosition(nodeB);
```

Возвращаемое значение — битовая маска (см. [Побитовые операторы](#)), биты в которой означают следующее:

Биты	Число	Значение
------	-------	----------

000000	0	nodeA и nodeB — один и тот же узел
000001	1	Узлы в разных документах (или один из них не в документе)
000010	2	nodeB предшествует nodeA (в порядке обхода документа)
000100	4	nodeA предшествует nodeB
001000	8	nodeB содержит nodeA
010000	16	nodeA содержит nodeB
100000	32	Зарезервировано для браузера

Понятие «предшествует» — означает не только «предыдущий сосед при общем родителе», но и имеет более общий смысл: «раньше встречается в порядке [прямого обхода](#) дерева документа».

Могут быть и сочетания битов. Примеры реальных значений:

```

<p>...</p>
<ul>
  <li>1.1</li>
</ul>

<script>
  var p = document.body.children[0];
  var ul = document.body.children[1];
  var li = ul.children[0];

  // 1. <ul> находится после <p>
  alert( ul.compareDocumentPosition(p) ); // 2 = 10

  // 2. <p> находится до <ul>
  alert( p.compareDocumentPosition(ul) ); // 4 = 100

  // 3. <ul> родитель <li>
  alert( ul.compareDocumentPosition(li) ); // 20 = 10100

  // 4. <ul> потомок <body>
  alert( ul.compareDocumentPosition(document.body) ); // 10 = 1010
</script>

```

Более подробно:

1. Узлы не вложены один в другой, поэтому стоит только бит «предшествования», отсюда **10**.
2. То же самое, но обратный порядок узлов, поэтому **100**.
3. Здесь стоят сразу два бита: **10100** означает, что **ul** одновременно содержит **li** и является его предшественником, то есть при прямом обходе дерева документа сначала встречается **ul**, а потом **li**.
4. Аналогично предыдущему, **1010** означает, что **document.body** содержит **ul** и предшествует ему.

Перевод в двоичную систему

Самый простой способ самостоятельно посмотреть, как число выглядит в 2-ной системе — вызвать для него `toString(2)`, например:

```
var x = 20;  
alert( x.toString(2) ); // "10100"
```

Или так:

```
alert( 20..toString(2) );
```

Здесь после `20` две точки, так как если одна, то JS подумает, что после неё десятичная часть — будет ошибка.

Проверить конкретное условие, например, «`nodeA` содержит `nodeB`», можно при помощи битовых операций, в данном случае: `nodeA.compareDocumentPosition(nodeB) & 16`, например:

```
<ul>  
  <li>1</li>  
</ul>  
  
<script>  
  var body = document.body;  
  var li = document.body.children[0].children[0];  
  
  if (body.compareDocumentPosition(li) & 16) {  
    alert( body + ' содержит ' + li );  
  }  
</script>
```

Более подробно о битовых масках: [Побитовые операторы](#).

Поддержка в IE8-

В IE8- поддерживаются свои, нестандартные, метод и свойство:

`nodeA.contains(nodeB)` ↗

Результат: `true`, если `nodeA` содержит `nodeB`, а также в том случае, если `nodeA == nodeB`.

`node.sourceIndex` ↗

Номер элемента `node` в порядке прямого обхода дерева. Только для узлов-элементов.

На их основе можно написать полифилл для `compareDocumentPosition`:

```
// код с http://compatibility.shwups-cms.ch/en/polyfills/?&id=82
(function() {
  var el = document.documentElement;
  if (!el.compareDocumentPosition && el.sourceIndex !== undefined) {

    Element.prototype.compareDocumentPosition = function(other) {
      return (this != other && this.contains(other) && 16) +
        (this != other && other.contains(this) && 8) +
        (this.sourceIndex >= 0 && other.sourceIndex >= 0 ?
          (this.sourceIndex < other.sourceIndex && 4) +
          (this.sourceIndex > other.sourceIndex && 2) : 1
        ) + 0;
    }
  }
})();
```

С этим полифиллом метод доступен для элементов во всех браузерах.

Итого

- Для проверки, является ли один узел предком другого, достаточно метода `nodeA.contains(nodeB)`.
- Для расширенной проверки на предшествование есть метод `compareDocumentPosition`.
- Для IE8 нужен полифилл для `compareDocumentPosition`.

Добавление и удаление узлов

Изменение DOM — ключ к созданию «живых» страниц.

В этой главе мы рассмотрим, как создавать новые элементы «на лету» и заполнять их данными.

Пример: показ сообщения

В качестве примера рассмотрим добавление сообщения на страницу, чтобы оно было оформлено красивее чем обычный `alert`.

HTML-код для сообщения:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert">
  <strong>Ура!</strong> Вы прочитали это важное сообщение.
</div>
```

Ура! Вы прочитали это важное сообщение.

Создание элемента

Для создания элементов используются следующие методы:

document.createElement(tag)

Создает новый элемент с указанным тегом:

```
var div = document.createElement('div');
```

document.createTextNode(text)

Создает новый *текстовый* узел с данным текстом:

```
var textElem = document.createTextNode('Тут был я');
```

Создание сообщения

В нашем случае мы хотим сделать DOM-элемент `div`, дать ему классы и заполнить текстом:

```
var div = document.createElement('div');
div.className = "alert alert-success";
div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение。";
```

После этого кода у нас есть готовый DOM-элемент. Пока что он присвоен в переменную `div`, но не виден, так как никак не связан со страницей.

Добавление элемента: `appendChild`, `insertBefore`

Чтобы DOM-узел был показан на странице, его необходимо вставить в `document`.

Для этого первым делом нужно решить, куда мы будем его вставлять. Предположим, что мы решили, что вставлять будем в некий элемент `parentElem`, например `var parentElem = document.body`.

Для вставки внутрь `parentElem` есть следующие методы:

parentElem.appendChild(elem)

Добавляет `elem` в конец дочерних элементов `parentElem`.

Следующий пример добавляет новый элемент в конец ``:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  var newLi = document.createElement('li');
  newLi.innerHTML = 'Привет, мир!';

  list.appendChild(newLi);
</script>
```

parentElem.insertBefore(elem, nextSibling)

Вставляет elem в коллекцию детей parentElem, перед элементом nextSibling.

Следующий код вставляет новый элемент перед вторым :

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  var newLi = document.createElement('li');
  newLi.innerHTML = 'Привет, мир!';

  list.insertBefore(newLi, list.children[1]);
</script>
```

Для вставки элемента в начало достаточно указать, что вставлять будем перед первым потомком:

```
list.insertBefore(newLi, list.firstChild);
```

У читателя, который посмотрит на этот код внимательно, наверняка возникнет вопрос: «А что, если list вообще пустой, в этом случае list.firstChild = null, произойдёт ли вставка?»

Ответ — да, произойдёт.

Дело в том, что если вторым аргументом указать null, то insertBefore сработает как appendChild:

```
parentElem.insertBefore(elem, null);
// то же, что и:
parentElem.appendChild(elem)
```

Так что insertBefore универсален.

На заметку:

Все методы вставки возвращают вставленный узел.

Например, `parentElem.appendChild(elem)` возвращает `elem`.

Пример использования

Добавим сообщение в конец `<body>`:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<body>
  <h3>Моя страница</h3>
</body>

<script>
  var div = document.createElement('div');
  div.className = "alert alert-success";
  div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

  document.body.appendChild(div);
</script>
```

Моя страница

Ура! Вы прочитали это важное сообщение.

...А теперь — в начало `<body>`:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<body>
  <h3>Моя страница</h3>
</body>

<script>
  var div = document.createElement('div');
  div.className = "alert alert-success";
  div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

  document.body.insertBefore(div, document.body.firstChild);
</script>
```

Ура! Вы прочитали это важное сообщение.

Моя страница

Клонирование узлов: `cloneNode`

А как бы вставить второе похожее сообщение?

Конечно, можно сделать функцию для генерации сообщений и поместить туда этот код, но в ряде случаев гораздо эффективнее — **клонировать** существующий `div`, а потом изменить текст внутри. В частности, если элемент большой, то клонировать его будет гораздо быстрее, чем пересоздавать.

Вызов `elem.cloneNode(true)` создаст «глубокую» копию элемента — вместе с атрибутами, включая подэлементы. Если же вызвать с аргументом `false`, то он копия будет без подэлементов, но это нужно гораздо реже.

Копия сообщения

Пример со вставкой копии сообщения:

```

<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<body>
  <h3>Моя страница</h3>
</body>

<script>
  var div = document.createElement('div');
  div.className = "alert alert-success";
  div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

  document.body.insertBefore(div, document.body.firstChild);

  // создать копию узла
  var div2 = div.cloneNode(true);
  // копию можно подправить
  div2.querySelector('strong').innerHTML = 'Супер!';
  // вставим её после текущего сообщения
  div.parentNode.insertBefore(div2, div.nextSibling);
</script>

```

Ура! Вы прочитали это важное сообщение.

Супер! Вы прочитали это важное сообщение.

Моя страница

Обратите внимание на последнюю строку, которая вставляет div2 после div:

```
div.parentNode.insertBefore(div2, div.nextSibling);
```

- Для вставки нам нужен будущий родитель. Мы, возможно, не знаем, где точно находится div (или не хотим зависеть от того, где он), но если нужно вставить рядом с div, то родителем определённо будет div.parentNode.
- Мы хотели бы вставить *после* div, но метода insertAfter нет, есть только insertBefore, поэтому вставляем *перед* его правым соседом div.nextSibling.

Удаление узлов: removeChild

Для удаления узла есть два метода:

parentElem.removeChild(elem)

Удаляет elem из списка детей parentElem.

```
parentElem.replaceChild(newElem, elem)
```

Среди детей `parentElem` удаляет `elem` и вставляет на его место `newElem`.

Оба этих метода возвращают удаленный узел, то есть `elem`. Если нужно, его можно вставить в другое место DOM тут же или в будущем.

i На заметку:

Если вы хотите *переместить* элемент на новое место — не нужно его удалять со старого.

Все методы вставки автоматически удаляют вставляемый элемент со старого места.

Конечно же, это очень удобно.

Например, поменяем элементы местами:

```
<div>Первый</div>
<div>Второй</div>
<script>
  var first = document.body.children[0];
  var last = document.body.children[1];

  // нет необходимости в предварительном removeChild(last)
  document.body.insertBefore(last, first); // поменять местами
</script>
```

i Метод `remove`

В современном стандарте есть также метод `elem.remove()` ↗, который удаляет элемент напрямую, не требуя ссылки на родителя. Это зачастую удобнее, чем `removeChild`.

Он поддерживается во всех современных браузерах, кроме IE11-. Впрочем, легко подключить или даже сделать полифилл.

Удаление сообщения

Сделаем так, что через секунду сообщение пропадёт:

```

<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<body>
  <h3>Сообщение пропадёт через секунду</h3>
</body>

<script>
  var div = document.createElement('div');
  div.className = "alert alert-success";
  div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

  document.body.appendChild(div);

  setTimeout(function() {
    div.parentNode.removeChild(div);
  }, 1000);
</script>

```

Текстовые узлы для вставки текста

При работе с сообщением мы использовали только узлы-элементы и `innerHTML`.

Но и текстовые узлы тоже имеют интересную область применения!

Если текст для сообщения нужно показать именно как текст, а не как HTML, то можно обернуть его в текстовый узел.

Например:

```

<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  var div = document.createElement('div');
  div.className = "alert alert-success";
  document.body.appendChild(div);

  var text = prompt("Введите текст для сообщения", "Жили были <a> и <b>!");

  // вставится именно как текст, без HTML-обработки
  div.appendChild(document.createTextNode(text));
</script>

```

В современных браузерах (кроме IE8-) в качестве альтернативы можно использовать присвоение `textContent`.

Итого

Методы для создания узлов:

- `document.createElement(tag)` — создает элемент
- `document.createTextNode(value)` — создает текстовый узел
- `elem.cloneNode(deep)` — клонирует элемент, если `deep == true`, то со всеми потомками, если `false` — без потомков.

Вставка и удаление узлов:

- `parent.appendChild(elem)`
- `parent.insertBefore(elem, nextSibling)`
- `parent.removeChild(elem)`
- `parent.replaceChild(newElem, elem)`

Все эти методы возвращают `elem`.

Методы для изменения DOM также описаны в спецификации [DOM Level 1 ↗](#).

✓ Задачи

createTextNode vs innerHTML

важность: 5

Есть *пустой* узел DOM `elem`.

Одинаковый ли результат дадут эти скрипты?

Первый:

```
elem.appendChild(document.createTextNode(text));
```

Второй:

```
elem.innerHTML = text;
```

Если нет — дайте пример значения `text`, для которого результат разный.

[К решению](#)

Удаление элементов

важность: 5

Напишите полифилл для метода `remove` для старых браузеров.

Вызов `elem.remove()`:

- Если у `elem` нет родителя — ничего не делает.
- Если есть — удаляет элемент из родителя.

```
<div>Это</div>
<div>Все</div>
<div>Элементы DOM</div>

<script>
  /* ваш код полифилла */

  var elem = document.body.children[0];

  elem.remove(); // <-- вызов должен удалить элемент
</script>
```

[К решению](#)

insertAfter

важность: 5

Напишите функцию `insertAfter(elem, refElem)`, которая добавит `elem` после узла `refElem`.

```
<div>Это</div>
<div>Элементы</div>

<script>
  var elem = document.createElement('div');
  elem.innerHTML = '<b>Новый элемент</b>';

  function insertAfter(elem, refElem) { /* ваш код */ }

  var body = document.body;

  // вставить elem после первого элемента
  insertAfter(elem, body.firstChild); // <-- должно работать

  // вставить elem за последним элементом
  insertAfter(elem, body.lastChild); // <-- должно работать
</script>
```

[К решению](#)

removeChildren

важность: 5

Напишите функцию `removeChildren`, которая удаляет всех потомков элемента.

```
<table id="table">
  <tr>
    <td>Это</td>
    <td>Все</td>
    <td>Элементы DOM</td>
  </tr>
</table>

<ol id="ol">
  <li>Вася</li>
  <li>Петя</li>
  <li>Маша</li>
  <li>Даша</li>
</ol>

<script>
  function removeChildren(elem) { /* ваш код */ }

  removeChildren(table); // очищает таблицу
  removeChildren(ol); // очищает список
</script>
```

[К решению](#)

Почему остаётся «aaa» ?

важность: 1

Запустите этот пример. Почему вызов `removeChild` не удалил текст "aaa"?

```
<table>
  aaa
  <tr>
    <td>Test</td>
  </tr>
</table>

<script>
  var table = document.body.children[0];

  alert( table ); // таблица, пока всё правильно

  document.body.removeChild(table);
  // почему в документе остался текст?
</script>
```

[К решению](#)

Создать список

важность: 4

Напишите интерфейс для создания списка.

Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью `prompt`.
2. Создавайте пункт и добавляйте его к UL.
3. Процесс прерывается, когда пользователь нажимает ESC или вводит пустую строку.

Все элементы должны создаваться динамически.

Если посетитель вводит теги — пусть в списке они показываются как обычный текст.

[Демо в новом окне ↗](#)

[К решению](#)

Создайте дерево из объекта

важность: 5

Напишите функцию, которая создаёт вложенный список UL/LI (дерево) из объекта.

Например:

```
var data = {  
    "Рыбы": {  
        "Форель": {},  
        "Щука": {}  
    },  
  
    "Деревья": {  
        "Хвойные": {  
            "Лиственница": {},  
            "Ель": {}  
        },  
        "Цветковые": {  
            "Берёза": {},  
            "Тополь": {}  
        }  
    }  
};
```

Синтаксис:

```
var container = document.getElementById('container');  
createTree(container, data); // создаёт
```

Результат (дерево):

- Рыбы
 - Форель
 - Щука
- Деревья
 - Хвойные
 - Лиственница
 - Ель
 - Цветковые
 - Берёза
 - Тополь

Выберите один из двух способов решения этой задачи:

1. Создать строку, а затем присвоить через `container.innerHTML`.
2. Создавать узлы через методы DOM.

Если получится — сделайте оба.

P.S. Желательно, чтобы в дереве не было лишних элементов, в частности — пустых `` на нижнем уровне.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Дерево

важность: 5

Есть дерево [в песочнице](#) ↗.

Напишите код, который добавит каждому элементу списка `` количество вложенных в него элементов. Узлы нижнего уровня, без детей — пропускайте.

Результат:

- Животные [9]
 - Млекопитающие [4]
 - Коровы
 - Ослы
 - Собаки
 - Тигры
 - Другие [3]
 - Змеи
 - Птицы
 - Ящерицы
- Рыбы [5]
 - Аквариумные [2]
 - Гуппи
 - Скалярии
 - Морские [1]
 - Морская форель

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Создать календарь в виде таблицы

важность: 4

Напишите функцию, которая умеет генерировать календарь для заданной пары (месяц, год).

Календарь должен быть таблицей, где каждый день — это TD. У таблицы должен быть заголовок с названиями дней недели, каждый день — TH.

Синтаксис: `createCalendar(id, year, month)`.

Такой вызов должен генерировать текст для календаря месяца `month` в году `year`, а затем помещать его внутрь элемента с указанным `id`.

Например: `createCalendar("cal", 2012, 9)` сгенерирует в `<div id='cal'></div>` следующий календарь:

пн	вт	ср	чт	пт	сб	вс
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

P.S. Достаточно сгенерировать календарь, кликабельным его делать не нужно.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Часики с использованием «setInterval»

важность: 4

Создайте цветные часики как в примере ниже:

19:17:41

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Мультивставка: insertAdjacentHTML и DocumentFragment

Обычные методы вставки работают с одним узлом. Но есть и способы вставлять множество узлов одновременно.

Оптимизация вставки в документ

Рассмотрим задачу: сгенерировать список UL/LI.

Есть две возможных последовательности:

1. Сначала вставить UL в документ, а потом добавить к нему LI:

```
var ul = document.createElement('ul');
document.body.appendChild(ul); // сначала в документ
for (...) ul.appendChild(li); // потом узлы
```

2. Полностью создать список «вне DOM», а потом — вставить в документ:

```
var ul = document.createElement('ul');
for(...) ul.appendChild(li); // сначала вставить узлы
document.body.appendChild(ul); // затем в документ
```

Как ни странно, между этими последовательностями есть разница. В большинстве браузеров, второй вариант — быстрее.

Почему же? Иногда говорят: «потому что браузер перерисовывает каждый раз при добавлении элемента». Это не так. Дело вовсе не в перерисовке.

Браузер достаточно «умён», чтобы ничего не перерисовывать понапрасну. В большинстве случаев процессы перерисовки и сопутствующие вычисления будут отложены до окончания работы скрипта, и на тот момент уже совершенно без разницы, в какой последовательности были изменены узлы.

Тем не менее, при вставке узла происходят разные внутренние события и обновления внутренних структур данных, скрытые от наших глаз.

Что именно происходит — зависит от конкретной, внутренней браузерной реализации DOM, но это отнимает время. Конечно, браузеры развиваются и стараются свести лишние действия к минимуму.

Добавление множества узлов

Продолжим работать со вставкой узлов.

Рассмотрим случай, когда в документе уже есть большой список UL. И тут понадобилось срочно добавить еще 20 элементов LI.

Как это сделать?

Если новые элементы пришли в виде строки, то можно попробовать добавить их так:

```
ul.innerHTML += "<li>1</li><li>2</li>...";
```

Но операцию `ul.innerHTML += "..."` можно по-другому переписать как `ul.innerHTML = ul.innerHTML + "...".` Иначе говоря, она *не прибавляет, а заменяет* всё содержимое списка на дополненную строку. Это и нехорошо с точки зрения производительности, но и будут побочные эффекты. В частности, все внешние ресурсы (картинки) внутри перезаписываемого `innerHTML` будут загружены заново. Если в каких-то переменных были ссылки на элементы списка — они станут неверны, так как содержимое полностью заменяется. В общем, так лучше не делать.

А если нужно вставить в середину списка? Здесь `innerHTML` вообще не поможет.

Можно, конечно, вставить строку во временный DOM-элемент и перенести оттуда элементы, но есть и гораздо лучший вариант: метод `insertAdjacentHTML!`

insertAdjacent*

Метод [insertAdjacentHTML ↗](#) позволяет вставлять произвольный HTML в любое место документа, в том числе *и между узлами!*

Он поддерживается всеми браузерами, кроме Firefox меньше версии 8, ну а там его можно

эмулировать.

Синтаксис:

```
elem.insertAdjacentHTML(where, html);
```

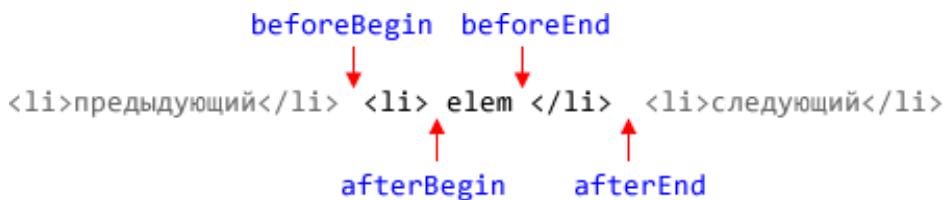
html

Строка HTML, которую нужно вставить

where

Куда по отношению к elem вставлять строку. Всего четыре варианта:

1. `beforeBegin` — перед elem.
2. `afterBegin` — внутрь elem, в самое начало.
3. `beforeEnd` — внутрь elem, в конец.
4. `afterEnd` — после elem.



Например, вставим пропущенные элементы списка *перед* `5`:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>5</li>
</ul>

<script>
  var ul = document.body.children[0];
  var li5 = ul.children[2];

  li5.insertAdjacentHTML("beforeBegin", "<li>3</li><li>4</li>");
</script>
```

Единственный недостаток этого метода — он не работает в Firefox до версии 8. Но его можно легко добавить, используя [полифилл insertAdjacentHTML для Firefox](#).

У этого метода есть «близнецы-братья», которые поддерживаются везде, кроме Firefox, но в него они добавляются тем же полифиллом:

- `elem.insertAdjacentElement(where, newElem)` — вставляет в произвольное место не строку HTML, а элемент newElem.
- `elem.insertAdjacentText(where, text)` — создаёт текстовый узел из строки text и вставляет его в указанное место относительно elem.

Синтаксис этих методов, за исключением последнего параметра, полностью совпадает с

`insertAdjacentHTML`. Вместе они образуют «универсальный швейцарский нож» для вставки чего угодно куда угодно.

DocumentFragment

Важно для старых браузеров

Оптимизация, о которой здесь идёт речь, важна в первую очередь для старых браузеров, включая IE9-. В современных браузерах эффект от нее, как правило, небольшой, а иногда может быть и отрицательным.

До этого мы говорили о вставке строки в DOM. А что делать в случае, когда надо в существующий UL вставить много *DOM-элементов*?

Можно вставлять их один за другим, вызовом `insertBefore/appendChild`, но при этом получится много операций с большим живым документом.

Вставить пачку узлов единовременно поможет `DocumentFragment`. Это особенный кроссбраузерный DOM-объект, который похож на обычный DOM-узел, но им не является.

Синтаксис для его создания:

```
var fragment = document.createDocumentFragment();
```

В него можно добавлять другие узлы.

```
fragment.appendChild(node);
```

Его можно клонировать:

```
fragment.cloneNode(true); // клонирование с подэлементами
```

У `DocumentFragment` нет обычных свойств DOM-узлов, таких как `innerHTML`, `tagName` и т.п. Это не узел.

Его «Фишка» заключается в том, что когда `DocumentFragment` вставляется в DOM — то он исчезает, а вместо него вставляются его дети. Это свойство является уникальной особенностью `DocumentFragment`.

Например, если добавить в него много LI, и потом вызвать `ul.appendChild(fragment)`, то фрагмент растворится, и в DOM вставятся именно LI, причём в том же порядке, в котором были во фрагменте.

Псевдокод:

```
// хотим вставить в список UL много LI
// делаем вспомогательный DocumentFragment
var fragment = document.createDocumentFragment();

for (цикл по li) {
    fragment.appendChild(list[i]); // вставить каждый LI в DocumentFragment
}

ul.appendChild(fragment); // вместо фрагмента вставятся элементы списка
```

В современных браузерах эффект от такой оптимизации может быть различным, а на небольших документах иногда и отрицательным.

Понять текущее положение вещей вы можете, запустив следующий [небольшой бенчмарк ↗](#).

append/prepend, before/after, replaceWith

Сравнительно недавно в [стандарте ↗](#) появились методы, которые позволяют вставить что угодно и куда угодно.

Синтаксис:

- `node.append(...nodes)` — вставляет `nodes` в конец `node`,
- `node.prepend(...nodes)` — вставляет `nodes` в начало `node`,
- `node.after(...nodes)` — вставляет `nodes` после узла `node`,
- `node.before(...nodes)` — вставляет `nodes` перед узлом `node`,
- `node.replaceWith(...nodes)` — вставляет `nodes` вместо `node`.

Эти методы ничего не возвращают.

Во всех этих методах `nodes` — DOM-узлы или строки, в любом сочетании и количестве. Причём строки вставляются именно как текстовые узлы, в отличие от `insertAdjacentHTML`.

Пример (с полифиллом):

```

<html>
  <head>
    <meta charset="utf-8">
    <script src="https://cdn.polyfill.io/v1/polyfill.js?features=Element.prototype.append,Element.prototype
  </head>

  <body>
    <script>
      // добавим элемент в конец <body>
      var p = document.createElement('p');
      document.body.append(p);

      var em = document.createElement('em');
      em.append('Мир!');

      // вставить в параграф текстовый и обычный узлы
      p.append("Привет, ", em);

      // добавить элемент после <p>
      p.after(document.createElement('hr'))
    </script>
  </body>
</html>

```

Привет, *Mир!*

Итого

- Манипуляции, меняющие структуру DOM (вставка, удаление элементов), как правило, быстрее с отдельным маленьkim узлом, чем с большим DOM, который находится в документе.

Конкретная разница зависит от внутренней реализации DOM в браузере.

- Семейство методов для вставки HTML/элемента/текста в произвольное место документа:

- elem.insertAdjacentHTML(where, html)
- elem.insertAdjacentElement(where, node)
- elem.insertAdjacentText(where, text)

Два последних метода не поддерживаются в Firefox, на момент написания текста, но есть небольшой полифилл [insertAdjacentFF.js](#), который добавляет их. Конечно, он нужен только для Firefox.

- DocumentFragment позволяет минимизировать количество вставок в большой живой DOM. Эта оптимизация особо эффективна в старых браузерах, в новых эффект от неё меньше или наоборот отрицательный.

Элементы сначала вставляются в него, а потом — он вставляется в DOM. При вставке DocumentFragment «растворяется», и вместо него вставляются содержащиеся в нём узлы.

`DocumentFragment`, в отличие от `insertAdjacent*`, работает с коллекцией DOM-узлов.

- Современные методы, работают с любым количеством узлов и текста, желателен полифилл:
 - `append/prepend` — вставка в конец/начало.
 - `before/after` — вставка после/перед.
 - `replaceWith` — замена.

✓ Задачи

Вставьте элементы в конец списка

важность: 5

Напишите код для вставки текста `html` в конец списка `ul` с использованием метода `insertAdjacentHTML`. Такая вставка, в отличие от присвоения `innerHTML+=`, не будет перезаписывать текущее содержимое.

Добавьте к списку ниже элементы `345`:

```
<ul>
  <li>1</li>
  <li>2</li>
</ul>
```

[К решению](#)

Отсортировать таблицу

важность: 5

Есть таблица:

Имя	Фамилия	Отчество	Возраст
Вася	Петров	Александрович	10
Петя	Иванов	Петрович	15
Владимир	Ленин	Ильич	9
...

Строк в таблице много: может быть 20, 50, 100.. Есть и другие элементы в документе.

Как бы вы предложили отсортировать содержимое таблицы по полю Возраст? Обдумайте алгоритм, реализуйте его.

Как сделать, чтобы сортировка работала как можно быстрее? А если в таблице 10000 строк (бывает и такое)?

P.S. Может ли здесь помочь `DocumentFragment`?

P.P.S. Если предположить, что у нас заранее есть массив данных для таблицы в JavaScript — что быстрее: отсортировать эту таблицу или сгенерировать новую?

[К решению](#)

Метод `document.write`

Метод `document.write` — один из наиболее древних методов добавления текста к документу.

У него есть существенные ограничения, поэтому он используется редко, но по своей сути он совершенно уникален и иногда, хоть и редко, может быть полезен.

Как работает `document.write`

Метод `document.write(str)` работает только пока HTML-страница находится в процессе загрузки. Он дописывает текст в текущее место HTML ещё до того, как браузер построит из него DOM.

HTML-документ ниже будет содержать 1 2 3.

```
<body>
  1
  <script>
    document.write(2);
  </script>
  3
</body>
```

Нет никаких ограничений на содержимое `document.write`.

Строка просто пишется в HTML-документ без проверки структуры тегов, как будто она всегда там была.

Например:

```
<script>
  document.write('<style> td { color: #F40 } </style>');
</script>
<table>
  <tr>
    <script>
      document.write('<td>')
    </script>
    Текст внутри TD.
    <script>
      document.write('</td>')
    </script>
  </tr>
</table>
```

Также существует метод `document.writeln(str)` — не менее древний, который добавляет после `str` символ перевода строки "`\n`".

Только до конца загрузки

Во время загрузки браузер читает документ и тут же строит из него DOM, по мере получения информации достраивая новые и новые узлы, и тут же отображая их. Этот процесс идет непрерывным потоком. Вы наверняка видели это, когда заходили на сайты в качестве посетителя — браузер зачастую отображает неполный документ, добавляя его новыми узлами по мере их получения.

Методы `document.write` и `document.writeln` пишут напрямую в текст документа, до того как браузер построит из него DOM, поэтому они могут записать в документ все, что угодно, любые стили и незакрытые теги.

Браузер учитёт их при построении DOM, точно так же, как учитывает очередную порцию HTML-текста.

Технически, вызвать `document.write` можно в любое время, однако, когда HTML загрузился, и браузер полностью построил DOM, документ становится «закрытым». Попытка дописать что-то в закрытый документ открывает его заново. При этом все текущее содержимое удаляется.

Текущая страница, скорее всего, уже загрузилась, поэтому если вы нажмёте на эту кнопку — её содержимое удалится:

Запустить `document.write('Пустая страница!')`

Из-за этой особенности `document.write` для загруженных документов не используют.

⚠ XHTML и `document.write`

В некоторых современных браузерах при получении страницы с заголовком `Content-Type: text/xml` или `Content-Type: text/xhtml+xml` включается «XML-режим» чтения документа. Метод `document.write` при этом не работает.

Это лишь одна из причин, по которой XML-режим обычно не используют.

Преимущества перед innerHTML

Метод `document.write` — динозавр, он существовал десятки ~~миллионов~~ лет назад. С тех пор, как появился и стал стандартным методом `innerHTML`, нужда в нём возникает редко, но некоторые преимущества, всё же, есть.

- Метод `document.write` работает быстрее, фактически это самый быстрый способ добавить на страницу текст, сгенерированный скриптом.

Это естественно, ведь он не модифицирует существующий DOM, а пишет в текст страницы до его генерации.

- Метод `document.write` вставляет любой текст на страницу «как есть», в то время как `innerHTML` может вписать лишь валидный HTML (при попытке подсунуть невалидный — браузер скорректирует его).

Эти преимущества являются скорее средством оптимизации, которое нужно использовать именно там, где подобная оптимизация нужна или уместна.

Однако, `document.write` по своей природе уникален: он добавляет текст «в текущее место документа», без всяких хитроумных DOM. Поэтому он бывает просто-напросто удобен, из-за чего его нередко используют не по назначению.

Антитипмер: реклама

Например, `document.write` используют для вставки рекламных скриптов и различных счетчиков, когда URL скрипта необходимо генерировать динамически, добавляя в него параметры из JavaScript, например:

```
<script>
  // в url указано текущее разрешение экрана посетителя
  var url = 'http://ads.com/buyme?screen=' + screen.width + "x" + screen.height;

  // загрузить такой скрипт прямо сейчас
  document.write('<script src="' + url + '"></scr' + 'ipt>');
</script>
```

На заметку:

Закрывающий тег `</script>` в строке разделён, чтобы браузер не увидел `</script>` и не посчитал его концом скрипта.

Также используют запись:

```
document.write('<script src="' + url + '"><\script>');
```

Здесь `<\script>` вместо `</script>`: обратный слеш \ обычно используется для вставки спецсимволов типа \n, а если такого спецсимвола нет, в данном случае \/ не является спецсимволом, то будет проигнорирован. Так что получается такой альтернативный способ безопасно вставить строку `</script>`.

Сервер, получив запрос с такими параметрами, обрабатывает его и, учитывая переданную информацию, генерирует текст скрипта, в котором обычно есть какой-то другой `document.write`, рисующий на этом месте баннер.

Проблема здесь в том, что загрузка такого скрипта блокирует отрисовку всей страницы.

То есть, дело даже не в самом `document.write`, а в том, что в страницу вставляется сторонний скрипт, а браузер устроен так, что пока он его не загрузит и не выполнит — он не будет дальше строить DOM и показывать документ.

Представим на минуту, что сервер ads.com, с которого грузится скрипт, работает медленно или вообще завис — зависнет и наша страница.

Что делать?

В современных браузерах у скриптов есть атрибуты `async` и `defer`, которые разрешают браузеру продолжать обработку страницы, но применить их здесь нельзя, так как рекламный скрипт захочет вызвать `document.write` именно на этом месте, и браузер не должен двигаться вперёд по документу.

Альтернатива — использовать другие техники вставки рекламы и счётчиков. Примеры вы можете увидеть в коде Google Analytics, Яндекс.Метрики и других.

Если это невозможно — применяют всякие хитрые оптимизации, например заменяют метод `document.write` своей функцией, и она уже разбирается со скриптами и баннерами.

Итого

Метод `document.write` (или `writeln`) пишет текст прямо в HTML, как будто он там всегда был.

- Этот метод редко используется, так как работает только из скриптов, выполняемых в процессе загрузки страницы.

Запуск после загрузки приведёт к очистке документа.

- Метод `document.write` очень быстр.

В отличие от установки `innerHTML` и DOM-методов, он не изменяет существующий документ, а работает на стадии текста, до того как DOM-структура сформирована.

- Иногда `document.write` используют для добавления скриптов с динамическим URL.

Рекомендуется избегать этого, так как браузер остановится на месте добавления скрипта и будет ждать его загрузки. Если скрипт будет тормозить, то и страница — тоже.

Поэтому желательно подключать внешние скрипты, используя вставку скрипта через DOM или `async/defer`. Современные системы рекламы и статистики так и делают.

Стили, `getComputedStyle`

Эта глава — о свойствах стиля, получении о них информации и изменении при помощи JavaScript.

Перед прочтением убедитесь, что хорошо знакомы с [блочной моделью CSS](#) и понимаете, что такое `padding`, `margin`, `border`.

Стили элемента: свойство `style`

Объект `element.style` дает доступ к стилю элемента на чтение и запись.

С его помощью можно изменять большинство CSS-свойств, например `element.style.width="100px"` работает так, как будто у элемента в атрибуте прописано `style="width:100px"`.



⚠ Единицы измерения обязательны в `style`

Об этом иногда забывают, но в `style` так же, как и в CSS, нужно указывать единицы измерения, например `px`.

Ни в коем случае не просто `elem.style.width = 100` — работать не будет.

Для свойств, названия которых состоят из нескольких слов, используется вот Такая Запись:

```
background-color  => elem.style.backgroundColor  
z-index          => elem.style.zIndex  
border-left-width => elem.style.borderLeftWidth
```

Пример использования `style`:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

⚠️ style.cssFloat вместо style.float

Исключением является свойство `float`. В старом стандарте JavaScript слово "float" было зарезервировано и недоступно для использования в качестве свойства объекта. Поэтому используется не `elem.style.float`, а `elem.style.cssFloat`.

ℹ️ Свойства с префиксами

Специфические свойства браузеров, типа `-moz-border-radius`, `-webkit-border-radius`, записываются следующим способом:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

То есть, каждый дефис даёт большую букву.

**Чтобы сбросить поставленный стиль, присваивают в `style` пустую строку:
`elem.style.width = ""`.**

При сбросе свойства `style` стиль будет взят из CSS.

Например, для того, чтобы спрятать элемент, можно присвоить: `elem.style.display = "none"`.

А вот чтобы показать его обратно — не обязательно явно указывать другой `display`, наподобие `elem.style.display = "block"`. Можно просто снять поставленный стиль: `elem.style.display = ""`.

```
// если запустить этот код, то <body> "мигнёт"
document.body.style.display = "none";

setTimeout(function() {
  document.body.style.display = "";
}, 1000);
```

Стиль в `style` находится в формате браузера, а не в том, в котором его присвоили.

Например:

```
<body>
<script>
  document.body.style.margin = '20px';
  alert( document.body.style.marginTop ); // 20px!

  document.body.style.color = '#abc';
  alert( document.body.style.color ); // rgb(170, 187, 204)
</script>
</body>
```

Обратите внимание на то, как браузер «распаковал» свойство `style.margin`, предоставив для чтения `style.marginTop`. То же самое произойдет и для `border`, `background` и т.д.

⚠ Свойство `style` мы используем лишь там, где не работают классы

В большинстве случаев внешний вид элементов задаётся классами. А JavaScript добавляет или удаляет их. Такой код красив и гибок, дизайн можно легко изменять.

Свойство `style` нужно использовать лишь там, где классы не подходят, например если точное значение цвета/отступа/высоты вычисляется в JavaScript.

Строка стилей `style.cssText`

Свойство `style` является специальным объектом, ему нельзя присваивать строку.

Запись `div.style="color:blue"` работать не будет. Но как же, всё-таки, поставить свойство стиля, если хочется задать его строкой?

Можно попробовать использовать атрибут: `elem.setAttribute("style", ...)`, но самым правильным и, главное, кросс-браузерным (с учётом старых IE) решением такой задачи будет использование свойства `style.cssText`.

Свойство `style.cssText` позволяет поставить стиль целиком в виде строки.

Например:

```
<div>Button</div>

<script>
  var div = document.body.children[0];

  div.style.cssText="color: red !important; \
    background-color: yellow; \
    width: 100px; \
    text-align: center; \
    blabla: 5; \
  ";

  alert(div.style.cssText);
</script>
```

Браузер разбирает строку `style.cssText` и применяет известные ему свойства. Неизвестные, наподобие `blabla`, большинство браузеров просто проигнорируют.

При установке `style.cssText` все предыдущие свойства `style` удаляются.

Итак, `style.cssText` осуществляет полную перезапись `style`. Если же нужно заменить какое-то конкретно свойство стиля, то обращаются именно к нему: `style.color`, `style.width` и т.п., чтобы не затереть что-то важное по ошибке.

Свойство `style.cssText` используют, например, для новосозданных элементов, когда старых стилей точно нет.

Чтение стиля из `style`

Записать в стиль очень просто. А как прочитать?

Например, мы хотим узнать размер, отступы элемента, его цвет... Как это сделать?

Свойство `style` содержит лишь тот стиль, который указан в атрибуте элемента, без учёта каскада CSS.

Вот так `style` уже ничего не увидит:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  Красный текст
  <script>
    alert(document.body.style.color); //в большинстве браузеров
    alert(document.body.style.marginTop); // ничего не выведет
  </script>
</body>
```

Полный стиль из `getComputedStyle`

Итак, свойство `style` дает доступ только к той информации, которая хранится в `elem.style`.

Он не скажет ничего об отступе, если он появился в результате наложения CSS или встроенных стилей браузера:

А если мы хотим, например, сделать анимацию и плавно увеличивать `marginTop` от текущего значения? Как нам сделать это? Ведь для начала нам надо это текущее значение получить.

Для того, чтобы получить текущее используемое значение свойства, используется метод `window.getComputedStyle`, описанный в стандарте DOM Level 2 ↗.

Его синтаксис таков:

```
getComputedStyle(element[, pseudo])
```

element

Элемент, значения для которого нужно получить

pseudo

Указывается, если нужен стиль псевдо-элемента, например "`::before`". Пустая строка или отсутствие аргумента означают сам элемент.

Поддерживается всеми браузерами, кроме IE8-. Следующий код будет работать во всех не-IE браузерах и в IE9+:

```
<style>
  body {
    margin: 10px
  }
</style>

<body>
  <script>
    var computedStyle = getComputedStyle(document.body);
    alert( computedStyle.marginTop ); // выведет отступ в пикселях
    alert( computedStyle.color ); // выведет цвет
  </script>
</body>
```

Вычисленное (computed) и окончательное (resolved) значения

В CSS есть две концепции:

1. *Вычисленное (computed)* значение — это то, которое получено после применения всех правил CSS и CSS-наследования. Например, `width: auto` или `font-size: 125%`.
2. *Окончательное (resolved)* значение — непосредственно применяемое к элементу. При этом все размеры приводятся к пикселям, например `width: 212px` или `font-size: 16px`. В некоторых браузерах пиксели могут быть дробными.

Когда-то `getComputedStyle` задумывалось для возврата вычисленного значения, но со временем оказалось, что окончательное гораздо удобнее.

Поэтому сейчас в целом все значения возвращаются именно окончательные, кроме некоторых небольших глюков в браузерах, которые постепенно вычищаются.

getComputedStyle требует полное свойство!

Для правильного получения значения нужно указать точное свойство. Например: paddingLeft, marginTop, borderLeftWidth.

При обращении к сокращенному: padding, margin, border — правильный результат не гарантируется.

Действительно, допустим свойства paddingLeft/paddingTop взяты из разных классов CSS. Браузер не обязан объединять их в одно свойство padding. Иногда, в простейших случаях, когда свойство задано сразу целиком, getComputedStyle сработает для сокращённого свойства, но не во всех браузерах.

Например, некоторые браузеры (Chrome) выведут 10px в документе ниже, а некоторые (Firefox) — нет:

```
<style>
  body {
    margin: 10px;
  }
</style>
<script>
  var style = getComputedStyle(document.body);
  alert( style.margin ); // в Firefox пустая строка
</script>
```

Стили посещенных ссылок — тайна!

У посещенных ссылок может быть другой цвет, фон, чем у обычных. Это можно поставить в CSS с помощью псевдокласса :visited.

Но getComputedStyle не дает доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку.

Кроме того, большинство браузеров запрещают применять к :visited CSS-стили, которые могут изменить геометрию элемента, чтобы даже окольным путем нельзя было это понять. В целях безопасности.

currentStyle для IE8-

В IE8- нет getComputedStyle, но у элементов есть свойство [currentStyle ↗](#), которое возвращает вычисленное (computed) значение: уже с учётом CSS-каскада, но не всегда в окончательном формате.

Чтобы код работал и в старых и новых браузерах, обычно пишут кросс-браузерный код, наподобие такого:

```
function getStyle(elem) {
  return window.getComputedStyle ? getComputedStyle(elem, "") : elem.currentStyle;
}
```

Если вы откроете такой документ в IE8-, то размеры будут в процентах, а в современных браузерах — в пикселях.

```
<style>
  body {
    margin: 10%
  }
</style>

<body>
  <script>
    var elem = document.body;

    function getStyle(elem) {
      return window.getComputedStyle ? getComputedStyle(elem, "") : elem.currentStyle;
    }

    var marginTop = getStyle(elem).marginTop;
    alert( marginTop ); // IE8-: 10%, иначе пиксели
  </script>
</body>
```

i IE8-: перевод pt, em, % из currentStyle в пиксели

Эта информация — дополнительная, она не обязательна для освоения.

В IE для того, чтобы получить из процентов реальное значение в пикселях существует метод «`runtimeStyle+pixel`», [описанный Дином Эдвардсом](#).

Он основан на свойствах `runtimeStyle` и `pixelLeft`, работающих только в IE.

В [песочнице](#) вы можете найти функцию `getIEComputedStyle(elem, prop)`, которая получает значение в пикселях для свойства `prop`, используя `elem.currentStyle` и метод Дина Эдвардса, и пример её применения.

Если вам интересно, как он работает, ознакомьтесь со свойствами с [runtimeStyle](#) и [pixelLeft](#) в MSDN и раскройте код.

Конечно, это актуально только для IE8- и полифиллов.

Итого

Все DOM-элементы предоставляют следующие свойства.

- Свойство `style` — это объект, в котором CSS-свойства пишутся вот так вот. Чтение и изменение его свойств — это, по сути, работа с компонентами атрибута `style`.
- `style.cssText` — строка стилей для чтения или записи. Аналог полного атрибута `style`.

- Свойство `currentStyle`(IE8-) и метод `getComputedStyle` (IE9+, стандарт) позволяют получить реальное, применённое сейчас к элементу свойство стиля с учётом CSS-каскада и браузерных стилей по умолчанию.

При этом `currentStyle` возвращает значение из CSS, до окончательных вычислений, а `getComputedStyle` — окончательное, непосредственно применённое к элементу (как правило).

Более полная информация о свойстве `style`, включающая другие, реже используемые методы работы с ним, доступна в [документации](#).

✓ Задачи

Скругленая кнопка со стилями из JavaScript

важность: 3

Создайте кнопку в виде элемента `<a>` с заданным стилем, используя JavaScript.

В примере ниже такая кнопка создана при помощи HTML/CSS. В вашем решении кнопка должна создаваться, настраиваться и добавляться в документ при помощи *только* `JavaScript`, без тегов `<style>` и `<a>`.

```
<style>
.button {
    -moz-border-radius: 8px;
    -webkit-border-radius: 8px;
    border-radius: 8px;
    border: 2px groove green;
    display: block;
    height: 30px;
    line-height: 30px;
    width: 100px;
    text-decoration: none;
    text-align: center;
    color: red;
    font-weight: bold;
}
</style>

<a class="button" href="/">Нажми меня</a>
```

Нажми меня

Проверьте себя: вспомните, что означает каждое свойство. В чём состоит эффект его появления здесь?

[Открыть песочницу для задачи.](#)

[К решению](#)

Создать уведомление

важность: 5

Напишите функцию `showNotification(options)`, которая показывает уведомление, пропадающее через 1.5 сек.

Описание функции:

```
/**  
 * Показывает уведомление, пропадающее через 1.5 сек  
 *  
 * @param options.top {number} вертикальный отступ, в px  
 * @param options.right {number} правый отступ, в px  
 * @param options.cssText {string} строка стиля  
 * @param options.className {string} CSS-класс  
 * @param options.html {string} HTML-текст для показа  
 */  
function showNotification(options) {  
    // ваш код  
}
```

Пример использования:

```
// покажет элемент с текстом "Привет" и классом welcome справа-сверху окна  
showNotification({  
    top: 10,  
    right: 10,  
    html: "Привет",  
    className: "welcome"  
});
```

[Демо в новом окне ↗](#)

Элемент уведомления должен иметь CSS-класс `notification`, к которому добавляется класс из `options.className`, если есть. Исходный документ содержит готовые стили.

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Размеры и прокрутка элементов

Для того, чтобы показывать элементы на произвольных местах страницы, необходимо во-первых, знать CSS-позиционирование, а во-вторых — уметь работать с «геометрией элементов» из JavaScript.

В этой главе мы поговорим о размерах элементов DOM, способах их вычисления и *метриках* — различных свойствах, которые содержат эту информацию.

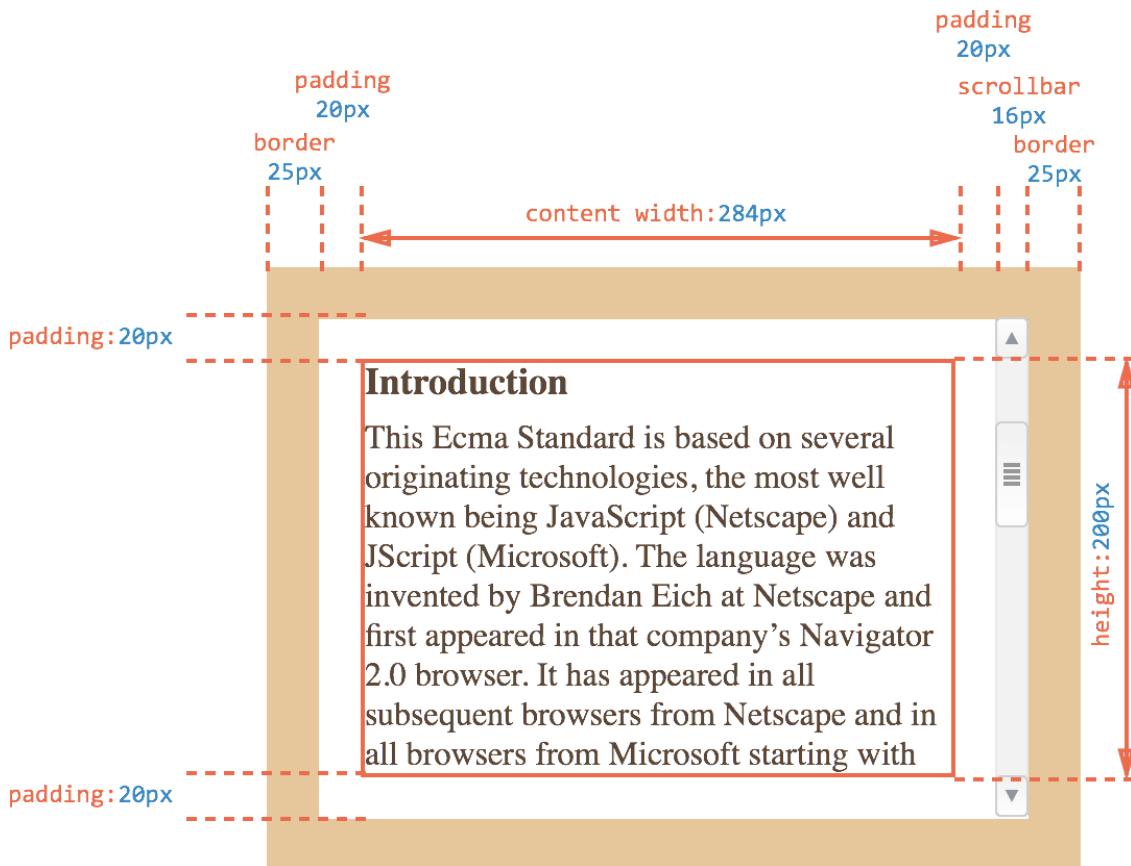
Образец документа

Мы будем использовать для примера вот такой элемент, у которого есть рамка (border), поля (padding), и прокрутка:

```
<div id="example">
  ...Текст...
</div>
<style>
  #example {
    width: 300px;
    height: 200px;
    border: 25px solid #E8C48F; /* рамка 25px */
    padding: 20px;           /* поля 20px */
    overflow: auto;         /* прокрутка */
  }
</style>
```

У него нет отступов `margin`, в этой главе они не важны, так как метрики касаются именно размеров самого элемента, отступы в них не учитываются.

Результат выглядит так:



Вы можете открыть [этот документ](#) в песочнице ↗.

Внимание, полоса прокрутки!

В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки.

В этом случае полоса прокрутки «отодвигает» содержимое вместе с `padding` влево, отбирая у него место.

Именно поэтому ширина содержимого обозначена как `content width` и равна 284px, а не 300px, как в CSS.

Точное значение получено в предположении, что ширина полосы прокрутки равна 16px, то есть после её вычитания из содержимого остаётся $300 - 16 = 284$ px. Конечно, она сильно зависит от браузера, устройства, ОС.

Мы должны в точности понимать, что происходит с размерами элемента при наличии полосы прокрутки, поэтому на картинке выше это отражено.

Поле padding заполнено текстом

Обычно поля `padding` изображают пустыми, но так как текста много, то он заполняет нижнее поле `padding-bottom` в примере выше.

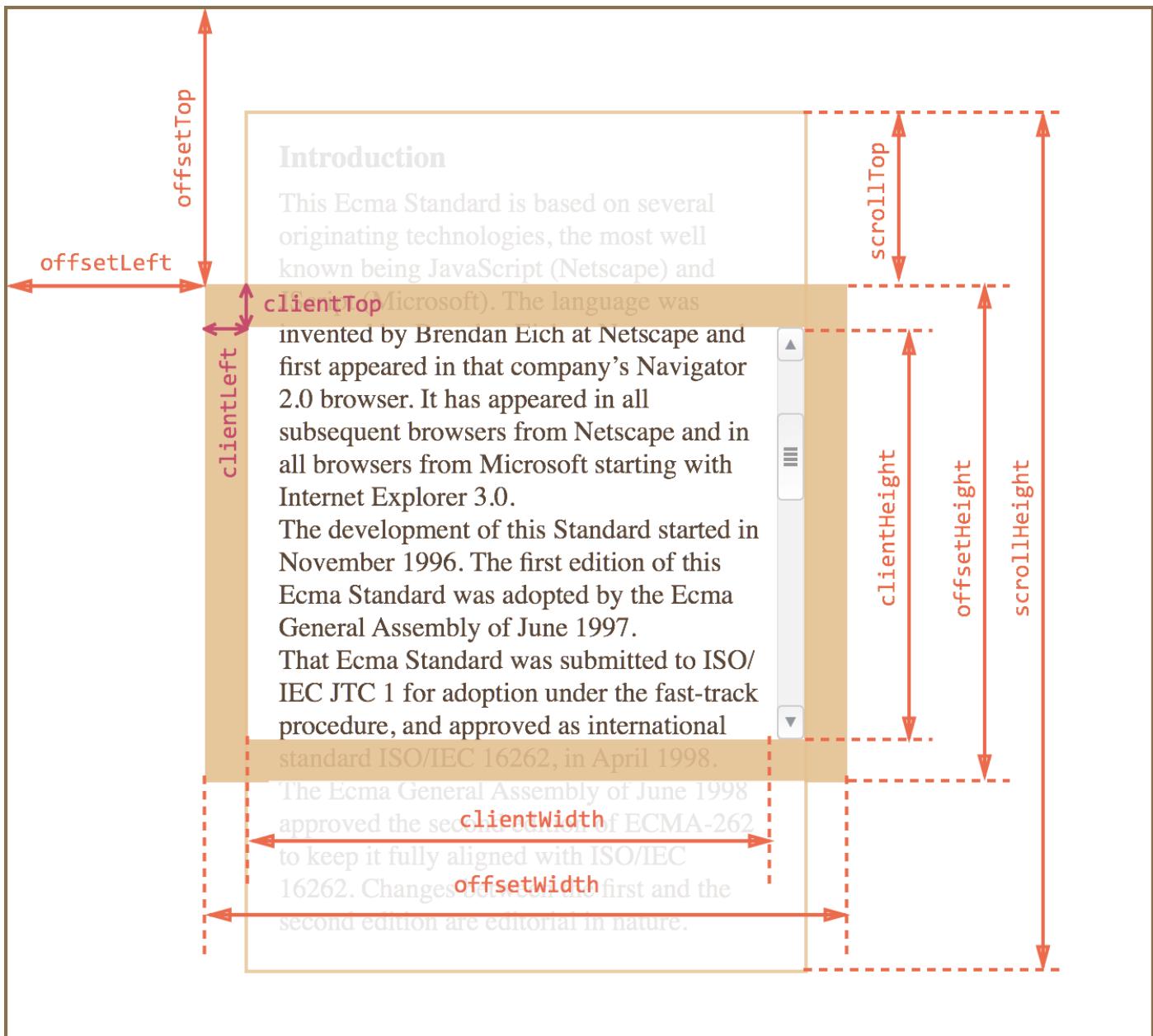
Во избежание путаницы заметим, что `padding` там, всё же, есть. Поля `padding` по CSS в элементе выше одинаковы со всех сторон. А такое заполнение — нормальное поведение браузера.

Метрики

У элементов существует ряд свойств, содержащих их внешние и внутренние размеры. Мы будем называть их «метриками».

Метрики, в отличие от свойств CSS, содержат числа, всегда в пикселях и без единиц измерения на конце.

Вот общая картина:



На картинке все они с трудом помещаются, но, как мы увидим далее, их значения просты и понятны.

Будем исследовать их снаружи элемента и вовнутрь.

offsetParent, offsetLeft/Top

Ситуации, когда эти свойства нужны, можно перечислить по пальцам. Они возникают действительно редко. Как правило, эти свойства используют, потому что не знают средств правильной работы с координатами, о которых мы поговорим позже.

Несмотря на то, что эти свойства нужны реже всего, они — самые «внешние», поэтому начнём с них.

В offsetParent находится ссылка на родительский элемент в смысле отображения на странице.

Уточним, что имеется в виду.

Когда браузер рисует страницу, то он высчитывает дерево расположения элементов, иначе говоря «дерево геометрии» или «дерево рендеринга», которое содержит всю информацию о размерах.

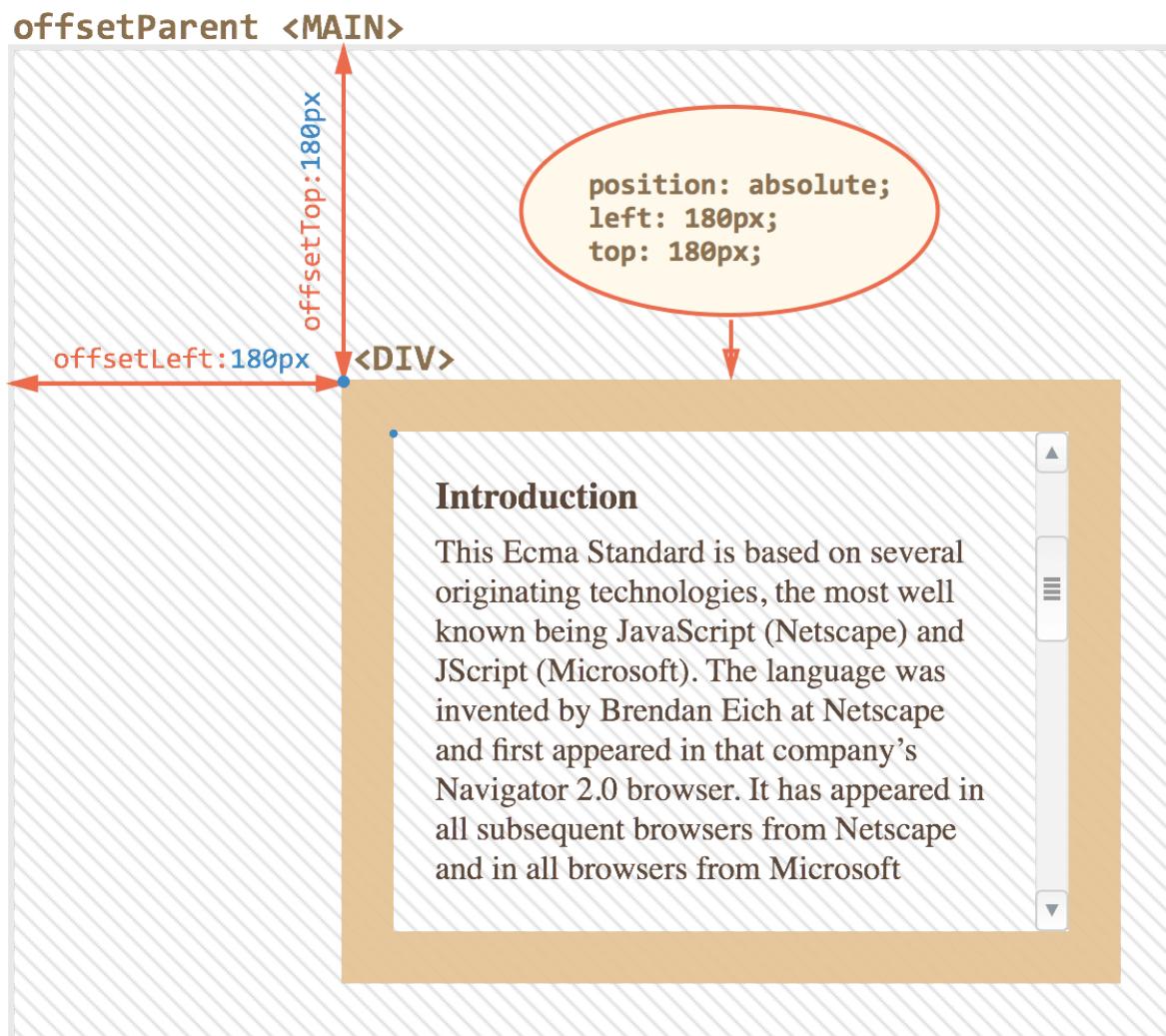
При этом одни элементы естественным образом рисуются внутри других. Но, к примеру, если у элемента стоит `position: absolute`, то его расположение вычисляется уже не относительно непосредственного родителя `parentNode`, а относительно ближайшего [позиционированного элемента ↗](#) (т.е. свойство `position` которого не равно `static`), или `BODY`, если такой отсутствует.

Получается, что элемент имеет в дополнение к обычному родителю в DOM — ещё одного «родителя по позиционированию», то есть относительно которого он рисуется. Этот элемент и будет в свойстве `offsetParent`.

Свойства `offsetLeft/Top` задают смещение относительно `offsetParent`.

В примере ниже внутренний `<div>` имеет DOM-родителя `<form>`, но `offsetParent` у него `<main>`, и сдвиги относительно его верхнего-левого угла будут в `offsetLeft/Top`:

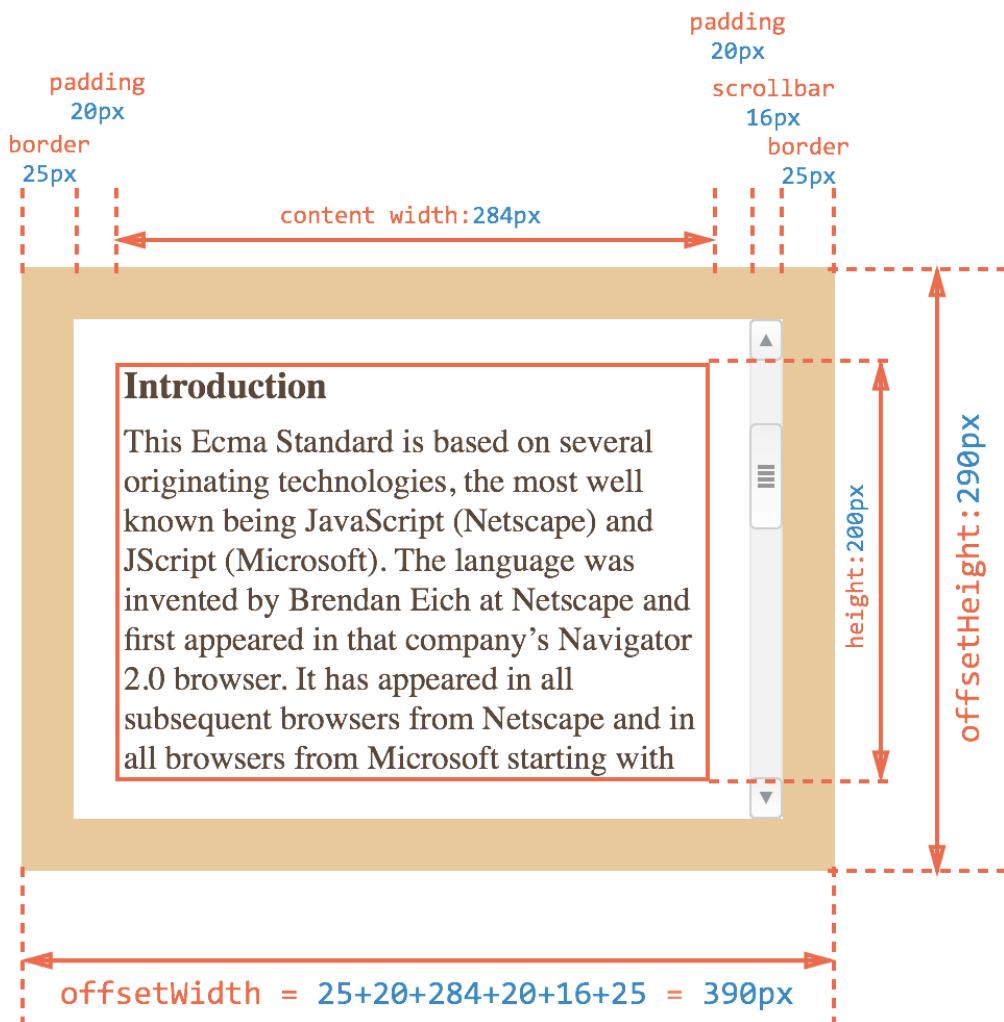
```
<main style="position: relative">
  <form>
    <div id="example" style="position: absolute; left: 180px; top: 180px">...</div>
  </form>
</main>
```



offsetWidth/Height

Теперь переходим к самому элементу.

Эти два свойства — самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки border.



Для нашего элемента:

- $\text{offsetWidth} = 390$ — внешняя ширина блока, её можно получить сложением CSS-ширины (300px, но её часть на рисунке выше отнимает прокрутка, поэтому 284 + 16), полей(2*20px) и рамок (2*25px).
- $\text{offsetHeight} = 290$ — внешняя высота блока.

Метрики для невидимых элементов равны нулю.

Координаты и размеры в JavaScript устанавливаются только для видимых элементов.

Для элементов с `display:none` или находящихся вне документа дерево рендеринга не строится. Для них метрики равны нулю. Кстати, и `offsetParent` для таких элементов тоже `null`.

Это дает нам замечательный способ для проверки, виден ли элемент:

```
function isHidden(elem) {  
    return !elem.offsetWidth && !elem.offsetHeight;  
}
```

- Работает, даже если родителю элемента установлено свойство `display:none`.
- Работает для всех элементов, кроме TR, с которым возникают некоторые проблемы в разных браузерах. Обычно, проверяются не TR, поэтому всё ок.
- Считает элемент видимым, даже если позиционирован за пределами экрана или имеет свойство `visibility:hidden`.
- «Схлопнутый» элемент, например пустой `div` без высоты и ширины, будет считаться невидимым.

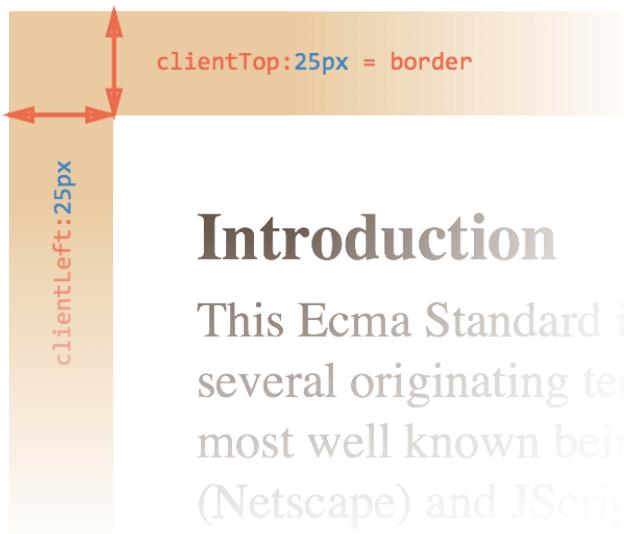
clientTop/Left

Далее внутри элемента у нас рамки `border`.

Для них есть свойства-метрики `clientTop` и `clientLeft`.

В нашем примере:

- `clientLeft = 25` — ширина левой рамки
- `clientTop = 25` — ширина верхней рамки



Introduction

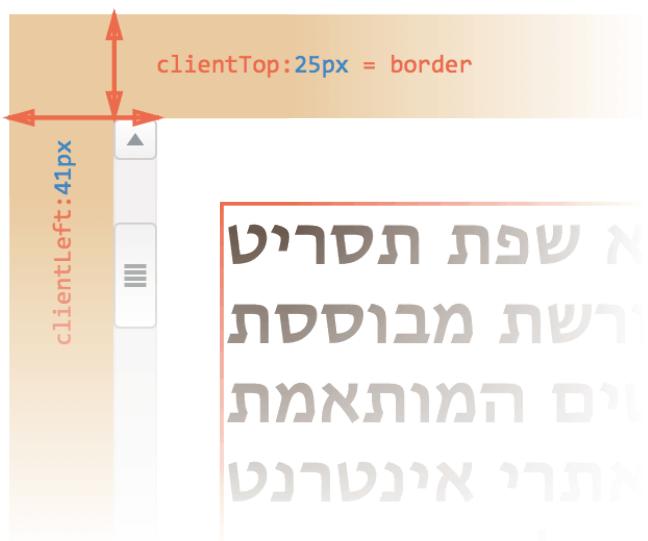
This Ecma Standard is several originating tec most well known bei (Netscape) and JScript

...Но на самом деле они — вовсе не рамки, а отступ внутренней части элемента от внешней.

В чём же разница?

Она возникает тогда, когда документ располагается *справа налево* (операционная система на арабском языке или иврите). Полоса прокрутки в этом случае находится слева, и тогда свойство `clientLeft` включает в себя еще и ширину полосы прокрутки.

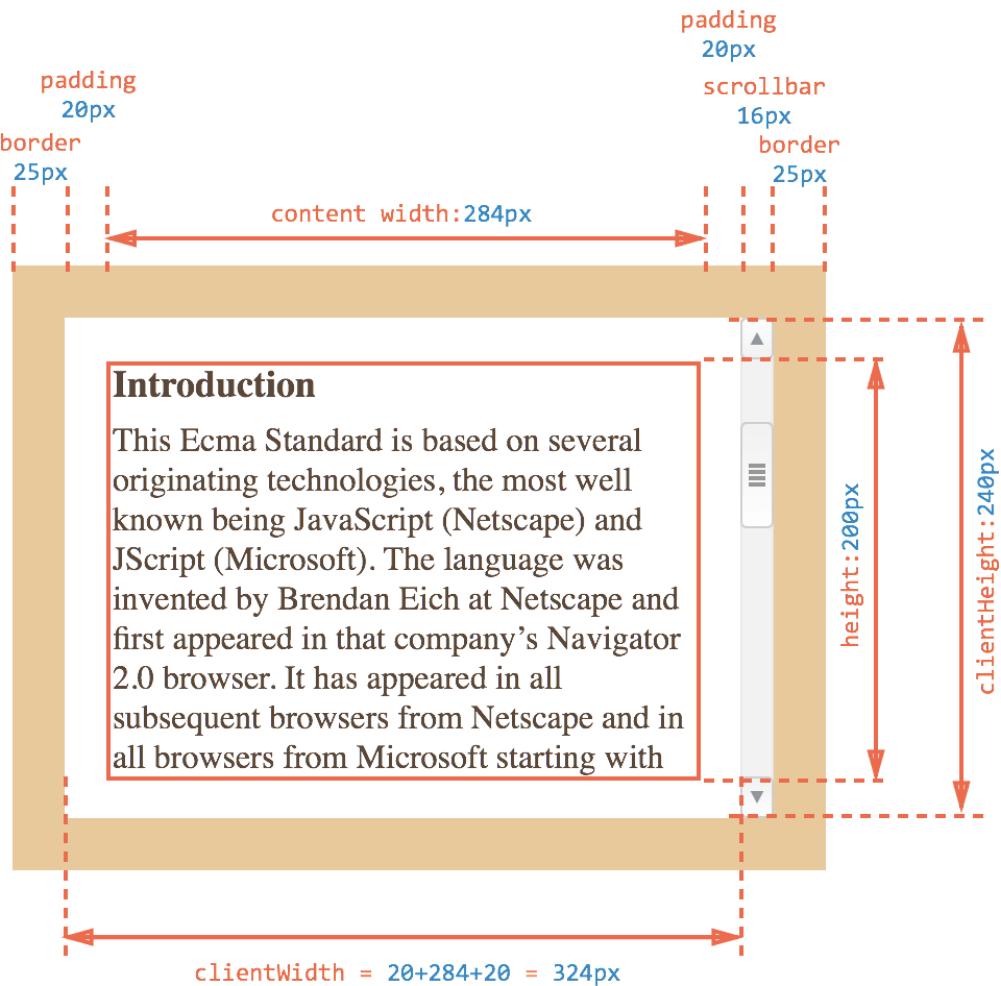
Получится так:



clientWidth/Height

Эти свойства — размер элемента внутри рамок `border`.

Они включают в себя ширину содержимого `width` вместе с полями `padding`, но без прокрутки.

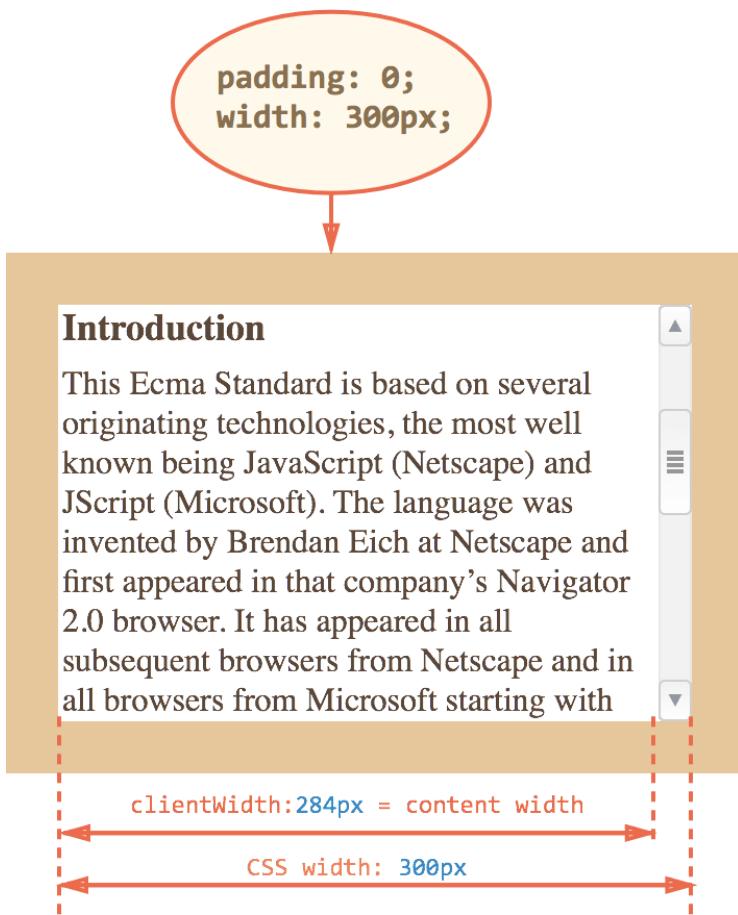


На рисунке выше посмотрим вначале на `clientHeight`, её посчитать проще всего. Прокрутки нет, так что это в точности то, что внутри рамок: CSS-ширина 200px плюс верхнее и нижнее поля padding (по 20px), итого 240px.

На рисунке нижний padding заполнен текстом, но это неважно: по правилам он всегда входит в `clientHeight`.

Теперь `clientWidth` — ширина содержимого здесь не равна CSS-ширине, её часть «съедает» полоса прокрутки. Поэтому в `clientWidth` входит не CSS-ширина, а реальная ширина содержимого 284px плюс левое и правое поля padding (по 20px), итого 324px.

Если padding нет, то clientWidth/Height в точности равны размеру области содержимого, внутри рамок и полосы прокрутки.



Поэтому в тех случаях, когда мы точно знаем, что padding нет, их используют для определения внутренних размеров элемента.

scrollWidth/Height

Эти свойства — аналоги clientWidth/clientHeight, но с учетом прокрутки.

Свойства clientWidth/clientHeight относятся только к видимой области элемента, а scrollWidth/scrollHeight добавляют к ней прокрученную (которую не видно) по горизонтали/вертикали.

Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0. The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997. That Ecma Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.



На рисунке выше:

- `scrollHeight` = 723 — полная внутренняя высота, включая прокрученную область.
- `scrollWidth` = 324 — полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна `clientWidth`.

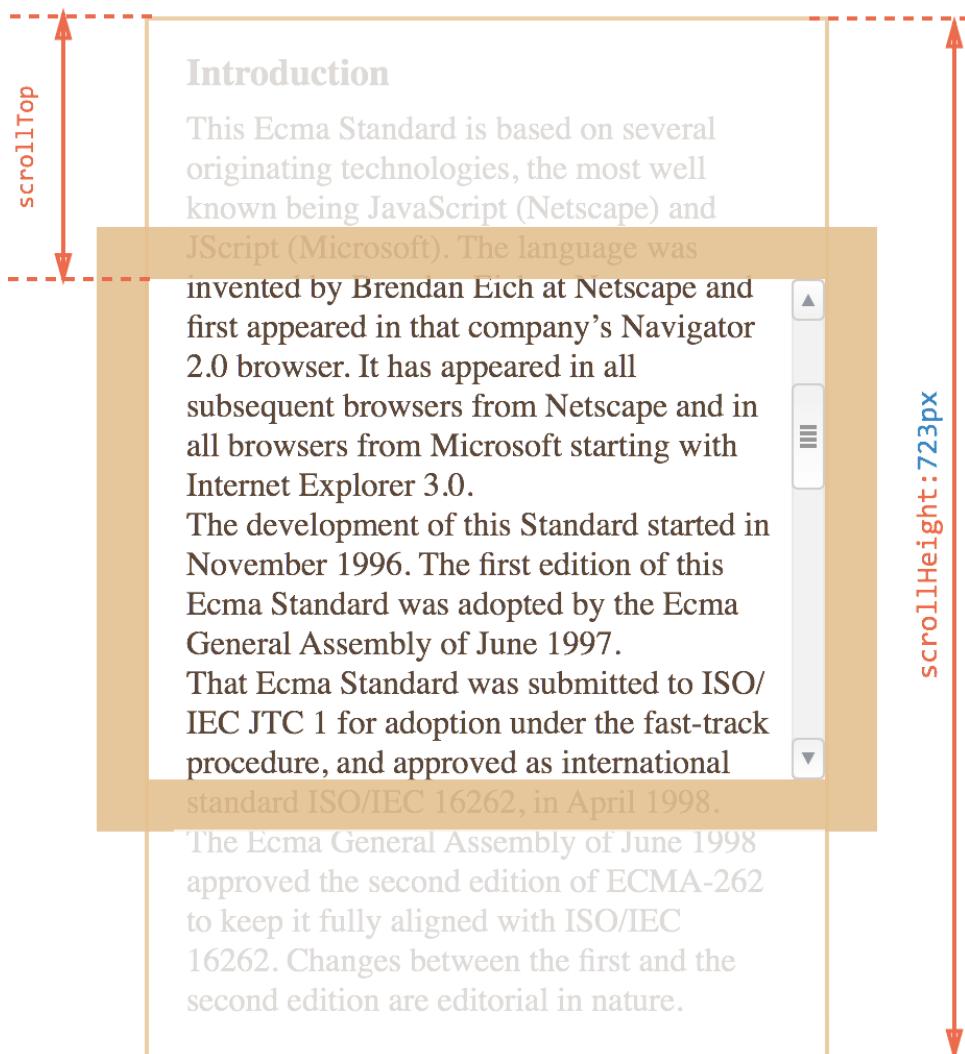
Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту, таким кодом:

```
element.style.height = element.scrollHeight + 'px';
```

scrollLeft/scrollTop

Свойства `scrollLeft`/`scrollTop` — ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Следующее иллюстрация показывает значения `scrollHeight` и `scrollTop` для блока с вертикальной прокруткой.



i scrollLeft(scrollTop) можно изменять

В отличие от большинства свойств, которые доступны только для чтения, значения scrollLeft/scrollTop можно изменить, и браузер выполнит прокрутку элемента.

Не стоит брать width/height из CSS

Мы рассмотрели метрики — свойства, которые есть у DOM-элементов. Их обычно используют для получения их различных высот, ширин и прочих расстояний.

Теперь несколько слов о том, как *не* надо делать.

Как мы знаем, CSS-высоту и ширину можно установить с помощью elem.style и извлечь, используя getComputedStyle, которые в подробностях обсуждаются в главе [Стили, getComputedStyle](#).

Получение ширины элемента может быть таким:

```
var elem = document.body;  
  
alert( getComputedStyle(elem).width ); // вывести CSS-ширину для elem
```

Не лучше ли получать ширину так, вместо метрик? Вовсе нет!

1. Во-первых, CSS-свойства `width/height` зависят от другого свойства — `box-sizing`, которое определяет, что такое, собственно, эти ширина и высота. Получается, что изменение `box-sizing`, к примеру, для более удобной вёрстки, сломает такой JavaScript.
2. Во-вторых, в CSS свойства `width/height` могут быть равны `auto`, например, для инлайн-элемента:

```
<span id="elem">Привет!</span>

<script>
  alert( getComputedStyle(elem).width ); // auto
</script>
```

Конечно, с точки зрения CSS размер `auto` — совершенно нормально, но нам-то в JavaScript нужен конкретный размер в пикселях, который мы могли бы использовать для вычислений. Получается, что в данном случае ширина `width` из CSS вообще бесполезна.

Есть и ещё одна причина.

Полоса прокрутки — причина многих проблем и недопониманий. Как говорится, «дьявол кроется в деталях». Недопустимо, чтобы наш код работал на элементах без прокрутки и начинал «глючить» с ней.

Как мы говорили ранее, при наличии вертикальной полосы прокрутки, в зависимости от браузера, устройства и операционной системы, она может сдвинуть содержимое.

Получается, что реальная ширина содержимого меньше CSS-ширины. И это учитывают свойства `clientWidth/clientHeight`.

...Но при этом некоторые браузеры также учитывают это в результате `getComputedStyle(elem).width`, то есть возвращают реальную внутреннюю ширину, а некоторые — именно CSS-свойство. Эти кросс-браузерные отличия — ещё один повод не использовать такой подход, а использовать свойства-метрики.

Описанные разнотечения касаются только чтения свойства `getComputedStyle(...).width` из JavaScript, визуальное отображение корректно в обоих случаях.

Итого

У элементов есть следующие метрики:

- `offsetParent` — «родитель по дереву рендеринга» — ближайшая ячейка таблицы, `body` для статического позиционирования или ближайший позиционированный элемент для других типов позиционирования.
- `offsetLeft/offsetTop` — позиция в пикселях левого верхнего угла блока, относительно его `offsetParent`.
- `offsetWidth/offsetHeight` — «внешняя» ширина/высота блока, включая рамки.
- `clientLeft/clientTop` — отступ области содержимого от левого-верхнего угла элемента. Если

операционная система располагает вертикальную прокрутку справа, то равны ширинам левой/верхней рамки, если же слева (ОС на иврите, арабском), то `clientLeft` включает в себя прокрутку.

- `clientWidth/clientHeight` — ширина/высота содержимого вместе с полями `padding`, но без полосы прокрутки.
- `scrollWidth/scrollHeight` — ширина/высота содержимого, включая прокручиваемую область. Включает в себя `padding` и не включает полосы прокрутки.
- `scrollLeft/scrollTop` — ширина/высота прокрученной части документа, считается от верхнего левого угла.

Все свойства, доступны только для чтения, кроме `scrollLeft/scrollTop`. Изменение этих свойств заставляет браузер прокручивать элемент.

В этой главе мы считали, что страница находится в режиме соответствия стандартам. В режиме совместимости — некоторые старые браузеры требуют `document.body` вместо `documentElement`, в остальном всё так же. Конечно, по возможности, стоит использовать только режим соответствия стандарту.

✓ Задачи

Найти размер прокрутки снизу

важность: 5

Свойство `elem.scrollTop` содержит размер прокрученной области при отсчете сверху. А как подсчитать размер прокрутки снизу?

Напишите соответствующее выражение для произвольного элемента `elem`.

Проверьте: если прокрутки нет вообще или элемент полностью прокручен — оно должно давать ноль.

[К решению](#)

Узнать ширину полосы прокрутки

важность: 3

Напишите код, который возвращает ширину стандартной полосы прокрутки. Именно самой полосы, где ползунок. Обычно она равна 16px, в редких и мобильных браузерах может колебаться от 14px до 18px, а кое-где даже равна 0px.

P.S. Ваш код должен работать на любом HTML-документе, независимо от его содержимого.

[К решению](#)

Подменить div на другой с таким же размером

важность: 3

Посмотрим следующий случай из жизни. Был текст, который, в частности, содержал div с зелеными границами:

```
<style>
  #moving-div {
    border: 5px groove green;
    padding: 5px;
    margin: 10px;
    background-color: yellow;
  }
</style>
```

Before Before Before

```
<div id="moving-div">
  Text Text Text<br>
  Text Text Text<br>
</div>
```

After After After

Программист Валера из вашей команды написал код, который позиционирует его абсолютно и смещает в правый верхний угол. Вот этот код:

```
var div = document.getElementById('moving-div');
div.style.position = 'absolute';
div.style.right = div.style.top = 0;
```

Побочным результатом явилось смещение текста, который раньше шел после DIV. Теперь он поднялся вверх:

Before Before Before After After

Text Text Text
Text Text Text

Допишите код Валеры, сделав так, чтобы текст оставался на своем месте после того, как DIV будет смещен.

Сделайте это путем создания вспомогательного DIV с теми же width, height, border, margin, padding, что и у желтого DIV.

Используйте только JavaScript, без CSS.

Должно быть так (новому блоку задан фоновый цвет для демонстрации):

Before Before Before

Text Text Text
Text Text Text

After After After

[Открыть песочницу для задачи.](#) ↗

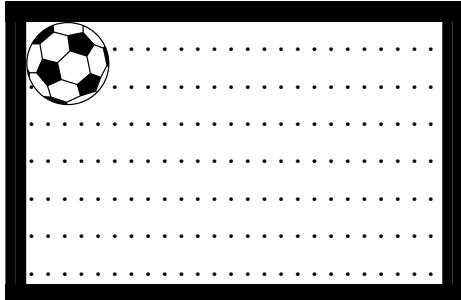
[К решению](#)

Поместите мяч в центр поля

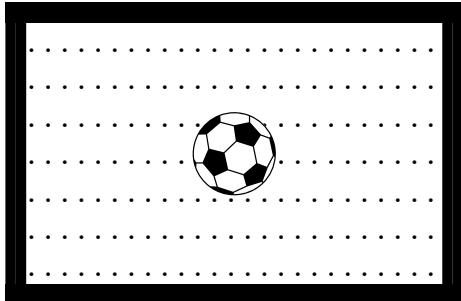
важность: 5

Поместите мяч в центр поля.

Исходный документ выглядит так:



Используйте JavaScript, чтобы поместить мяч в центр:



- Менять CSS нельзя, мяч должен переносить в центр ваш скрипт, через установку нужных стилей элемента.
- JavaScript-код должен работать при разных размерах мяча (10, 20, 30 пикселей) без изменений.
- JavaScript-код должен работать при различных размерах и местоположениях поля на странице без изменений. Также он не должен зависеть от ширины рамки поля border.

P.S. Да, центрирование можно сделать при помощи чистого CSS, но задача именно на JavaScript. Далее будут другие темы и более сложные ситуации, когда JavaScript будет уже точно необходим, это — своего рода «разминка».

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Расширить элемент

важность: 4

В `<body>` есть элемент `<div>` с заданной шириной `width`.

Задача — написать код, который «распахнет» `<div>` по ширине на всю страницу.

Исходный документ (`<div>` содержит текст и прокрутку):

```
текст текст текст текст  
текст текст текст текст
```

P.S. Пользоваться следует исключительно средствами JS, CSS в этой задаче менять нельзя. Также ваш код должен быть универсален и не ломаться, если цифры в CSS станут другими.

P.P.S. При расширении элемент `<div>` не должен вылезти за границу `<body>`.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

В чём отличие «`width`» и «`clientWidth`» ?

важность: 5

В чём отличия между `getComputedStyle(elem).width` и `elem.clientWidth`?

Укажите хотя бы три отличия, лучше — больше.

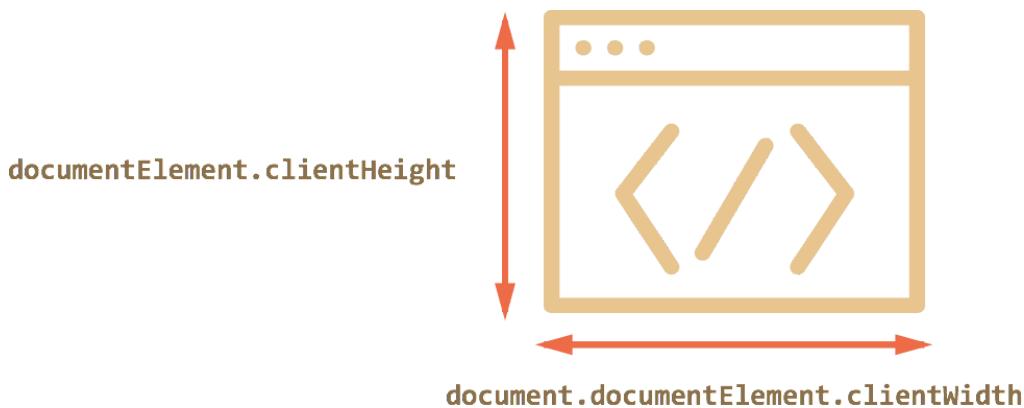
Размеры и прокрутка страницы

Как найти ширину окна браузера? Как узнать всю высоту страницы, с учётом прокрутки? Как прокрутить её из JavaScript?

С точки зрения HTML, документ — это `document.documentElement`. У этого элемента, соответствующего тегу `<html>`, есть все стандартные свойства и метрики и, в теории, они и должны нам помочь. Однако, на практике есть ряд нюансов, именно их мы рассмотрим в этой главе.

Ширина/высота видимой части окна

Свойства `clientWidth/Height` для элемента `document.documentElement` — это как раз ширина/высота видимой области окна.



⚠ Не `window.innerWidth/Height`

Все браузеры, кроме IE8-, также поддерживают свойства `window.innerWidth/innerHeight`. Они хранят текущий размер окна браузера.

В чём отличие? Оно небольшое, но чрезвычайно важное.

Свойства `clientWidth/Height`, если есть полоса прокрутки, возвращают именно ширину/высоту документа, за вычетом прокрутки, а эти свойства — игнорируют её наличие.

Если справа часть страницы занимает полоса прокрутки, то эти строки выведут разное:

```
alert( window.innerWidth ); // вся ширина окна  
alert( document.documentElement.clientWidth ); // ширина минус прокрутка
```

Обычно нам нужна именно *доступная* ширина окна, например, чтобы нарисовать что-либо, то есть за вычетом полосы прокрутки. Поэтому используем `documentElement.clientWidth`.

Ширина/высота страницы с учётом прокрутки

Теоретически, видимая часть страницы — это `documentElement.clientWidth/Height`, а полный размер с учётом прокрутки — по аналогии, `documentElement.scrollWidth/scrollHeight`.

Это верно для обычных элементов.

А вот для страницы с этими свойствами возникает проблема, когда *прокрутка то есть, то нет*. В этом случае они работают некорректно. В браузерах Chrome/Safari и Opera при отсутствии прокрутки значение `documentElement.scrollHeight` в этом случае может быть даже меньше, чем `documentElement.clientHeight`, что, конечно же, выглядит как совершеннейшая чепуха и нонсенс.

Эта проблема возникает именно для `documentElement`, то есть для всей страницы.

Надёжно определить размер страницы с учетом прокрутки можно, взяв максимум из нескольких свойств:

```
var scrollHeight = Math.max(  
    document.body.scrollHeight, document.documentElement.scrollHeight,  
    document.body.offsetHeight, document.documentElement.offsetHeight,  
    document.body.clientHeight, document.documentElement.clientHeight  
);  
  
alert( 'Высота с учетом прокрутки: ' + scrollHeight );
```

Почему так? Лучше и не спрашивайте, это одно из редких мест, где просто ошибки в браузерах. Глубокой логики здесь нет.

Получение текущей прокрутки

У обычного элемента текущую прокрутку можно получить в `scrollLeft/scrollTop`.

Что же со страницей?

Большинство браузеров корректно обработает запрос к `documentElement.scrollLeft/Top`, однако Safari/Chrome/Opera есть ошибки (к примеру [157855 ↗](#), [106133 ↗](#)), из-за которых следует использовать `document.body`.

Чтобы вообще обойти проблему, можно использовать специальные свойства `window.pageYOffset/pageXOffset`:

```
alert( 'Текущая прокрутка сверху: ' + window.pageYOffset );  
alert( 'Текущая прокрутка слева: ' + window.pageXOffset );
```

Эти свойства:

- Не поддерживаются IE8-
- Их можно только читать, а менять нельзя.

Если IE8- не волнует, то просто используем эти свойства.

Кросс-браузерный вариант с учётом IE8 предусматривает откат на `documentElement`:

```
var scrollTop = window.pageYOffset || document.documentElement.scrollTop;  
alert( "Текущая прокрутка: " + scrollTop );
```

Изменение прокрутки: `scrollTo`, `scrollBy`, `scrollIntoView`

Важно:

Чтобы прокрутить страницу при помощи JavaScript, её DOM должен быть полностью загружен.

На обычных элементах свойства `scrollTop`/`scrollLeft` можно изменять, и при этом элемент будет прокручиваться.

Никто не мешает точно так же поступать и со страницей. Во всех браузерах, кроме Chrome/Safari/Opera можно осуществить прокрутку установкой `document.documentElement.scrollTop`, а в указанных — использовать для этого `document.body.scrollTop`. И будет работать. Можно попробовать прокручивать и так и эдак и проверять, подействовала ли прокрутка, будет кросс-браузерно.

Но есть и другое, простое и универсальное решение — специальные методы прокрутки страницы `window.scrollBy(x,y)` ↗ и `window.scrollTo(pageX,pageY)` ↗.

- Метод `scrollBy(x,y)` прокручивает страницу относительно текущих координат.
- Метод `scrollTo(pageX,pageY)` прокручивает страницу к указанным координатам относительно документа.

Он эквивалентен установке свойств `scrollLeft`/`scrollTop`.

Чтобы прокрутить в начало документа, достаточно указать координаты `(0,0)`.

`scrollIntoView`

Для полноты картины рассмотрим также метод `elem.scrollIntoView(top)` ↗.

Метод `elem.scrollIntoView(top)` вызывается на элементе и прокручивает страницу так, чтобы элемент оказался вверху, если параметр `top` равен `true`, и внизу, если `top` равен `false`. Причем, если параметр `top` не указан, то он считается равным `true`.

Кнопка ниже прокрутит страницу так, чтобы кнопка оказалась вверху:

```
this.scrollIntoView()
```

А следующая кнопка прокрутит страницу так, чтобы кнопка оказалась внизу:

```
this.scrollIntoView(false)
```

Запрет прокрутки

Иногда бывает нужно временно сделать документ «непрокручиваемым». Например, при показе большого диалогового окна над документом — чтобы посетитель мог прокручивать это окно, но не документ.

Чтобы запретить прокрутку страницы, достаточно поставить `document.body.style.overflow = "hidden"`.

При этом страница замрёт в текущем положении.

Вместо `document.body` может быть любой элемент, прокрутку которого необходимо запретить.

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место.

Это может быть не очень красиво, но легко обходится, если вычислить размер прокрутки и добавить такой же по размеру `padding`.

Итого

Размеры:

- Для получения размеров видимой части окна: `document.documentElement.clientWidth/Height`
- Для получения размеров страницы с учётом прокрутки:

```
var scrollHeight = Math.max(  
    document.body.scrollHeight, document.documentElement.scrollHeight,  
    document.body.offsetHeight, document.documentElement.offsetHeight,  
    document.body.clientHeight, document.documentElement.clientHeight  
)
```

Прокрутка окна:

- Прокрутку окна можно получить как `window.pageYOffset` (для горизонтальной — `window.pageXOffset`) везде, кроме IE8-.

На всякий случай — вот самый кросс-браузерный способ, учитывающий IE7- в том числе:

```
var html = document.documentElement;  
var body = document.body;  
  
var scrollTop = html.scrollTop || body && body.scrollTop || 0;  
scrollTop -= html.clientTop; // в IE7- <html> смешён относительно (0,0)  
  
alert( "Текущая прокрутка: " + scrollTop );
```

- Установить прокрутку можно при помощи специальных методов:
 - `window.scrollTo(pageX,pageY)` — абсолютные координаты,
 - `window.scrollBy(x,y)` — прокрутить относительно текущего места.
 - `elem.scrollIntoView(true)` — прокрутить, чтобы элемент `elem` стал виден.

✓ Задачи

Полифилл для pageYOffset в IE8

важность: 3

Обычно в IE8 не поддерживается свойство `pageYOffset`. Напишите полифилл для него.

При подключённом полифилле такой код должен работать в IE8:

```
// текущая прокрутка страницы в IE8
alert( window.pageYOffset );
```

[К решению](#)

Координаты в окне

Для того, чтобы поместить один элемент рядом с другим на странице, а также двигать его произвольным образом, к примеру, рядом с указателем мыши — используются координаты.

Координатная система относительно окна браузера начинается в левом-верхнем углу текущей видимой области окна.

Мы будем называть координаты в ней `clientX/clientY`.

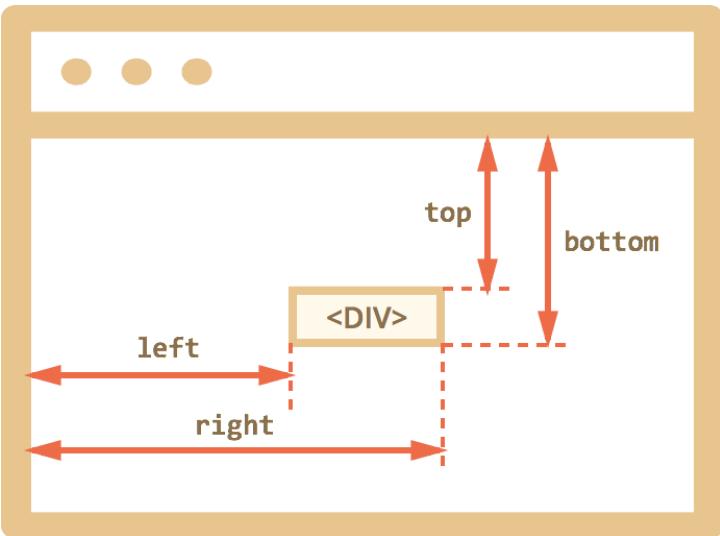
getBoundingClientRect()

Метод `elem.getBoundingClientRect()` возвращает координаты элемента, под которыми понимаются размеры «воображаемого прямоугольника», который охватывает весь элемент.

Координаты возвращаются в виде объекта со свойствами:

- `top` — Y-координата верхней границы элемента,
- `left` — X-координата левой границы,
- `right` — X-координата правой границы,
- `bottom` — Y-координата нижней границы.

Например:



Координаты относительно окна не учитывают прокрутку, они высчитываются от границ текущей видимой области.

Иначе говоря, если страницу прокрутить, то элемент поднимется выше или опустится ниже — его координаты относительно окна изменятся.

- Координаты могут быть дробными — это нормально, так как они возвращаются из внутренних структур браузера.
- Координаты могут быть и отрицательными, например если прокрутить страницу так, что верх элемента будет выходить за верхнюю границу окна, то его `top`-координата будет меньше нуля.
- Некоторые современные браузеры также добавляют к результату `getBoundingClientRect` свойства для ширины и высоты: `width/height`, но их можно получить и простым вычитанием: `height = bottom - top, width = right - left`.

⚠ Координаты `right/bottom` отличаются от CSS-свойств

Если рассмотреть позиционирование элементов при помощи CSS-свойства `position`, то там тоже указываются `left, right, top, bottom`.

Однако, по CSS свойство `right` задаёт расстояние от правой границы, а `bottom` — от нижней.

Если вы взглянете на иллюстрацию выше, то увидите, что в JavaScript это не так. Все координаты отсчитываются слева/сверху, в том числе и эти.

Метод elem.getBoundingClientRect() изнутри

Браузер отображает любое содержимое, используя прямоугольники.

В случае с блочным элементом, таким как DIV, элемент сам по себе образует прямоугольник. Но если элемент строчный и содержит в себе длинный текст, то каждая строка будет отдельным прямоугольником, с одинаковой высотой но разной длиной (у каждой строки — своя длина).

Более подробно это описано в: [спецификации ↗](#).

Если обобщить, содержимое элемента может отображаться в одном прямоугольнике или в нескольких.

Все эти прямоугольники можно получить с помощью [elem.getClientRects\(\) ↗](#). А метод [elem.getBoundingClientRect\(\) ↗](#) возвращает один охватывающий прямоугольник для всех `getClientRects()`.

elementFromPoint(x, y)

Возвращает элемент, который находится на координатах (x, y) относительно окна.

Синтаксис:

```
var elem = document.elementFromPoint(x, y);
```

Например, код ниже выделяет и выводит тег у элемента, который сейчас в середине окна:

```
var centerX = document.documentElement.clientWidth / 2;
var centerY = document.documentElement.clientHeight / 2;

var elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";
alert( elem.tagName );
elem.style.background = "";
```

Аналогично предыдущему методу, используются координаты относительно окна, так что, в зависимости от прокрутки страницы, в центре может быть разный элемент.

⚠ Для координат вне окна elementFromPoint возвращает null

Метод `document.elementFromPoint(x,y)` работает только если координаты `(x,y)` находятся в пределах окна.

Если одна из них отрицательна или больше чем ширина/высота окна — он возвращает `null`.

В большинстве сценариев использования это не является проблемой, но нужно проверять, что результат — не `null`.

Координаты для `position:fixed`

Координаты обычно требуются не просто так, а, например, чтобы переместить элемент на них.

В CSS для позиционирования элемента относительно окна используется свойство `position:fixed`. Как правило, вместе с ним идут и координаты, например `left/top`.

Например, функция `createMessageUnder` из кода ниже покажет сообщение под элементом `elem`:

```
var elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, text) {
    // получить координаты
    var coords = elem.getBoundingClientRect();

    // создать элемент для сообщения
    var message = document.createElement('div');
    // стиль лучше задавать классом
    message.style.cssText = "position:fixed; color: red";

    // к координатам обязательно добавляем "px"!
    message.style.left = coords.left + "px";
    message.style.top = coords.bottom + "px";

    message.innerHTML = text;

    return message;
}

// Использование
// добавить на 5 сек в документ
var message = createMessageUnder(elem, 'Привет, мир!');
document.body.appendChild(message);
setTimeout(function() {
    document.body.removeChild(message);
}, 5000);
```

Этот код можно модифицировать, чтобы показывать сообщение слева, справа, сверху, делать это вместе с CSS-анимацией и так далее. Для этого нужно всего лишь понимать, как получить координаты.

Заметим, однако, важную деталь: при прокрутке страницы сообщение будет визуально отдаляться от кнопки.

Причина очевидна, ведь оно использует `position: fixed`, так что при прокрутке остаётся на месте,

а страница скроллируется.

Как сделать, чтобы сообщение было именно на конкретном месте документа, а не окна, мы рассмотрим в следующей главе.

✓ Задачи

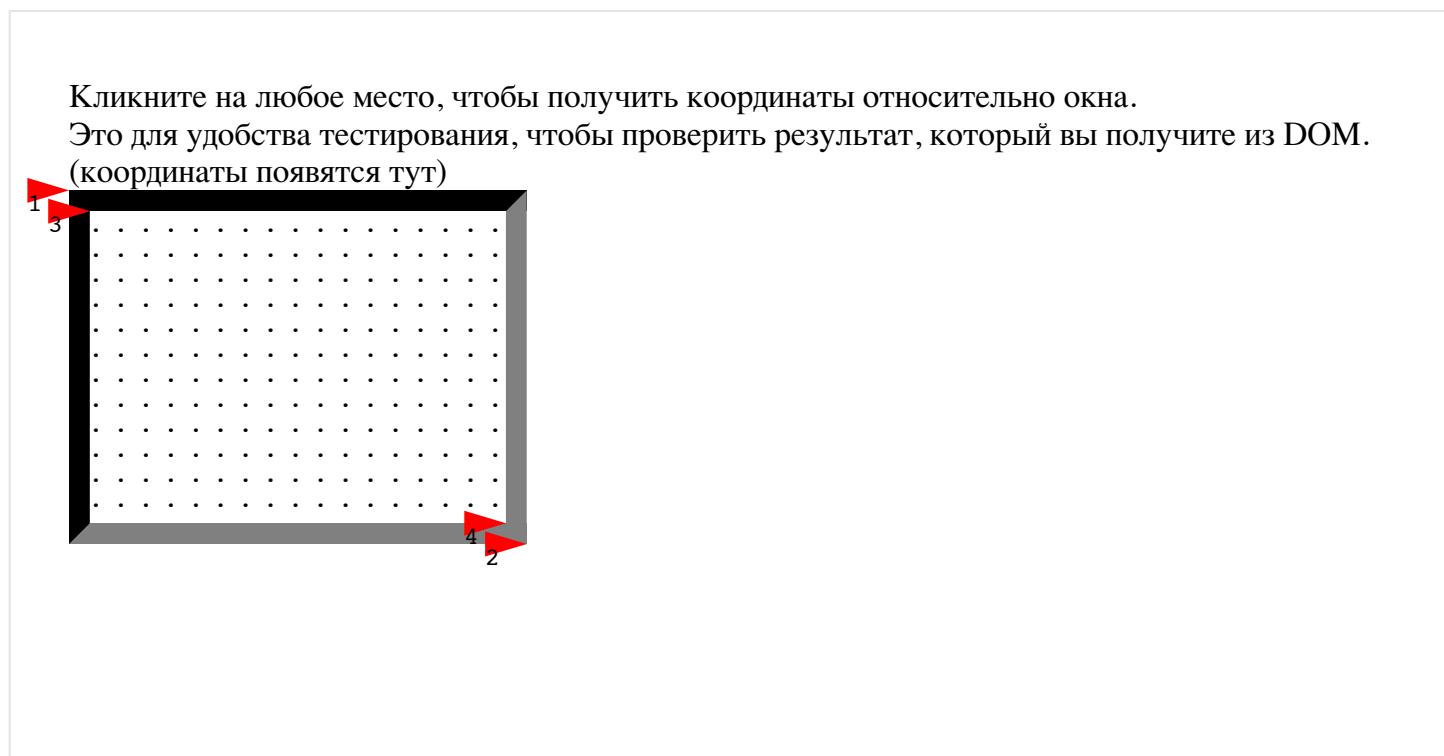
Найдите координаты точки в документе

важность: 5

В iframe'е ниже вы видите документ с зеленым «полем».

При помощи JavaScript найдите координаты указанных стрелками углов относительно окна браузера.

Для тестирования в документ добавлено удобство: клик в любом месте отображает координаты мыши относительно окна.



Ваш код должен при помощи DOM получить четыре пары координат:

1. Левый-верхний угол снаружи, это просто.
2. Правый-нижний угол снаружи, это тоже просто.
3. Левый-верхний угол внутри, это чуть сложнее.
4. Правый-нижний угол внутри, это ещё сложнее, но можно сделать даже несколькими способами.

Они должны совпадать с координатами, которые вы получите кликом по полю.

P.S. Код не должен быть как-то привязан к конкретным размерам элемента, стилям, наличию или

отсутствию рамки.

Открыть песочницу для задачи. ↗

К решению

Разместить заметку рядом с элементом

важность: 5

Создайте функцию `positionAt(anchor, position, elem)`, которая позиционирует элемент `elem`, в зависимости от `position`, сверху ("top"), справа ("right") или снизу ("bottom") от элемента `anchor`.

Используйте её, чтобы сделать функцию `showNote(anchor, position, html)`, которая показывает элемент с классом `note` и текстом `html` на позиции `position` рядом с элементом `anchor`.

Выведите заметки как здесь:

Лорем ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad
incident voluptas nisi amet! Odio temporibus nulla id unde quaerat dignissimos enim
nisi rem providez заметка сверху mpare omnis recusandae esse sequi officia sapiente.

66

- Что на завтрак, Бэрримор? - Овсянка, сэр. - А на обед? - Овсянка, сэр. - Ну а на

заметка снизу

*Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad
 incident voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim
 nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.*

Открыть песочницу для задачи. ↗

К решению

Координаты в документе

Система координат относительно страницы или, иначе говоря, относительно документа, начинается в левом-верхнем углу, но не окна, а именно страницы.

И координаты в ней означают позицию по отношению не к окну браузера, а к документу в целом.

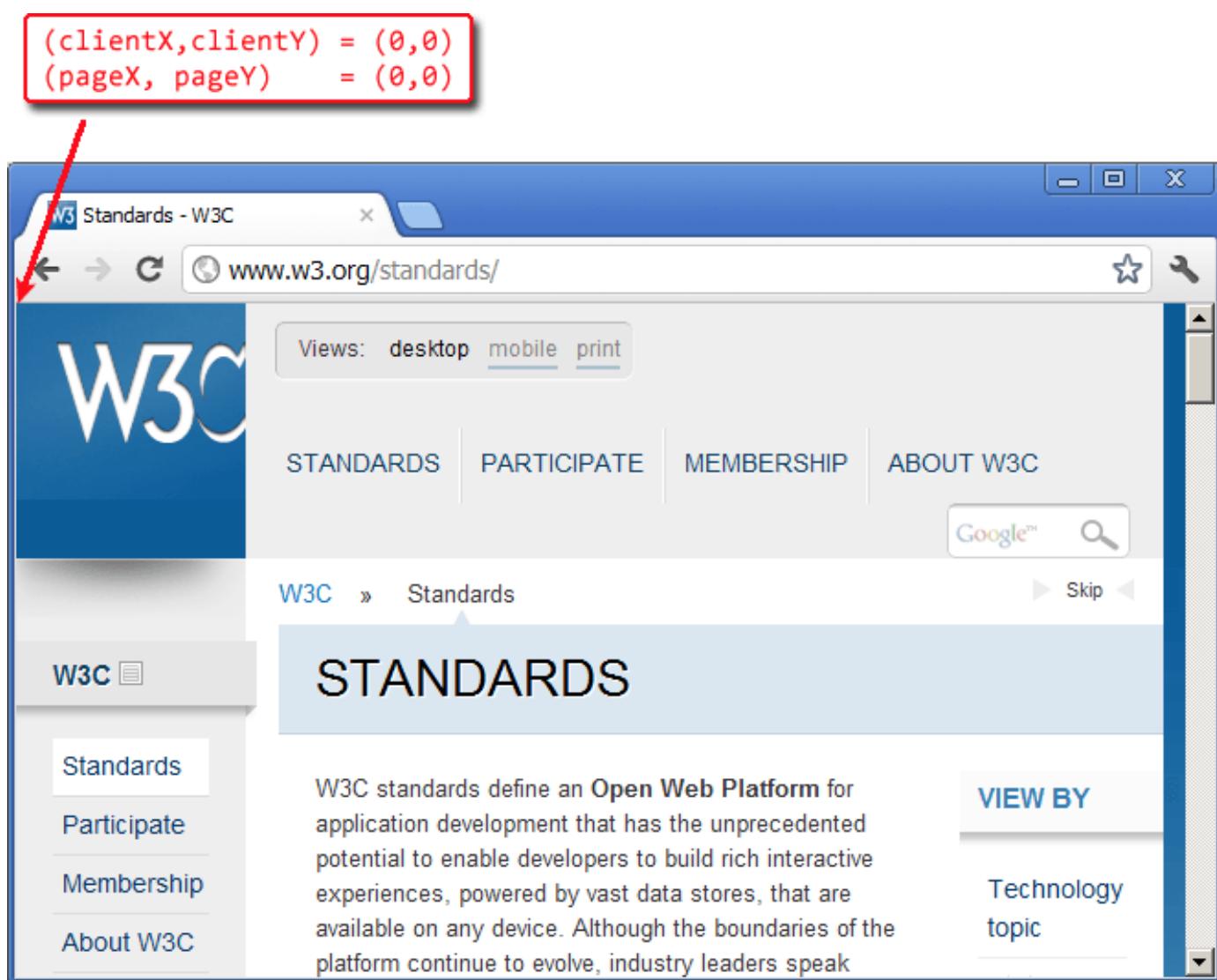
Если провести аналогию с CSS, то координаты относительно окна — это `position:fixed`, а относительно документа — `position:absolute` (при позиционировании вне других элементов, естественно).

Мы будем называть координаты в ней `pageX/pageY`.

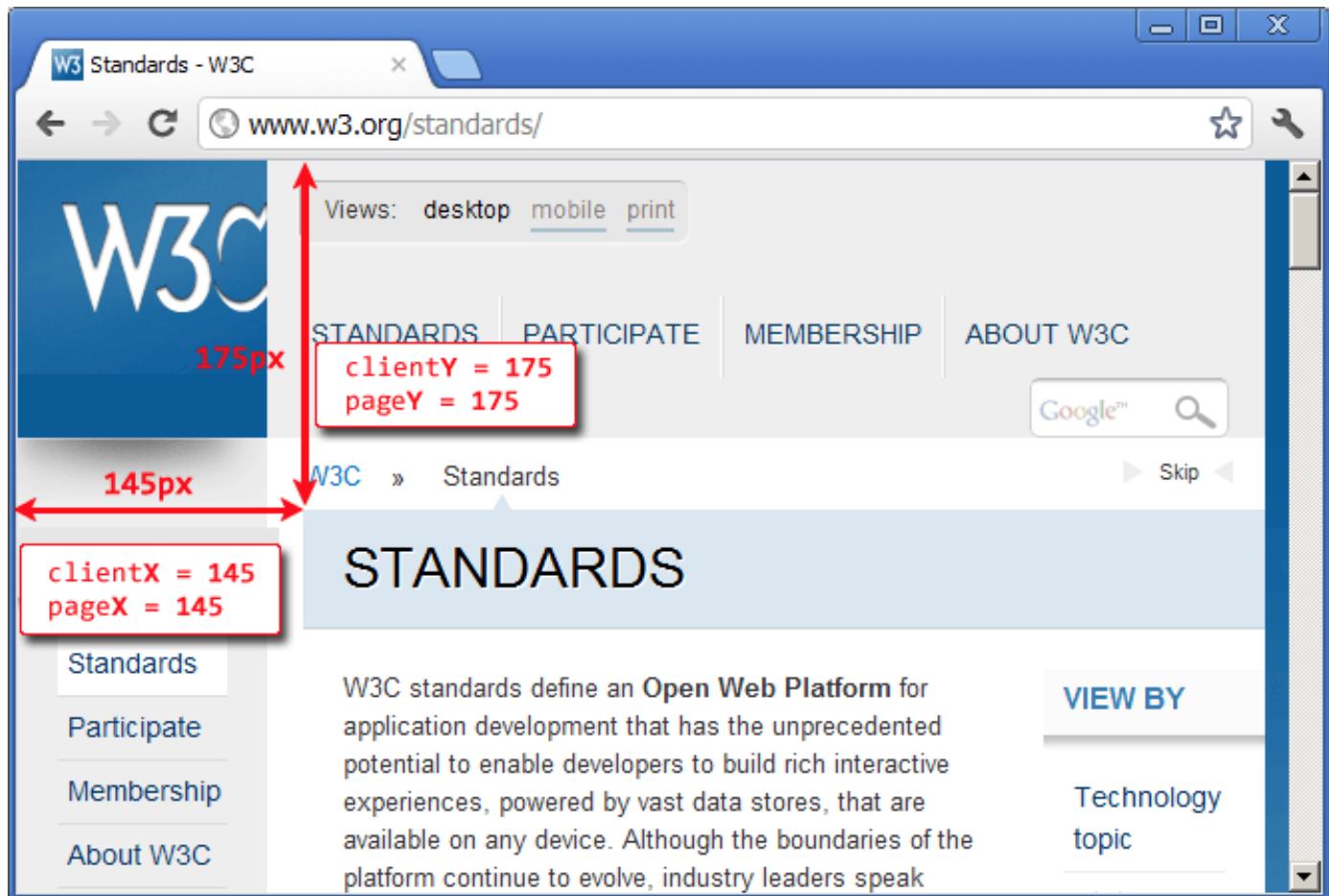
Они нужны в первую очередь для того, чтобы показывать элемент в определённом месте страницы, а не окна.

Сравнение систем координат

Когда страница не прокручена, точки начала координат относительно окна (`clientX,clientY`) и документа (`pageX,pageY`) совпадают:



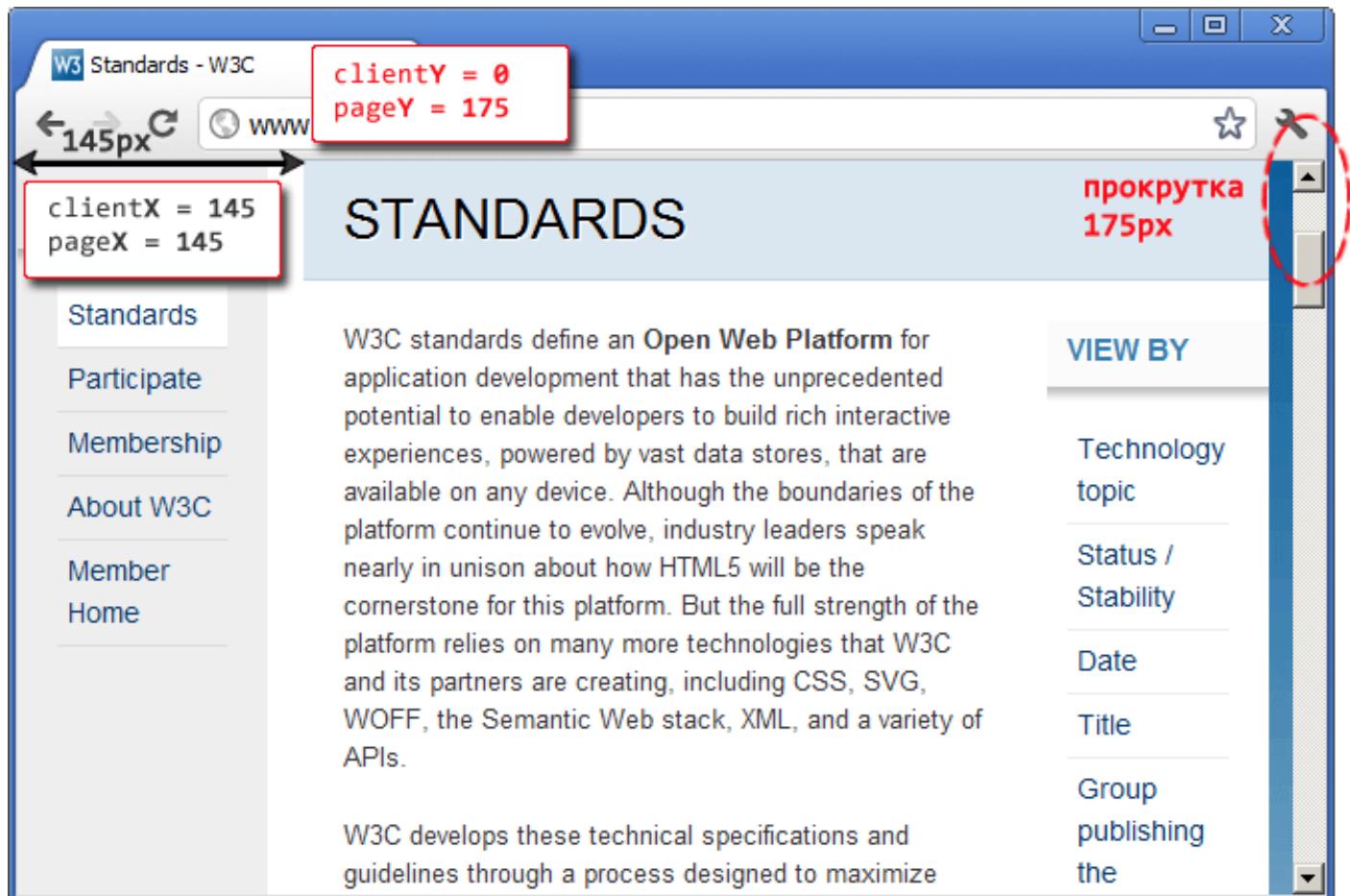
Например, координаты элемента с надписью «STANDARDS» равны расстоянию от верхней/левой границы окна:



Прокрутим страницу, чтобы элемент был на самом верху:

Посмотрите на рисунок ниже, на нём — та же страница, только прокрученная, и тот же элемент «STANDARDS».

- Координата `clientY` изменилась. Она была 175, а стала 0, так как элемент находится вверху окна.
- Координата `pageY` осталась такой же, так как отсчитывается от левого-верхнего угла документа.



Итак, координаты pageX/pageY не меняются при прокрутке, в отличие от clientX/clientY.

Получение координат

К сожалению, готовой функции для получения координат элемента относительно страницы нет. Но её можно легко написать самим.

Эти две системы координат жёстко связаны: `pageY = clientY + текущая вертикальная прокрутка`.

Наша функция `getCoords(elem)` будет брать результат `elem.getBoundingClientRect()` и прибавлять текущую прокрутку документа.

Результат `getCoords`: объект с координатами `{left: .., top: ..}`

```
function getCoords(elem) { // кроме IE8-
  var box = elem.getBoundingClientRect();

  return {
    top: box.top + pageYOffset,
    left: box.left + pageXOffset
  };
}
```

Если нужно поддерживать более старые IE, то вот альтернативный, самый кросс-браузерный вариант:

```

function getCoords(elem) {
  // (1)
  var box = elem.getBoundingClientRect();

  var body = document.body;
  var docEl = document.documentElement;

  // (2)
  var scrollTop = window.pageYOffset || docEl.scrollTop || body.scrollTop;
  var scrollLeft = window.pageXOffset || docEl.scrollLeft || body.scrollLeft;

  // (3)
  var clientTop = docEl.clientTop || body.clientTop || 0;
  var clientLeft = docEl.clientLeft || body.clientLeft || 0;

  // (4)
  var top = box.top + scrollTop - clientTop;
  var left = box.left + scrollLeft - clientLeft;

  return {
    top: top,
    left: left
  };
}

```

Разберем что и зачем, по шагам:

1. Получаем прямоугольник.
2. Считаем прокрутку страницы. Все браузеры, кроме IE8- поддерживают свойство `pageXOffset/pageYOffset`. В более старых IE, когда установлен DOCTYPE, прокрутку можно получить из `documentElement`, ну и наконец если DOCTYPE некорректен — использовать `body`.
3. В IE документ может быть смещен относительно левого верхнего угла. Получим это смещение.
4. Добавим прокрутку к координатам окна и вычтем смещение `html/body`, чтобы получить координаты всего документа.

Устаревший метод: offset*

Есть альтернативный способ нахождения координат — это пройти всю цепочку `offsetParent` от элемента вверх и сложить отступы `offsetLeft/offsetTop`.

Мы разбираем его здесь с учебной целью, так как он используется лишь в старых браузерах.

Вот функция, реализующая такой подход.

```

function getOffsetSum(elem) {
  var top = 0,
  left = 0;

  while (elem) {
    top = top + parseInt(elem.offsetTop);
    left = left + parseInt(elem.offsetLeft);
    elem = elem.offsetParent;
  }

  return {
    top: top,
    left: left
  };
}

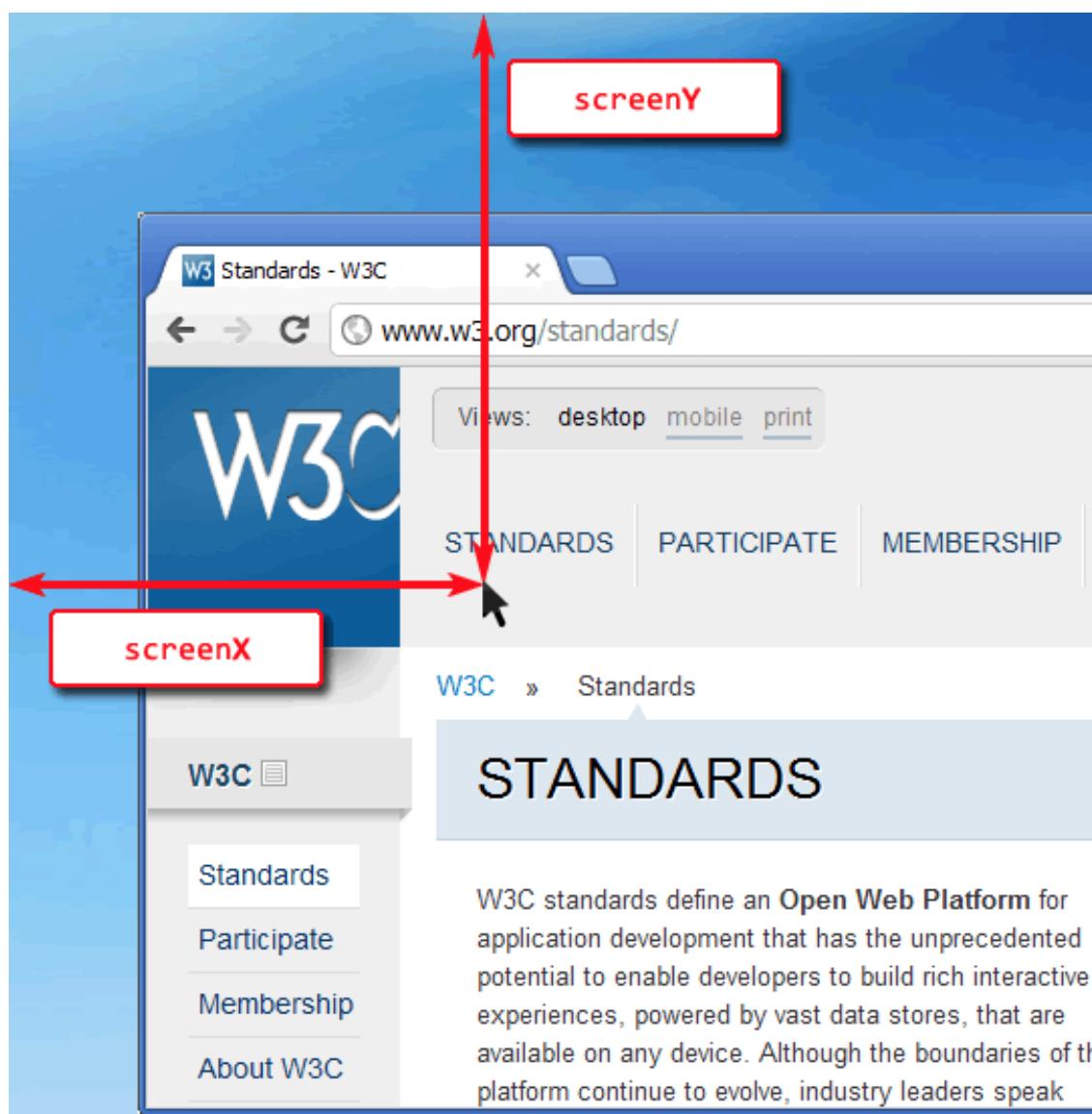
```

Казалось бы, код нормальный. И он как-то работает, но разные браузеры преподносят «сюрпризы», включая или выключая размер рамок и прокрутку из `offsetTop/Left`, некорректно учитывая позиционирование. В итоге результат не всегда верен. Можно, конечно, разобрать эти проблемы и посчитать действительно аккуратно и правильно этим способом, но зачем? Ведь есть `getBoundingClientRect`.

Координаты на экране screenX/screenY

Есть ещё одна система координат, которая используется очень редко, но для полноты картины необходимо её упомянуть.

Координаты относительно экрана `screenX/screenY` отсчитываются от его левого-верхнего угла. Имеется в виду именно весь экран, а не окно браузера.



Такие координаты могут быть полезны, например, при работе с мобильными устройствами или для открытия нового окна посередине экрана вызовом `window.open ↗`.

- Размеры экрана хранятся в глобальной переменной `screen ↗`:

```
// общая ширина/высота  
alert( screen.width + ' x ' + screen.height );  
  
// доступная ширина/высота (за вычетом таскбара и т.п.)  
alert( screen.availWidth + ' x ' + screen.availHeight );  
  
// есть и ряд других свойств screen (см. документацию)
```

- Координаты левого-верхнего угла браузера на экране хранятся в `window.screenX`, `window.screenY` (не поддерживаются IE8-):

```
alert( "Браузер находится на " + window.screenX + "," + window.screenY );
```

Они могут быть и меньше нуля, если окно частично вне экрана.

Заметим, что общую информацию об экране и браузере получить можно, а вот координаты конкретного элемента на экране — нельзя, нет аналога `getBoundingClientRect` или иного метода для этого.

Итого

У любой точки в браузере есть координаты:

- Относительно окна `window` — `elem.getBoundingClientRect()`.
- Относительно документа `document` — добавляем прокрутку, во всех фреймворках есть готовая функция.
- Относительно экрана `screen` — можно узнать координаты браузера, но не элемента.

Иногда в старом коде можно встретить использование `offsetTop/Left` для подсчёта координат. Это очень старый и неправильный способ, не стоит его использовать.

Координаты будут нужны нам далее, при работе с событиями мыши (координаты клика) и элементами (перемещение).

✓ Задачи

Область видимости для документа

важность: 5

Напишите функцию `getDocumentScroll()`, которая возвращает объект с информацией о текущей прокрутке и области видимости.

Свойства объекта-результата:

- `top` — координата верхней границы видимой части (относительно документа).
- `bottom` — координата нижней границы видимой части (относительно документа).

- `height` — полная высота документа, включая прокрутку.

В этой задаче учитываем только вертикальную прокрутку: горизонтальная делается аналогично, а нужна сильно реже.

[К решению](#)

Разместить заметку рядом с элементом (absolute)

важность: 5

Модифицируйте решение задачи [Разместить заметку рядом с элементом](#), чтобы при прокрутке страницы заметка не убегала от элемента.

Используйте для этого координаты относительно документа и `position: absolute` вместо `position: fixed`.

В качестве исходного документа используйте решение задачи [Разместить заметку рядом с элементом](#), для тестирования прокрутки добавьте стиль `<body style="height: 2000px">`.

[К решению](#)

Разместить заметку внутри элемента

важность: 5

Расширьте предыдущую задачу [Разместить заметку рядом с элементом \(absolute\)](#): научите функцию `positionAt(anchor, position, elem)` вставлять `elem` внутрь `anchor`.

Новые значения `position`:

- `top-out, right-out, bottom-out` — работают так же, как раньше, то есть вставляют `elem` над/справа/под `anchor`.
- `top-in, right-in, bottom-in` — вставляют `elem` внутрь `anchor`: к верхней границе/правой/нижней.

Например:

```
// покажет note сверху blockquote
positionAt(blockquote, "top-out", note);

// покажет note сверху-внутри blockquote
positionAt(blockquote, "top-in", note);
```

Пример результата:

*Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad
incident voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim
nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.*

запоминаем
заметка top-out

“

Чтобы заметка
заметка bottom-in

*Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad
incident voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim
nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.*

В качестве исходного документа возьмите решение задачи [Разместить заметку рядом с элементом \(absolute\)](#).

[К решению](#)

Итого

В этой главе кратко перечислены основные свойства и методы DOM, которые мы изучили. Их уже довольно много.

Используйте её, чтобы по-быстрому вспомнить и прокрутить в голове то, что изучали ранее. Все ли эти свойства вам знакомы?

Кое-где стоит ограничение на версии IE, но на все свойства можно найти или сделать или найти полифилл, с которым их можно использовать везде.

Создание

`document.createElement(tag)`

Создать элемент с тегом `tag`

document.createTextNode(txt)

Создать текстовый узел с текстом txt

node.cloneNode(deep)

Клонировать существующий узел, если deep=false, то без потомков.

Свойства узлов

node.nodeType

Тип узла: 1(элемент) / 3(текст) / другие.

elem.tagName

Тег элемента.

elem.innerHTML

HTML внутри элемента.

elem.outerHTML

Весь HTML элемента, включая сам тег. На запись использовать с осторожностью, так как не модифицирует элемент, а вставляет новый вместо него.

node.data / node.nodeValue

Содержимое узла любого типа, кроме элемента.

node.textContent

Текстовое содержимое узла, для элементов содержит текст с вырезанными тегами (IE9+).

elem.hidden

Если поставить true, то элемент будет скрыт (IE10+).

Атрибуты

elem.getAttribute(name), elem.hasAttribute(name), elem.setAttribute(name, value)

Чтение атрибута, проверка наличия и запись.

elem.dataset.*

Значения атрибутов вида data-* (IE10+).

Ссылки

document.documentElement

Элемент <HTML>

document.body

Элемент <BODY>

document.head

Элемент <HEAD> (IE9+)

По всем узлам:

- parentNode
- nextSibling previousSibling
- childNodes firstChild lastChild

Только по элементам:

- parentElement
- nextElementSibling previousElementSibling
- children, firstElementChild lastElementChild

Все они IE9+, кроме children, который работает в IE8-, но содержит не только элементы, но и комментарии (ошибка в браузере).

Дополнительно у некоторых типов элементов могут быть и другие ссылки, свойства, коллекции для навигации, например для таблиц:

table.rows[N]

строка TR номер N.

tr.cells[N]

ячейка TH/TD номер N.

tr.sectionRowIndex

номер строки в таблице в секции THEAD/TBODY.

td.cellIndex

номер ячейки в строке.

Поиск

***.querySelector(css)**

По селектору, только первый элемент

***.querySelectorAll(css)**

По селектору CSS3, в IE8 по CSS 2.1

document.getElementById(id)

По уникальному id

document.getElementsByName(name)

По атрибуту name, в IE9+ работает только для элементов, где name предусмотрен стандартом.

***.getElementsByTagName(tag)**

По тегу tag

***.getElementsByClassName(class)**

По классу, IE9+, корректно работает с элементами, у которых несколько классов.

Вообще, обычно можно использовать только querySelector/querySelectorAll. Методы getElement* работают быстрее (за счёт более оптимальной внутренней реализации), но в 99% случаев это различие очень небольшое и роли не играет.

Дополнительно есть методы:

elem.matches(css)

Проверяет, подходит ли элемент под CSS-селектор.

elem.closest(css)

Ищет ближайший элемент сверху по иерархии DOM, подходящий под CSS-селектор. Первым проверяется сам elem. Этот элемент возвращается.

elemA.contains(elemB)

Возвращает true, если elemA является предком (содержит) elemB.

elemA.compareDocumentPosition(elemB)

Возвращает битовую маску, которая включает в себя отношение вложенности между elemA и elemB, а также — какой из элементов появляется в DOM первым.

Изменение

- parent.appendChild(newChild)
- parent.removeChild(child)
- parent.insertBefore(newChild, refNode)
- parent.insertAdjacentHTML("beforeBegin|afterBegin|beforeEnd|afterEnd", html)
- parent.insertAdjacentElement("beforeBegin|...|afterEnd", text) (кроме FF)
- parent.insertAdjacentText("beforeBegin|...|afterEnd", text) (кроме FF)
- document.write(...)

Скорее всего, понадобятся полифиллы для:

- node.append(...nodes)
- node.prepend(...nodes)

- `node.after(...nodes)`,
- `node.before(...nodes)`
- `node.replaceWith(...nodes)`

Классы и стили

`elem.className`

Атрибут `class`

`elem.classList.add(class) remove(class) toggle(class) contains(class)`

Управление классами, для IE9- есть [эмуляция ↗](#).

`elem.style`

Стили в атрибуте `style` элемента

`getComputedStyle(elem, "")`

Стиль, с учётом всего каскада, вычисленный и применённый (только чтение)

Размеры и прокрутка элемента

`clientLeft/Top`

Ширина левой/верхней рамки border

`clientWidth/Height`

Ширина/высота внутренней части элемента, включая содержимое и padding, не включая полосу прокрутки (если есть).

`scrollWidth/Height`

Ширина/высота внутренней части элемента, с учетом прокрутки.

`scrollLeft/Top`

Ширина/высота прокрученной области.

`offsetWidth/Height`

Полный размер элемента: ширина/высота, включая border.

Размеры и прокрутка страницы

- ширина/высота видимой области: `document.documentElement.clientHeight`
- прокрутка(чтение): `window.pageYOffset || document.documentElement.scrollTop`
- прокрутка(изменение):
 - `window.scrollBy(x,y)`: на x,y относительно текущей позиции.

- `window.scrollTo(pageX, pageY)`: на координаты в документе.
- `elem.scrollIntoView(true/false)`: прокрутить, чтобы `elem` стал видимым и оказался вверху окна(`true`) или внизу(`false`)

Координаты

- относительно окна: `elem.getBoundingClientRect()`
- относительно документа: `elem.getBoundingClientRect() + прокрутка страницы`
- получить элемент по координатам: `document.elementFromPoint(clientX, clientY)`

Список намеренно сокращён, чтобы было проще найти то, что нужно.

Основы работы с событиями

Введение в браузерные события, общие свойства всех событий и приёмы работы с ними.

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. Посмотрим список самых часто используемых, пока просто для ознакомления:

События мыши

- `click` — происходит, когда кликнули на элемент левой кнопкой мыши
- `contextmenu` — происходит, когда кликнули на элемент правой кнопкой мыши
- `mouseover` — возникает, когда на элемент наводится мышь
- `mousedown` и `mouseup` — когда кнопку мыши нажали или отжали
- `mousemove` — при движении мыши

События на элементах управления

- `submit` — посетитель отправил форму `<form>`
- `focus` — посетитель фокусируется на элементе, например нажимает на `<input>`

Клавиатурные события

- `keydown` — когда посетитель нажимает клавишу
- `keyup` — когда посетитель отпускает клавишу

События документа

- `DOMContentLoaded` — когда HTML загружен и обработан, DOM документа полностью построен и

доступен.

События CSS

- `transitionend` — когда CSS-анимация завершена.

Также есть и много других событий.

Назначение обработчиков событий

Событию можно назначить *обработчик*, то есть функцию, которая сработает, как только событие произошло.

Именно благодаря обработчикам JavaScript-код может реагировать на действия посетителя.

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 value="Нажми меня" onclick="alert('Клик!')" type="button">
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте `onclick`.

Обратите внимание, для содержимого атрибута `onclick` используются одинарные 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!")"`, с двойными кавычками внутри, не будет работать. Если вам действительно нужно использовать именно двойные кавычки, то это можно сделать, заменив их на `"`; то есть так: `onclick="alert("Клик!");"`.

Однако, обычно этого не требуется, так как прямо в разметке пишутся только очень простые обработчики. Если нужно сделать что-то сложное, то имеет смысл описать это в функции, и в обработчике вызвать уже её.

Следующий пример по клику запускает функцию `countRabbits()`.

```

<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">

    <script>
        function countRabbits() {
            for(var i=1; i<=3; i++) {
                alert("Кролик номер " + i);
            }
        }
    </script>
</head>
<body>
    <input type="button" onclick="countRabbits()" value="Считать кроликов!" />
</body>
</html>

```

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому **ONCLICK** будет работать так же, как **onClick** или **onCLICK...** Но, как правило, атрибуты пишут в нижнем регистре: **onclick**.

Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента **on<событие>**.

Пример установки обработчика **click**:

```

<input id="elem" type="button" value="Нажми меня" />
<script>
    elem.onclick = function() {
        alert( 'Спасибо' );
    };
</script>

```

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство **onclick**.

Этот способ, по сути, аналогичен предыдущему.

Обработчик хранится именно в DOM-свойстве, а атрибут — лишь один из способов его инициализации.

Эти два примера кода работают одинаково:

1. Только HTML:

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

2. HTML + JS:

```
<input type="button" id="button" value="Кнопка" />
<script>
  button.onclick = function() {
    alert( 'Клик!' );
  };
</script>
```

Так как DOM-свойство onclick, в итоге, одно, то назначить более одного обработчика так нельзя.

В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

```
<input type="button" id="elem" onclick="alert('До')" value="Нажми меня" />
<script>
  elem.onclick = function() { // перезапишет существующий обработчик
    alert( 'После' ); // выведется только это
  };
</script>
```

Нажми меня

Кстати, обработчиком можно назначить и уже существующую функцию:

```
function sayThanks() {
  alert( 'Спасибо!' );
}

elem.onclick = sayThanks;
```

Если обработчик надоел — его всегда можно убрать назначением elem.onclick = null.

Доступ к элементу через this

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором он сработал.

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>
```

Частые ошибки

Если вы только начинаете работать с событиями — обратите внимание на следующие особенности.

Функция должна быть присвоена как sayThanks, а не sayThanks().

```
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 из атрибута, он автоматически создает функцию из его содержимого. Поэтому последний пример — фактически то же самое, что:

```
button.onclick = function() {  
    sayThanks(); // содержимое атрибута  
};
```

Используйте именно функции, а не строки.

Назначение обработчика строкой `elem.onclick = "alert(1)"` можно иногда увидеть в древнем коде. Это будет работать, но не рекомендуется, могут быть проблемы при сжатии JavaScript Да и вообще, передавать код в виде строки по меньшей мере странно в языке, который поддерживает Function Expressions. Это возможно лишь по соображениям совместимости, не делайте так.

Не используйте `setAttribute`.

Такой вызов работать не будет:

```
// при нажатии на body будут ошибки  
// потому что при назначении в атрибут функция будет преобразована в строку  
document.body.setAttribute('onclick', function() { alert(1) });
```

Регистр DOM-свойства имеет значение.

При назначении через DOM нужно использовать свойство `onclick`, а не `ONCLICK`.

Недостаток назначения через свойство

Фундаментальный недостаток описанных выше 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
```

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов, которые свободны от указанного недостатка.

addEventListener и removeEventListener

Методы `addEventListener` и `removeEventListener` являются современным способом назначить или удалить обработчик, и при этом позволяют использовать сколько угодно любых обработчиков.

Назначение обработчика осуществляется вызовом `addEventListener` с тремя аргументами:

```
element.addEventListener(event, handler[, phase]);
```

event

Имя события, например `click`

handler

Ссылка на функцию, которую надо поставить обработчиком.

phase

Необязательный аргумент, «фаза», на которой обработчик должен сработать. Этот аргумент редко нужен, мы его рассмотрим позже.

Удаление обработчика осуществляется вызовом `removeEventListener`:

```
// передать те же аргументы, что были у addEventListener
element.removeEventListener(event, handler[, phase]);
```

⚠ Удаление требует именно ту же функцию

Для удаления нужно передать именно ту функцию-обработчик которая была назначена.

Вот так `removeEventListener` не сработает:

```
elem.addEventListener( "click" , function() {alert('Спасибо!')});  
// ....  
elem.removeEventListener( "click", function() {alert('Спасибо!')});
```

В `removeEventListener` передана не та же функция, а другая, с одинаковым кодом, но это не важно.

Вот так правильно:

```
function handler() {  
  alert( 'Спасибо!' );  
}  
  
input.addEventListener("click", handler);  
// ....  
input.removeEventListener("click", handler);
```

Обратим внимание — если функцию не сохранить где-либо, а просто передать в `addEventListener`, как в предыдущем коде, то потом получить её обратно, чтобы снять обработчик, будет невозможно. Нет метода, который позволяет считать обработчики событий, назначенные через `addEventListener`.

Метод `addEventListener` позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

```
<input id="elem" type="button" value="Нажми меня"/>  
  
<script>  
  function handler1() {  
    alert('Спасибо!');  
  };  
  
  function handler2() {  
    alert('Спасибо ещё раз!');  
  }  
  
  elem.onclick = function() { alert("Привет"); };  
  elem.addEventListener("click", handler1); // Спасибо!  
  elem.addEventListener("click", handler2); // Спасибо ещё раз!  
</script>
```

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство и через `addEventListener`. Однако, во избежание путаницы, рекомендуется выбрать один способ.

addEventListener работает всегда, а DOM-свойство — нет

У специальных методов есть ещё одно преимущество перед DOM-свойствами.

Есть некоторые события, которые нельзя назначить через DOM-свойство, но можно через `addEventListener`.

Например, таково событие `transitionend`, то есть окончание CSS-анимации. В большинстве браузеров оно требует назначения через `addEventListener`.

Вы можете проверить это, запустив код в примере ниже. Как правило, сработает лишь второй обработчик, но не первый.

```
<style>
  button {
    transition: width 1s;
    width: 100px;
  }

  .wide {
    width: 300px;
  }
</style>

<button id="elem" onclick="this.classList.toggle('wide');">
  Нажми меня
</button>

<script>
  elem.ontransitionend = function() {
    alert( "ontransitionend" ); // не сработает
 };

  elem.addEventListener("transitionend", function() {
    alert( "addEventListener" ); // сработает по окончании анимации
  });
</script>
```

Отличия IE8-

При работе с событиями в IE8- есть много отличий. Как правило, они формальны — некое свойство или метод называются по-другому. Начиная с версии 9, также работают и стандартные свойства и методы.

В IE8- вместо `addEventListener/removeEventListener` используются свои методы.

Назначение обработчика осуществляется вызовом `attachEvent`:

```
element.attachEvent("on" + event, handler);
```

Удаление обработчика — вызовом `detachEvent`:

```
element.detachEvent("on" + event, handler);
```

Например:

```
function handler() {  
    alert('Спасибо!');  
}  
button.attachEvent("onclick", handler) // Назначение обработчика  
// ....  
button.detachEvent("onclick", handler) // Удаление обработчика
```

Как видите, почти то же самое, только событие должно включать префикс on.

У обработчиков, назначенных с attachEvent, нет this

Обработчики, назначенные с attachEvent не получают this!

Это важная особенность и подводный камень старых IE.

Чтобы ваш код работал в старом IE, нужно либо использовать DOM-свойства, то есть onclick, либо подключить полифилл для современных методов, например [такой](#) или с сервиса [polyfill.io](#) или какой-то другой.

Итого

Есть три способа назначения обработчиков событий:

1. Атрибут HTML: onclick="...".
2. Свойство: elem.onclick = function.
3. Специальные методы:

- Современные: elem.addEventListener(событие, handler[, phase]), удаление через removeEventListener.
- Для старых IE8: elem.attachEvent(on+событие, handler), удаление через detachEvent.

Сравнение addEventListener и onclick:

Достоинства

- Некоторые события можно назначить только через addEventListener.
- Метод addEventListener позволяет назначить много обработчиков на одно событие.

Недостатки

- Обработчик, назначенный через onclick, проще удалить или заменить.
- Метод onclick кросс-браузерный.

Этим введением мы только открывает работу с событиями, но вы уже можете решать разнообразные задачи с их использованием.

✓ Задачи

Спрятать при клике

важность: 5

Используя JavaScript, сделайте так, чтобы при клике на кнопку исчезал элемент с id="text".

Демо:

Нажмите, чтобы спрятать текст

Текст

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Спрятаться

важность: 5

Создайте кнопку, при клике на которую, она будет скрывать сама себя.

[К решению](#)

Какие обработчики сработают?

важность: 5

В переменной `button` находится кнопка.

Изначально обработчиков на ней нет.

Что будет выведено при клике после выполнения кода?

```
button.addEventListener("click", function() { alert("1"); });
button.removeEventListener("click", function() { alert("1"); });
button.onclick = function() { alert(2); };
```

[К решению](#)

Раскрывающееся меню

важность: 5

Создайте меню, которое раскрывается/сворачивается при клике:

► Сладости (нажми меня)!

P.S. HTML/CSS исходного документа понадобится изменить.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Спрятать сообщение

важность: 5

Есть список сообщений. Добавьте каждому сообщению по кнопке для его скрытия.

Результат:

Лошадь

[x]

Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади, вымершей в дикой природе, за исключением небольшой популяции лошади Пржевальского.

Осёл

[x]

Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства и культуры человека. Все одомашненные ослы относятся к африканским ослам.

Корова, а также пара слов о диком быке, о волах и о тёлках.

[x]

Корова — самка домашнего быка, одомашненного подвида дикого быка, парнокопытного жвачного животного семейства полорогих. Самцы вида называются быками, молодняк — телятами, кастрированные самцы — волами. Молодых (до первой стельности) самок называют тёлками.

P.S. Как лучше отобразить кнопку справа-сверху: через position: absolute или float:right?
Почему?

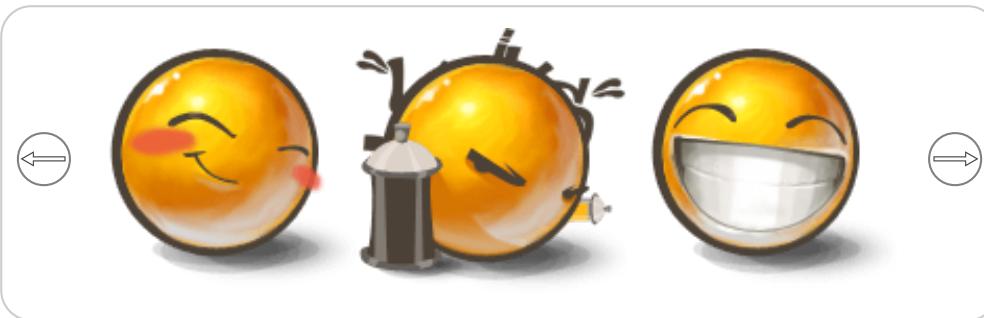
[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Карусель

важность: 4

Напишите «Карусель» — ленту изображений, которую можно листать влево-вправо нажатием на стрелочки.



В дальнейшем к ней можно легко добавить анимацию, динамическую подгрузку и другие возможности.

P.S. В этой задаче разработка HTML/CSS-структурь составляет 90% решения.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Порядок обработки событий

События могут возникать не только по очереди, но и «пачкой» по много сразу. Возможно и такое, что во время обработки одного события возникают другие, например пока выполнялся код для `onclick` — посетитель нажал кнопку на клавиатуре (событие `keydown`).

Здесь мы разберём, как браузер обычно работает с одновременно возникающими событиями и какие есть исключения из общего правила.

Главный поток

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

Он выполняет команды последовательно, может делать только одно дело одновременно и блокируется при выводе модальных окон, таких как `alert`.

ⓘ Дополнительные потоки тоже есть

Есть и другие, служебные потоки, например, для сетевых коммуникаций.

Поэтому скачивание файлов может продолжаться пока главный поток ждёт реакции на `alert`. Но управлять служебными потоками мы не можем.

ⓘ Web Workers

Существует спецификация [Web Workers](#), которая позволяет запускать дополнительные JavaScript-процессы(`workers`).

Они могут обмениваться сообщениями с главным процессом, но у них свои переменные, и работают они также сами по себе.

Такие дополнительные процессы не имеют доступа к DOM, поэтому они полезны, преимущественно, при вычислениях, чтобы загрузить несколько ядер/процессоров одновременно.

Очередь событий

Произошло одновременно несколько событий или во время работы одного случилось другое — как главному потоку обработать это?

Если главный поток прямо сейчас занят, то он не может срочно выйти из середины одной функции и прыгнуть в другую. А потом третью. Отладка при этом могла бы превратиться в кошмар, потому что пришлось бы разбираться с совместным состоянием нескольких функций сразу.

Поэтому используется альтернативный подход.

Когда происходит событие, оно попадает в очередь.

Внутри браузера непрерывно работает «главный внутренний цикл», который следит за состоянием очереди и обрабатывает события, запускает соответствующие обработчики и т.п.

Иногда события добавляются в очередь сразу пачкой.

Например, при клике на элементе генерируется несколько событий:

1. Сначала `mousedown` — нажата кнопка мыши.
2. Затем `mouseup` — кнопка мыши отпущена и, так как это было над одним элементом, то дополнительно генерируется `click` (два события сразу).

Таким образом, при нажатии кнопки мыши в очередь попадёт событие `mousedown`, а при отпускании — сразу два события: `mouseup` и `click`. Браузер обработает их строго одно за другим: `mousedown` → `mouseup` → `click`.

При этом каждое событие из очереди обрабатывается полностью отдельно от других.

Вложенные (синхронные) события

Обычно возникающие события «становятся в очередь».

Но в тех случаях, когда событие инициируется не посетителем, а кодом, то оно, как правило, обрабатывается синхронно, то есть прямо сейчас.

Рассмотрим в качестве примера событие `onfocus`.

Пример: событие `onfocus`

Когда посетитель фокусируется на элементе, возникает событие `onfocus`. Обычно оно происходит, когда посетитель кликает на поле ввода, например:

```
<p>При фокусе на поле оно изменит значение.</p>
<input type="text" onfocus="this.value = 'Фокус!'" value="Кликни меня">
```

При фокусе на поле оно изменит значение.

Кликни меня

Но ту же фокусировку можно вызвать и явно, вызовом метода `elem.focus()`:

```
<input type="text" id="elem" onfocus="this.value = 'Фокус!'">

<script>
  // сфокусируется на input и вызовет обработчик onfocus
  elem.focus();
</script>
```

В главе [Фокусировка: focus/blur](#) мы познакомимся с этим событием подробнее, а пока — нажмите на кнопку в примере ниже.

При этом обработчик `onclick` вызовет метод `focus()` на текстовом поле `text`. Код обработчика `onfocus`, который при этом запустится, сработает синхронно, прямо сейчас, до завершения `onclick`.

```
<input type="button" id="button" value="Нажми меня">
<input type="text" id="text" size="60">

<script>

button.onclick = function() {
  text.value += ' ->в onclick';

  text.focus(); // вызов инициирует событие onfocus

  text.value += ' из onclick-> ';
};

text.onfocus = function() {
  text.value += ' !focus! ';
};
</script>
```

Нажми меня

При клике на кнопке в примере выше будет видно, что управление вошло в `onclick`, затем перешло в `onfocus`, затем вышло из `onclick`.

⚠️ Исключение в IE

Так ведут себя все браузеры, кроме IE.

В нём событие `onfocus` — всегда 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.

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

А что, если мы хотим, чтобы *сначала* закончилась обработка `onclick`, а потом уже произошла обработка `onfocus` и связанные с ней действия?

Можно добиться и этого.

Один вариант — просто переместить строку `text.focus()` вниз кода обработчика `onclick`.

Если это неудобно, можно запланировать `text.focus()` чуть позже через `setTimeout(..., 0)`, вот так

```
<input type="button" id="button" value="Нажми меня">
<input type="text" id="text" size="60">

<script>
  button.onclick = function() {
    text.value += ' -> в onclick ';
    setTimeout(function() {
      text.focus(); // сработает после onclick
    }, 0);
    text.value += ' из onclick-> ';
  };
  text.onfocus = function() {
    text.value += ' !focus! ';
  };
</script>
```

Нажми меня

Такой вызов обеспечит фокусировку через минимальный «тик» таймера, по стандарту равный 4мс. Обычно такая задержка не играет роли, а необходимую асинхронность мы получили.

Итого

- JavaScript выполняется в едином потоке. Современные браузеры позволяют порождать подпроцессы [Web Workers ↗](#), они выполняются параллельно и могут отправлять/принимать сообщения, но не имеют доступа к DOM.
- Обычно события становятся в очередь и обрабатываются в порядке поступления, асинхронно, независимо друг от друга.
- Синхронными являются вложенные события, инициированные из кода.
- Чтобы сделать событие гарантированно асинхронным, используется вызов через `setTimeout(func, 0)`.

Отложенный вызов через `setTimeout(func, 0)` используется не только в событиях, а вообще — всегда, когда мы хотим, чтобы некая функция `func` сработала после того, как текущий скрипт завершится.

Объект 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.

Детали произошедшего браузер записывает в «объект события», который передаётся первым аргументом в обработчик.

Свойства объекта события

Пример ниже демонстрирует использование объекта события:

```
<input type="button" value="Нажми меня" id="elem">

<script>
elem.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert(event.clientX + ":" + event.clientY);
}
</script>
```

Свойства объекта event:

event.type

Тип события, в данном случае click

event.currentTarget

Элемент, на котором сработал обработчик. Значение — в точности такое же, как и у this, но бывают ситуации, когда обработчик является методом объекта и его this при помощи bind привязан к этому объекту, тогда мы можем использовать event.currentTarget.

event.clientX / event.clientY

Координаты курсора в момент клика (относительно окна)

Есть также и ряд других свойств, в зависимости от событий, которые мы разберём в дальнейших главах, когда будем подробно знакомиться с событиями мыши, клавиатуры и так далее.

Объект события доступен и в HTML

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

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

Тип события

Это возможно потому, что когда браузер из атрибута создаёт функцию-обработчик, то она выглядит так: `function(event) { alert(event.type) }`. То есть, её первый аргумент называется "event".

Особенности IE8-

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

Работает это так:

```
elem.onclick = function() {
  // window.event - объект события
  alert( window.event.clientX );
};
```

Кросбраузерное решение

Универсальное решение для получения объекта события:

```
element.onclick = function(event) {
  event = event || window.event; // (*)

  // Теперь event - объект события во всех браузерах.
};
```

Строка (*), в случае, если функция не получила `event` (IE8-), использует `window.event`-событие `event`.

Можно написать и иначе, если мы сами не используем переменную `event` в замыкании:

```
element.onclick = function(e) {
  e = e || event;

  // Теперь e - объект события во всех браузерах.
};
```

Итого

- Объект события содержит ценную информацию о деталях события.
- Он передается первым аргументом `event` в обработчик для всех браузеров, кроме IE8-, в которых используется глобальная переменная `window.event`.

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

```
element.onclick = function(event) {
    event = event || window.event;

    // Теперь event - объект события во всех браузерах.
};
```

Еще вариант:

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

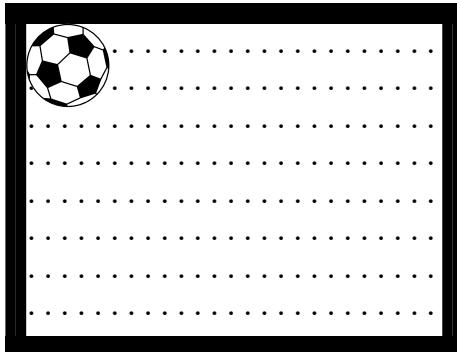
✓ Задачи

Передвигать мяч по полю

важность: 5

Сделайте так, что при клике по полю мяч перемещался на место клика.

Кликните на любое место поля, чтобы мяч перелетел туда.



Требования:

- Мяч после перелёта должен становиться центром ровно под курсор мыши, если это возможно без вылета за край поля.

- CSS-анимация не обязательна, но желательна.
- Мяч должен останавливаться у границ поля, ни в коем случае не вылетать за них.
- При прокрутке страницы с полем ничего не должно ломаться.

Замечания:

- Код не должен зависеть от конкретных размеров мяча и поля.
- Вам пригодятся свойства `event.clientX`/`event.clientY`

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Всплытие и перехват

Давайте сразу начнём с примера.

Этот обработчик для `<div>` сработает, если вы кликните по вложенному тегу `` 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

Вам не кажется это странным? Почему же сработал обработчик на `<div>`, если клик произошёл на ``?

Всплытие

Основной принцип всплытия:

При наступлении события обработчики сначала срабатывают на самом вложенном элементе, затем на его родителе, затем выше и так далее, вверх по цепочке вложенности.

Например, есть 3 вложенных элемента `FORM` > `DIV` > `P`, с обработчиком на каждом:

```

<style>
body * {
  margin: 10px;
  border: 1px solid blue;
}
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>

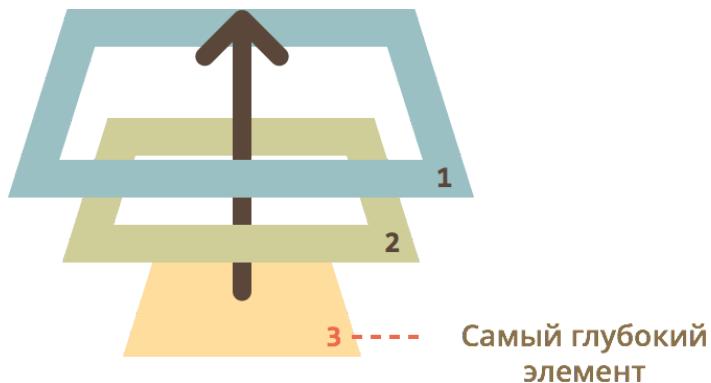
```

FORM

DIV

P

Всплытие гарантирует, что клик по внутреннему `<p>` вызовет обработчик `onclick` (если есть) сначала на самом `<p>`, затем на элементе `<div>` далее на элементе `<form>`, и так далее вверх по цепочке родителей до самого `document`.



Поэтому если в примере выше кликнуть на P, то последовательно выведутся `alert: p → div → form`.

Этот процесс называется *всплытием*, потому что события «всплывают» от внутреннего элемента вверх через родителей, подобно тому, как всплывает пузырек воздуха в воде.

⚠ Всплывают почти все события.

Ключевое слово в этой фразе — «почти».

Например, событие `focus` не всплывает. В дальнейших главах мы будем детально знакомиться с различными событиями и увидим ещё примеры.

Целевой элемент `event.target`

На каком бы элементе мы ни поймали событие, всегда можно узнать, где конкретно оно произошло.

Самый глубокий элемент, который вызывает событие, называется «целевым» или «исходным» элементом и доступен как `event.target`.

Отличия от `this` (`=event.currentTarget`):

- `event.target` — это **исходный элемент**, на котором произошло событие, в процессе всплытия он неизменен.
- `this` — это **текущий элемент**, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик `form.onclick`, то он «поймает» все клики внутри формы. Где бы ни был клик внутри — он всплывёт до элемента `<form>`, на котором сработает обработчик.

При этом:

- `this` (`=event.currentTarget`) всегда будет сама форма, так как обработчик сработал на ней.
- `event.target` будет содержать ссылку на конкретный элемент внутри формы, самый вложенный, на котором произошёл клик.

[Смотреть пример онлайн ↗](#)

Возможна и ситуация, когда `event.target` и `this` — один и тот же элемент, например если в форме нет других тегов и клик был на самом элементе `<form>`.

Прекращение всплытия

Всплытие идёт прямо наверх. Обычно событие будет всплывать наверх и наверх, до элемента `<html>`, а затем до `document`, а иногда даже до `window`, вызывая все обработчики на своем пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для остановки всплытия нужно вызвать метод `event.stopPropagation()`.

Например, здесь при клике на кнопку обработчик `body.onclick` не сработает:

```
<body onclick="alert('сюда обработка не дойдёт')">
  <button onclick="event.stopPropagation()">Кликни меня</button>
</body>
```

Кликни меня

event.stopImmediatePropagation()

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены.

То есть, `stopPropagation` препятствует продвижению события дальше, но на текущем элементе все обработчики отработают.

Для того, чтобы полностью остановить обработку, современные браузеры поддерживают метод `event.stopImmediatePropagation()`. Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

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:

1. Мы делаем меню. Оно обрабатывает клики на своих элементах и делает для них `stopPropagation`. Вроде бы, всё работает.
2. Позже мы решили отслеживать все клики в окне, для какой-то своей функциональности, к примеру, для статистики — где вообще у нас кликают люди. Например, Яндекс.Метрика так делает, если включить соответствующую опцию.
3. Над областью, где клики убиваются `stopPropagation`, статистика работать не будет! Получилась «мёртвая зона».

Проблема в том, что `stopPropagation` убивает всякую возможность отследить событие сверху, а это бывает нужно для реализации чего-нибудь «эдакого», что к меню отношения совсем не имеет.

Погружение

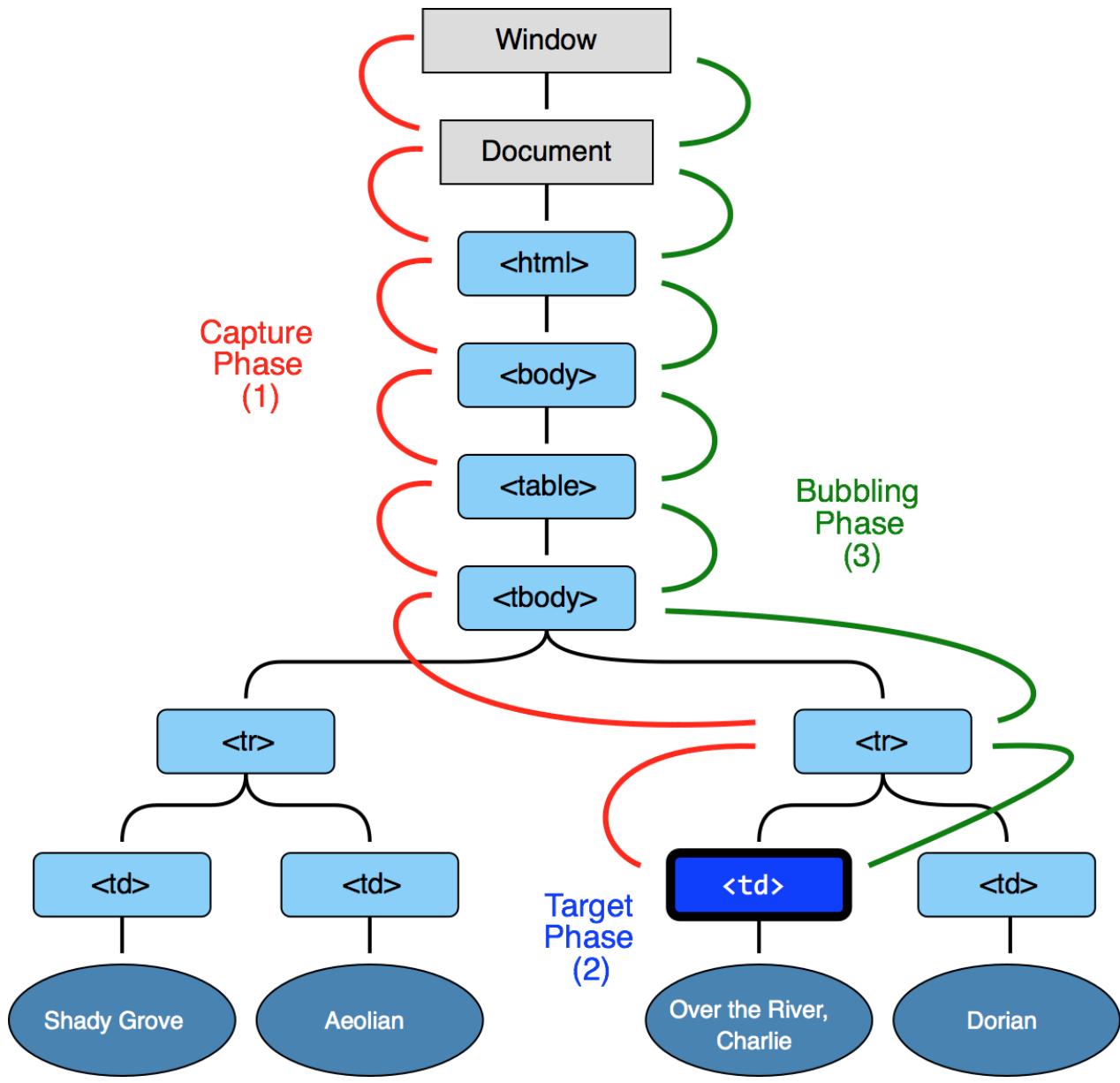
В современном стандарте, кроме «всплытия» событий, предусмотрено ещё и «погружение».

Оно гораздо менее востребовано, но иногда, очень редко, знание о нём может быть полезным.

Строго говоря, стандарт выделяет целых 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*).

В [стандарте DOM Events 3 ↗](#) это продемонстрировано так:



То есть, при клике на TD событие путешествует по цепочке родителей сначала вниз к элементу («погружается»), а потом наверх («всплывает»), по пути задействуя обработчики.

Ранее мы говорили только о всплытии, потому что другие стадии, как правило, не используются и проходят незаметно для нас.

Обработчики, добавленные через `on...-свойство`, ничего не знают о стадии перехвата, а начинают работать со всплытием.

Чтобы поймать событие на стадии перехвата, нужно использовать третий аргумент `addEventListener`:

- Если аргумент `true`, то событие будет перехвачено по дороге вниз.
- Если аргумент `false`, то событие будет поймано при всплытии.

Стадия цели, обозначенная на рисунке цифрой (2), особо не обрабатывается, так как обработчики, назначаемые обоими этими способами, срабатывают также на целевом элементе.

Есть события, которые не всплывают, но которые можно перехватить

Бывают события, которые можно поймать только на стадии перехвата, а на стадии всплытия — нельзя..

Например, таково событие фокусировки на элементе `onfocus`. Конечно, это большая редкость, такое исключение существует по историческим причинам.

Примеры

В примере ниже на `<form>`, `<div>`, `<p>` стоят те же обработчики, что и раньше, но на этот раз — на стадии погружения. Чтобы увидеть перехват в действии, кликните в нём на элементе `<p>`:

[Смотреть пример онлайн ↗](#)

Обработчики сработают в порядке «сверху-вниз»: FORM → DIV → P.

JS-код здесь такой:

```
var elems = document.querySelectorAll('form,div,p');

// на каждый элемент повесить обработчик на стадии перехвата
for (var i = 0; i < elems.length; i++) {
  elems[i].addEventListener("click", highlightThis, true);
}
```

Никто не мешает назначить обработчики для обеих стадий, вот так:

```
var elems = document.querySelectorAll('form,div,p');

for (var i = 0; i < elems.length; i++) {
  elems[i].addEventListener("click", highlightThis, true);
  elems[i].addEventListener("click", highlightThis, false);
}
```

Кликните по внутреннему элементу `<p>`, чтобы увидеть порядок прохода события:

[Смотреть пример онлайн ↗](#)

Должно быть FORM → DIV → P → P → DIV → FORM. Заметим, что элемент `<p>` участвует в обоих стадиях.

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` (=1, если погружение, =3, если всплытие).

Отличия IE8-

Чтобы было проще ориентироваться, я собрал отличия IE8-, которые имеют отношение ко всплытию, в одну секцию.

Их знание понадобится, если вы решите писать на чистом JS, без фреймворков и вам понадобится поддержка IE8-.

Нет свойства event.currentTarget

Обратим внимание, что при назначении обработчика через опсвойство у нас есть this, поэтому event.currentTarget, как правило, не нужно, а вот при назначении через attachEvent обработчик не получает this, так что текущий элемент, если нужен, можно будет взять лишь из замыкания.

Вместо event.target в IE8- используется event.srcElement

Если мы пишем обработчик, который будет поддерживать и IE8- и современные браузеры, то можно начать его так:

```
elem.onclick = function(event) {
  event = event || window.event;
  var target = event.target || event.srcElement;

  // ... теперь у нас есть объект события и target
  ...
}
```

Для остановки всплытия используется код event.cancelBubble=true.

Кросс-браузерно остановить всплытие можно так:

```
event.stopPropagation ? event.stopPropagation() : (event.cancelBubble=true);
```

Далее в учебнике мы будем использовать стандартные свойства и вызовы, поскольку добавление этих строк, обеспечивающих совместимость — достаточно простая и очевидная задача. Кроме того, никто не мешает подключить полифилл.

Ещё раз хотелось бы заметить — эти отличия нужно знать при написании JS-кода с поддержкой IE8- без фреймворков. Почти все JS-фреймворки обеспечивают кросс-браузерную поддержку target, currentTarget и stopPropagation().

Итого

Алгоритм:

- При наступлении события — элемент, на котором оно произошло, помечается как «целевой» (event.target).
- Далее событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(..., true).
- Далее событие двигается от event.target вверх к корню документа, по пути вызывая обработчики, поставленные через on* и addEventListener(..., false).

Каждый обработчик имеет доступ к свойствам события:

- `event.target` — самый глубокий элемент, на котором произошло событие.
- `event.currentTarget (=this)` — элемент, на котором в данный момент сработал обработчик (до которого «доплыло» событие).
- `event.eventPhase` — на какой фазе он сработал (погружение =1, всплытие = 3).

Любой обработчик может остановить событие вызовом `event.stopPropagation()`, но делать это не рекомендуется, так как в дальнейшем это событие может понадобиться, иногда для самых неожиданных вещей.

В современной разработке стадия погружения используется очень редко.

Этому есть две причины:

1. Историческая — так как IE лишь с версии 9 в полной мере поддерживает современный стандарт.
2. Разумная — когда происходит событие, то разумно дать возможность первому сработать обработчику на самом элементе, поскольку он наиболее конкретен. Код, который поставил обработчик именно на этот элемент, знает максимум деталей о том, что это за элемент, чем он занимается.

Далее имеет смысл передать обработку события родителю — он тоже понимает, что происходит, но уже менее детально, далее — выше, и так далее, до самого объекта `document`, обработчик на котором реализовывает самую общую функциональность уровня документа.

Делегирование событий

Всплытие событий позволяет реализовать один из самых важных приёмов разработки — **делегирование**.

Он заключается в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому — мы ставим один обработчик на их общего предка. Из него можно получить целевой элемент `event.target`, понять на каком именно потомке произошло событие и обработать его.

Пример «Ба Гуа»

Рассмотрим пример — [диаграмму «Ба Гуа»](#). Это таблица, отражающая древнюю китайскую философию.

Вот она:

Bagua Chart: Direction, Element, Color, Meaning

Northwest	North	Northeast
Metal	Water	Earth
Silver	Blue	Yellow
Elders	Change	Direction
West	Center	East
Metal	All	Wood
Gold	Purple	Blue
Youth	Harmony	Future
Southwest	South	Southeast
Earth	Fire	Wood
Brown	Orange	Green
Tranquility	Fame	Romance

Её HTML (схематично):

```
<table>
  <tr>
    <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
  </tr>
  <tr>
    <td>...<strong>Northwest</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>... еще 2 строки такого же вида...
  <tr>... еще 2 строки такого же вида...
</table>
```

В этой таблице всего 9 ячеек, но могло быть и 99, и даже 9999, не важно.

Наша задача — реализовать подсветку ячейки `<td>` при клике.

Вместо того, чтобы назначать обработчик для каждой ячейки, которых может быть очень много — мы повесим *единный обработчик* на элемент `<table>`.

Он будет использовать `event.target`, чтобы получить элемент, на котором произошло событие, и подсветить его.

Код будет таким:

```

var selectedTd;

table.onclick = function(event) {
  var target = event.target; // где был клик?

  if (target.tagName != 'TD') return; // не на TD? тогда не интересует

  highlight(target); // подсветить TD
};

function highlight(node) {
  if (selectedTd) {
    selectedTd.classList.remove('highlight');
  }
  selectedTd = node;
  selectedTd.classList.add('highlight');
}

```

Такому коду нет разницы, сколько ячеек в таблице. Обработчик всё равно один. Я могу добавлять, удалять `<td>` из таблицы, менять их количество — моя подсветка будет стablyно работать, так как обработчик стоит на `<table>`.

Однако, у текущей версии кода есть недостаток.

Клик может быть не на том теге, который нас интересует, а внутри него.

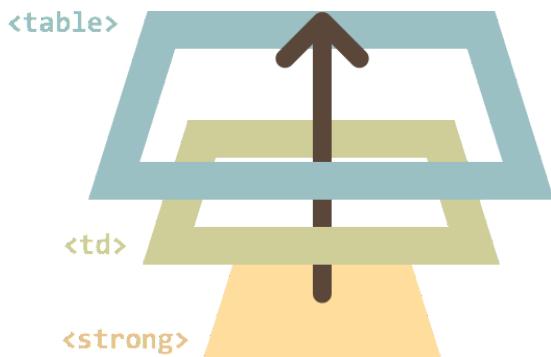
В нашем случае, если взглянуть на HTML таблицы внимательно, видно, что ячейка содержит вложенные теги, например ``:

```

<td>
  <strong>Northwest</strong>
  ...Metal..Silver..Elders...
</td>

```

Естественно, клик может произойти внутри `<td>`, на элементе ``. Такой клик будет пойман единственным обработчиком, но `target` у него будет не `<td>`, а ``:



Внутри обработчика `table.onclick` мы должны по `event.target` разобраться, в каком именно `<td>` был клик.

Для этого мы, используя ссылку `parentNode`, будем идти вверх по иерархии родителей от `event.target` и выше и проверять:

- Если нашли `<td>`, значит это то что нужно.

- Если дошли до элемента `table` и при этом `<td>` не найден, то наверное клик был вне `<td>`, например на элементе заголовка таблицы.

Улучшенный обработчик `table.onclick` с циклом `while`, который этот делает:

```
table.onclick = function(event) {
  var target = event.target;

  // цикл двигается вверх от target к родителям до table
  while (target != table) {
    if (target.tagName == 'TD') {
      // нашли элемент, который нас интересует!
      highlight(target);
      return;
    }
    target = target.parentNode;
  }

  // возможна ситуация, когда клик был вне <td>
  // если цикл дошёл до table и ничего не нашёл,
  // то обработчик просто заканчивает работу
}
```

На заметку:

Кстати, в проверке `while` можно было использовать `this` вместо `table`:

```
while (target != this) {
  // ...
}
```

Это тоже будет работать, так как в обработчике `table.onclick` значением `this` является текущий элемент, то есть `table`.

Можно для этого использовать и метод `closest`, при поддержке браузером:

```
table.onclick = function(event) {
  var target = event.target;

  var td = target.closest('td');
  if (!td) return; // клик вне <td>, не интересует

  // если клик на td, но вне этой таблицы (возможно при вложенных таблицах)
  // то не интересует
  if (!table.contains(td)) return;

  // нашли элемент, который нас интересует!
  highlight(td);
}
```

Применение делегирования: действия в разметке

Обычно делегирование — это средство оптимизации интерфейса. Мы используем один

обработчик для схожих действий на однотипных элементах.

Выше мы это делали для обработки кликов на `<td>`.

Но делегирование позволяет использовать обработчик и для абсолютно разных действий.

Например, нам нужно сделать меню с разными кнопками: «Сохранить», «Загрузить», «Поиск» и т.д. И есть объект с соответствующими методами: `save`, `load`, `search` и т.п...

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

Но более изящно решить задачу можно путем добавления одного обработчика на всё меню, а для каждой кнопки в специальном атрибуте, который мы назовем `data-action` (можно придумать любое название, но `data-*` является валидным в HTML5), укажем, что она должна вызывать:

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

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

```
<div id="menu">
  <button data-action="save">Сохранить</button>
  <button data-action="load">Загрузить</button>
  <button data-action="search">Поиск</button>
</div>

<script>
  function Menu(elem) {
    this.save = function() {
      alert( 'сохраняю' );
    };
    this.load = function() {
      alert( 'загружаю' );
    };
    this.search = function() {
      alert( 'ищу' );
    };

    var self = this;

    elem.onclick = function(e) {
      var target = e.target;
      var action = target.getAttribute('data-action');
      if (action) {
        self[action]();
      }
    };
  };

  new Menu(menu);
</script>
```

Сохранить Загрузить Поиск

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. Также можно использовать классы `.action-save`, `.action-load` вместо атрибута `data-action`.

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 `event.target` или один из его родителей в контейнере (`this`) — интересующий нас элемент — обработать его.

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. Нельзя, чтобы какой-то промежуточный обработчик вызвал `event.stopPropagation()` до того, как событие доплынет до нужного элемента.

- 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 настолько пустяковая, её даже не стоит принимать во внимание.

✓ Задачи

Скрытие сообщения с помощью делегирования

важность: 5

Дан список сообщений. Добавьте каждому сообщению кнопку для его удаления.

Используйте делегирование событий. Один обработчик для всего.

В результате, должно работать вот так(кликните на крестик):

Лошадь

[x]

Домашняя лошадь — животное семейства непарнокопытных, одомашненный и единственный сохранившийся подвид дикой лошади, вымершей в дикой природе, за исключением небольшой популяции лошади Пржевальского.

Осёл

[x]

Домашний осёл или ишак — одомашненный подвид дикого осла, сыгравший важную историческую роль в развитии хозяйства и культуры человека. Все одомашненные ослы относятся к африканским ослам.

Корова, а также пара слов о диком быке, о волах и о тёлках.

[x]

Корова — самка домашнего быка, одомашненного подвида дикого быка, парнокопытного жвачного животного семейства полорогих. Самцы вида называются быками, молодняк — телятами, кастрированные самцы — волами. Молодых (до первой стельности) самок называют тёлками.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Раскрывающееся дерево

важность: 5

Создайте дерево, которое по клику на заголовок скрывает-показывает детей:

- Животные
 - Млекопитающие
 - Коровы
 - Ослы
 - Собаки
 - Тигры
 - Другие
 - Змеи
 - Птицы
 - Ящерицы
- Рыбы
 - Аквариумные
 - Гуппи
 - Скалярии
 - Морские
 - Морская форель

Требования:

- Использовать делегирование.
- Клик вне текста заголовка (на пустом месте) ничего делать не должен.
- При наведении на заголовок — он становится жирным, реализовать через CSS.

P.S. При необходимости HTML/CSS дерева можно изменить.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Сортировка таблицы

важность: 4

Сделать сортировку таблицы при клике на заголовок.

Демо:

Возраст Имя

5	Вася
2	Петя
12	Женя
9	Маша
1	Илья

Требования:

- Использовать делегирование.
- Код не должен меняться при увеличении количества столбцов или строк.

P.S. Обратите внимание, тип столбца задан атрибутом у заголовка. Это необходимо, ведь числа сортируются иначе чем строки. Соответственно, код это может использовать.

P.P.S. Вам помогут дополнительные [навигационные ссылки по таблицам](#).

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Приём проектирования «поведение»

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

Описание

Приём проектирования «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, добавим «поведение», которое всем элементам, у которых стоит атрибут `data-counter`, будет при клике увеличивать значение на 1:

```
Счётчик:  
<button data-counter>1</button>  
Ещё счётчик:  
<button data-counter>2</button>  
  
<script>  
  document.onclick = function(event) {  
    if (!event.target.hasAttribute('data-counter')) return;  
  
    var counter = event.target;  
  
    counter.innerHTML++;  
  };  
</script>
```

Счётчик: 1 Ещё счётчик: 2

Если запустить HTML-код выше, то при клике на каждую кнопку — её значение будет увеличиваться.

Конечно, нам важны не счётчики, а общий подход, который они демонстрируют.

Элементов `data-counter` может быть сколько угодно. Новые могут добавляться в HTML в любой момент. При помощи делегирования мы, фактически, добавили новый «псевдо-стандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Ещё пример

Добавим ещё поведение.

Сделаем так, что при клике на элемент с атрибутом `data-toggle-id` будет скрываться/показываться элемент с заданным `id`:

```
<button data-toggle-id="subscribe-mail">  
  Показать форму подписки  
</button>  
  
<form id="subscribe-mail" hidden>  
  Ваша почта: <input type="email">  
</form>  
  
<script>  
  document.onclick = function(event) {  
    var target = event.target;  
  
    var id = target.getAttribute('data-toggle-id');  
    if (!id) return;  
  
    var elem = document.getElementById(id);  
  
    elem.hidden = !elem.hidden;  
  };  
</script>
```

Показать форму подписки

Еще раз заметим, что мы сделали. Теперь для того, чтобы добавить скрытие-раскрытие любому элементу — даже не надо знать JavaScript, можно просто написать атрибут `data-toggle-id`.

Это бывает очень удобно — не нужно писать JavaScript-код для каждого элемента, который должен служить такой кнопкой. Просто используем поведение.

Обратите внимание: обработчик поставлен на `document`, клик на любом элементе страницы пройдёт через него, так что поведение определено глобально.

Не только атрибут

Для своих целей мы можем использовать в HTML любые атрибуты, но стандарт рекомендует для своих целей называть атрибуты `data-*`.

В обработчике `document.onclick` мы могли бы проверять не атрибут, а класс или что-то ещё, но с атрибутом — проще и понятнее всего.

Также для добавления обработчиков на `document` рекомендуется использовать `addEventListener`, чтобы можно было добавить более одного обработчика для типа события.

Итого

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

Здесь мы рассмотрели базовый пример, который можно как угодно модифицировать и масштабировать. Важно не переусердствовать.

Приём разработки «поведение» рекомендуется использовать для расширения возможностей разметки, как альтернативу мини-фрагментам JavaScript.

Задачи

Поведение «подсказка»

важность: 5

При наведении мыши на элемент, на нём возникает событие `mouseover`, при удалении мыши с элемента — событие `mouseout`.

Зная это, напишите JS-код, который будет делать так, что при наведении на элемент, если у него есть атрибут `data-tooltip` — над ним будет показываться подсказка с содержимым этого атрибута.

Например, две кнопки:

```
<button data-tooltip="подсказка длиннее, чем элемент">Короткая кнопка</button>
<button data-tooltip="HTML<br>подсказка">Ещё кнопка</button>
```

Результат в ифрейме с документом:

ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя

ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя ЛяЛяЛя

Короткая кнопка

Ещё кнопка

Прокрутите страницу, чтобы ссылки были вверху и проверьте, правильно ли показываются подсказки.

В этой задаче можно полагать, что в элементе с атрибутом `data-tooltip` — только текст, то есть нет подэлементов.

Детали оформления:

- Подсказка должна появляться при наведении на элемент, по центру и на небольшом расстоянии сверху. При уходе курсора с элемента — исчезать.
- Текст подсказки брать из значения атрибута `data-tooltip`. Это может быть произвольный HTML.
- Оформление подсказки должно задаваться CSS.
- Подсказка не должна вылезать за границы экрана, в том числе если страница частично прокручена. Если нельзя показать сверху — показывать снизу элемента.

Важно: нужно использовать приём разработки «поведение», то есть поставить обработчик (точнее два) на `document`, а не на каждый элемент.

Плюс этого подхода — динамически добавленные в DOM позже элементы автоматически получат этот функционал.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

- Клик по ссылке инициирует переход на новый URL.
- Нажатие на кнопку «отправить» в форме — отсылку ее на сервер.
- Двойной клик на тексте — инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера нам не нужно. К счастью, его можно отменить.

Отмена действия браузера

Есть два способа отменить действие браузера:

- **Основной способ — это воспользоваться объектом события. Для отмены действия браузера существует стандартный метод event.preventDefault().**
- Если же обработчик назначен через оп событие (не через addEventListener), то можно просто вернуть false из обработчика.

В следующем примере при клике по ссылке переход не произойдет:

```
<a href="/" onclick="return false">Нажми здесь</a>  
или  
<a href="/" onclick="event.preventDefault()">здесь</a>
```

Нажми здесь или здесь

⚠ Возвращать true не нужно

Обычно значение, которое возвращает обработчик события, игнорируется.

Единственное исключение — это return false из обработчика, назначенного через оп событие.

Иногда в коде начинающих разработчиков можно увидеть return других значений. Но они не нужны и никак не обрабатываются.

Пример: меню

Рассмотрим задачу, когда нужно создать меню для сайта, например такое:

```
<ul id="menu" class="menu">  
  <li><a href="/php">PHP</a></li>  
  <li><a href="/html">HTML</a></li>  
  <li><a href="/javascript">JavaScript</a></li>  
  <li><a href="/flash">Flash</a></li>  
</ul>
```

Данный пример при помощи CSS может выводиться так:

PHP

HTML

JavaScript

Flash

HTML-разметка сделана так, что все элементы меню являются не кнопками, а ссылками, то есть тегами `<a>`.

Это потому, что некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Да, мы можем использовать и `<button>` и ``, но если правый клик не работает — это их огорчает. Кроме того, если на сайт зайдёт поисковик, то по ссылке из `` он перейдёт, а выполнить сложный JavaScript и получить результат — вряд ли захочет.

Поэтому в разметке мы используем именно `<a>`, но обычно клик будет обрабатываться полностью в JavaScript, а стандартное действие браузера (переход по ссылке) — отменяться.

Например, вот так:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  var href = event.target.getAttribute('href');
  alert(href); // может быть подгрузка с сервера, генерация интерфейса и т.п.

  return false; // отменить переход по url
};
```

В конце `return false`, иначе браузер перейдёт по адресу из `href`.

Так как мы применили делегирование, то меню может увеличиваться, можно добавить вложенные списки `ul/li`, стилизовать их при помощи CSS — обработчик не потребует изменений.

Другие действия браузера

Действий браузера по умолчанию достаточно много.

Вот некоторые примеры событий, которые вызывают действие браузера:

- `mousedown` — нажатие кнопкой мыши в то время как курсор находится на тексте начинает его выделение.
- `click` на `<input type="checkbox">` — ставит или убирает галочку.
- `submit` — при нажатии на `<input type="submit">` в форме данные отправляются на сервер.
- `wheel` — движение колёсика мыши инициирует прокрутку.
- `keydown` — при нажатии клавиши в поле ввода появляется символ.
- `contextmenu` — при правом клике показывается контекстное меню браузера.
- ...

Все эти действия можно отменить, если мы хотим обработать событие исключительно при помощи JavaScript.

⚠ События могут быть связаны между собой

Некоторые события естественным образом вытекают друг из друга.

Например, нажатие мышкой `mousedown` на поле ввода `<input>` приводит к фокусировке внутрь него. Если отменить действие `mousedown`, то и фокуса не будет.

Попробуйте нажать мышкой на первый `<input>` — произойдёт событие `onfocus`. Это обычная ситуация.

Но если нажать на второй, то фокусировки не произойдёт.

```
<input value="Фокус работает" onfocus="this.value=' '">
<input onmousedown="return false" onfocus="this.value=' '" value="Кликни меня">
```

Фокус работает Кликни меня

Это потому, что отменено стандартное действие при `onmousedown`.

...С другой стороны, во второй `<input>` можно перейти с первого нажатием клавиши `Tab`, и тогда фокусировка сработает. То есть, дело здесь именно в `onmousedown="return false"`.

Особенности IE8-

В IE8- для отмены действия по умолчанию нужно назначить свойство `event.returnValue = false`.

Кросбраузерный код для отмены действия по умолчанию:

```
element.onclick = function(event) {
  event = event || window.event;

  if (event.preventDefault) { // если метод существует
    event.preventDefault(); // то вызвать его
  } else { // иначе вариант IE8-:
    event.returnValue = false;
  }
}
```

Можно записать в одну строку:

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

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()` (IE8: `event.returnValue=false`), второй — `return false` из обработчика. Второй способ работает только если обработчик назначен через `on событие`.

✓ Задачи

Почему не работает `return false`?

важность: 3

Почему в этом документе `return false` не работает?

```
<script>  
  function handler() {  
    alert( "...");  
    return false;  
  }  
</script>  
w3.org</a>
```

[w3.org](#)

По замыслу, переход на `w3.org` при клике должен отменяться. Однако, на самом деле он происходит.

В чём дело и как поправить?

[К решению](#)

Поймайте переход по ссылке

важность: 5

Сделайте так, чтобы при клике на ссылки внутри элемента `#contents` пользователю выводился вопрос о том, действительно ли он хочет покинуть страницу и если он не хочет, то прерывать переход по ссылке.

Так это должно работать:

```
#contents
[ Как насчет почитать Википедию, или посетить W3.org и узнать про современные стандарты? ]
```

Детали:

- Содержимое `#contents` может быть загружено динамически и присвоено при помощи `innerHTML`. Так что найти все ссылки и поставить на них обработчики нельзя. Используйте делегирование.
- Содержимое может содержать вложенные теги, в том числе внутри ссылок, например, `<a href.."><i>...</i>`.

[Открыть песочницу для задачи.](#) ↗

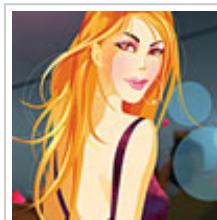
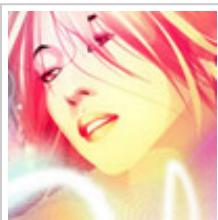
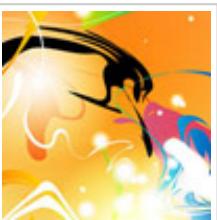
[К решению](#)

Галерея изображений

важность: 5

Создайте галерею изображений, в которой основное изображение изменяется при клике на уменьшенный вариант.

Результат должен выглядеть так:



Для обработки событий используйте делегирование, т.е. не более одного обработчика.

P.S. Обратите внимание — клик может быть как на маленьком изображении IMG, так и на A вне него. При этом event.target будет, соответственно, либо IMG, либо A.

Дополнительно:

- Если получится — сделайте предзагрузку больших изображений, чтобы при клике они появлялись сразу.
- Всё ли в порядке с семантической вёрсткой в HTML исходного документа? Если нет — поправьте, чтобы было, как нужно.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Генерация событий на элементах

Можно не только назначать обработчики на события, но и генерировать их самому.

Мы будем использовать это позже для реализации компонентной архитектуры, при которой элемент, представляющий собой, к примеру, меню, генерирует события, к этому меню относящиеся — `select` (выбран пункт меню) или `open` (меню раскрыто), и другие.

Кроме того, события можно генерировать для целей автоматического тестирования.

Конструктор Event

Вначале рассмотрим современный способ генерации событий, по стандарту [DOM 4 ↗](#). Он поддерживается всеми браузерами, кроме IE11-. А далее рассмотрим устаревшие варианты, поддерживаемые IE.

Объект события в нём создаётся при помощи встроенного конструктора `Event ↗`.

Синтаксис:

```
var event = new Event(тип события[, флаги]);
```

Где:

- *Тип события* — может быть как своим, так и встроенным, к примеру "click".
- *Флаги* — объект вида { `bubbles`: true/false, `cancelable`: true/false }, где свойство `bubbles` указывает, всплывает ли событие, а `cancelable` — можно ли отменить действие по умолчанию.

Флаги по умолчанию: {`bubbles`: false, `cancelable`: false}.

Метод dispatchEvent

Затем, чтобы инициировать событие, запускается `elem.dispatchEvent(event)`.

При этом событие срабатывает наравне с браузерными, то есть обычные браузерные обработчики на него отреагируют. Если при создании указан флаг `bubbles`, то оно будет всплывать.

При просмотре примера ниже кнопка обработчик `onclick` на кнопке сработает сам по себе, событие генерируется скриптом:

```
<button id="elem" onclick="alert('Клик');">Автоклик</button>

<script>
  var event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

Отмена действия по умолчанию

На сгенерированном событии, как и на встроенном браузерном, обработчик может вызвать метод

`event.preventDefault()`. Тогда `dispatchEvent` возвратит `false`.

Остановимся здесь подробнее. Обычно `preventDefault()` вызов предотвращает действие браузера. В случае, если событие придумано нами, имеет нестандартное имя — никакого действия браузера, конечно, нет.

Но код, который генерирует событие, может предусматривать какие-то ещё действия после `dispatchEvent`.

Вызов `event.preventDefault()` является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия продолжать не надо.

В примере ниже есть функция `hide()`, которая при вызове генерирует событие `hide` на элементе `#rabbit`, уведомляя всех интересующихся, что кролик собирается спрятаться.

Любой обработчик может узнать об этом, подписавшись на событие через `rabbit.addEventListener('hide', ...)` и, при желании, отменить действие по умолчанию через `event.preventDefault()`. Тогда кролик не исчезнет:

```
<pre id="rabbit">
  \_ /|
  \|_|/
  /..\
  =\_\_Y\_/=_
  {>o<}
</pre>

<script>

  function hide() {
    var event = new Event("hide", {
      cancelable: true
    });
    if (!rabbit.dispatchEvent(event)) {
      alert( 'действие отменено обработчиком' );
    } else {
      rabbit.hidden = true;
    }
  }

  rabbit.addEventListener('hide', function(event) {
    if (confirm("Вызвать preventDefault?")) {
      event.preventDefault();
    }
  });

  // прячемся через 2 секунды
  setTimeout(hide, 2000);

</script>
```

Как отличить реальное нажатие от скриптового?

В целях безопасности иногда хорошо бы знать — инициировано ли действие посетителем или это кликнул скрипт.

Единственный способ, которым код может отличить реальное нажатие от программного, является проверка свойства `event.isTrusted`.

Оно на момент написания статьи поддерживается IE и Firefox и равно `true`, если посетитель кликнул сам, и всегда `false` — если событие инициировал скрипт.

Другие свойства событий

При создании события браузер автоматически ставит следующие свойства:

- `isTrusted: false` — означает, что событие сгенерировано скриптом, это свойство изменить невозможно.
- `target: null` — это свойство ставится автоматически позже при `dispatchEvent`.
- `type: тип события` — первый аргумент `new Event`.
- `bubbles, cancelable` — по второму аргументу `new Event`.

Другие свойства события, если они нужны, например координаты для события мыши — можно присвоить в объект события позже, например:

```
var event = new Event("click", {bubbles: true, cancelable: false});
event.clientX = 100;
event.clientY = 100;
```

Пример со всплытием

Сгенерируем совершенно новое событие "hello" и поймаем его на `document`.

Всё, что для этого нужно — это флаг `bubbles`:

```
<h1 id="elem">Привет от скрипта!</h1>

<script>
  document.addEventListener("hello", function(event) { // (1)
    alert("Привет");
    event.preventDefault(); // (2)
  }, false);

  var event = new Event("hello", {bubbles: true, cancelable: true}); // (3)
  if (elem.dispatchEvent(event) === false) {
    alert('Событие было отменено preventDefault');
  }
</script>
```

Обратите внимание:

1. Обработчик события `onclick` стоит на `document`. Мы его поимаем на всплытии.
2. Вызов `event.preventDefault()` приведёт к тому, что `dispatchEvent` вернёт `false`.
3. Чтобы событие всплывало и его можно было отменить, указан второй аргумент `new Event`.

Никакой разницы между встроенными событиями (`click`) и своими (`hello`) здесь нет, их можно сгенерировать и запустить совершенно одинаково.

Конструкторы `MouseEvent`, `KeyboardEvent` и другие

Для некоторых конкретных типов событий есть свои, специфические, конструкторы.

Вот список конструкторов для различных событий интерфейса которые можно найти в спецификации [UI Event ↗](#):

- `UIEvent`
- `FocusEvent`
- `MouseEvent`
- `WheelEvent`
- `KeyboardEvent`
- `CompositionEvent`

Вместо `new Event("click")` можно вызвать `new MouseEvent("click")`.

Специфический конструктор позволяет указать стандартные свойства для данного типа события.

Например, `clientX/clientY` для события мыши:

```
var e = new MouseEvent("click", {  
  bubbles: true,  
  cancelable: true,  
  clientX: 100,  
  clientY: 100  
});
```

```
alert( e.clientX ); // 100
```

Это нельзя было бы сделать с обычным конструктором `Event`:

```
var e = new Event("click", {  
  bubbles: true,  
  cancelable: true,  
  clientX: 100,  
  clientY: 100  
});
```

```
alert( e.clientX ); // undefined, свойство не присвоено!
```

Обычный конструктор `Event` не знает про «мышиные» свойства, поэтому их игнорирует.

Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после конструктора. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип.

Полный список свойств по типам событий вы найдёте в спецификации, например для MouseEvent: [MouseEvent Constructor](#).

Свои события

Для генерации своих, нестандартных, событий, хоть и можно использовать конструктор Event, но существует и специфический конструктор [CustomEvent](#).

Технически, он абсолютно идентичен Event, кроме небольшой детали: у второго аргумента-объекта есть дополнительное свойство `detail`, в котором можно указывать информацию для передачи в событие.

Например:

```
<h1 id="elem">Привет для Васи!</h1>

<script>
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  }, false);

  var event = new CustomEvent("hello", {
    detail: { name: "Вася" }
  });

  elem.dispatchEvent(event);
</script>
```

Надо сказать, что никто не мешает и в обычное Event записать любые свойства. Но CustomEvent более явно говорит, что событие не встроенное, а своё, и выделяет отдельно «информационное» поле `detail`, в которое можно записать что угодно без конфликта со стандартными свойствами объекта.

Старое API для IE9+

Способ генерации событий, описанный выше, не поддерживается в IE11-, там нужен другой, более старый способ, описанный в стандарте [DOM 3 Events](#).

В нём была предусмотрена [иерархия событий](#), с различными методами инициализации.

Она поддерживается как современными браузерами, так и IE9+. Там используется немного другой синтаксис, но по возможностям — всё то же самое, что и в современном стандарте.

Можно использовать этот немного устаревший способ, если нужно поддерживать IE9+. Далее мы на его основе создадим полифилл.

Объект события создаётся вызовом `document.createEvent`:

```
var event = document.createEvent(eventInterface);
```

Аргументы:

- `eventInterface` — это тип события, например `MouseEvent`, `FocusEvent`, `KeyboardEvent`. В [секции 5 DOM 3 Events](#) есть подробный список, какое событие к какому интерфейсу относится.

На практике можно всегда использовать самый общий интерфейс:
`document.createEvent("Event")`.

Далее событие нужно инициализовать:

```
event.initEvent(type, boolean bubbles, boolean cancelable);
```

Аргументы:

- `type` — тип события, например "click".
- `bubbles` — всплывает ли событие.
- `cancelable` — можно ли отменить событие.

Эти два кода аналогичны:

```
// современный стандарт
var event = new Event("click", {
  bubbles: true,
  cancelable: true
});

// старый стандарт
var event = document.createEvent("Event");
event.initEvent("click", true, true);
```

Единственная разница — старый стандарт поддерживается IE9+.

Этот пример с событием `hello` будет работать во всех браузерах, кроме IE8-:

```
<h1 id="elem">Привет от скрипта!</h1>

<script>
  document.addEventListener("hello", function(event) {
    alert( "Привет" );
    event.preventDefault();
  }, false);

  var event = document.createEvent("Event");
  event.initEvent("hello", true, true);

  if (elem.dispatchEvent(event) === false) {
    alert( 'Событие было отменено preventDefault' );
  }
</script>
```

initMouseEvent, initKeyboardEvent и другие...

У конкретных типов событий, например MouseEvent, KeyboardEvent, есть методы, которые позволяют указать стандартные свойства.

Они называются по аналогии: initMouseEvent, initKeyboardEvent.

Их можно использовать вместо базового initEvent, если хочется, чтобы свойства событий соответствовали встроенным браузерным.

Выглядят они немного страшновато, например (взято из [спецификации](#)):

```
void initMouseEvent(  
    DOMString typeArg, // тип  
    boolean bubblesArg, // всплывает?  
    boolean cancelableArg, // можно отменить?  
    AbstractView ? viewArg, // объект window, null означает текущее окно  
    long detailArg, // свойство detail и другие...  
    long screenXArg,  
    long screenYArg,  
    long clientXArg,  
    long clientYArg,  
    boolean ctrlKeyArg,  
    boolean altKeyArg,  
    boolean shiftKeyArg,  
    boolean metaKeyArg,  
    unsigned short buttonArg,  
    EventTarget ? relatedTargetArg);  
};
```

Для инициализации мышиного события нужно обязательно указать *все* аргументы, например:

```
<button id="elem">Автоклик</button>  
  
<script>  
elem.onclick = function(e) {  
    alert( 'Клик на координатах ' + e.clientX + ':' + e.clientY );  
};  
  
var event = document.createEvent("MouseEvent");  
event.initMouseEvent("click", true, true, null, 0, 0, 0, 100, 100, true, true, true, null, 1, null);  
elem.dispatchEvent(event);  
</script>
```

Браузер, по стандарту, может сгенерировать отсутствующие свойства самостоятельно, например pageX, но это нужно проверять в конкретных случаях, иногда это не работает или работает некорректно, так что лучше указать все.

Полифилл CustomEvent

Для поддержки CustomEvent в IE9+ можно сделать небольшой полифилл:

```

try {
  new CustomEvent("IE has CustomEvent, but doesn't support constructor");
} catch (e) {

  window.CustomEvent = function(event, params) {
    var evt;
    params = params || {
      bubbles: false,
      cancelable: false,
      detail: undefined
    };
    evt = document.createEvent("CustomEvent");
    evt.initCustomEvent(event, params.bubbles, params.cancelable, params.detail);
    return evt;
  };

  CustomEvent.prototype = Object.create(window.Event.prototype);
}

```

Здесь мы сначала проверяем — в IE9-11 есть CustomEvent, но его нельзя создать через new, будет ошибка. В этом случае заменяем браузерную реализацию на свою, совместимую.

Антистандарт: IE8-

В совсем старом IE были «свои» методы document.createEventObject() и elem.fireEvent().

Пример с ними для IE8:

```

<button id="elem">Автоклик</button>

<script>
  document.body.onclick = function() {
    alert( "Клик, event.type=" + event.type );
    return false;
  };

  var event = document.createEventObject();
  if (!elem.fireEvent("onclick", event)) {
    alert( 'Событие было отменено' );
  }
</script>

```

При помощи fireEvent можно сгенерировать только встроенные события.

Если указать "hello" вместо "onclick" в примере выше — будет ошибка.

Параметры bubbles и cancelable настраивать нельзя, браузер использует стандартные для данного типа событий.

Существуют полифиллы для генерации произвольных событий и для IE8-, но они, по сути, полностью подменяют встроенную систему обработки событий браузером. И кода это требует тоже достаточно много.

Альтернатива — фреймворк, например jQuery, который также реализует свою мощную систему работы с событиями, доступную через методы jQuery.

Итого

- Все браузеры, кроме IE9-11, позволяют генерировать любые события, следуя стандарту DOM4.
- В IE9+ поддерживается более старый стандарт, можно легко сделать полифилл, например для CustomEvent он рассмотрен в этой главе.
- IE8- может генерировать только встроенные события.

Несмотря на техническую возможность генерировать встроенные браузерные события типа `click` или `keydown` — пользоваться ей стоит с большой осторожностью.

В 98% случаев, когда разработчик начинающего или среднего уровня хочет сгенерировать *встроенное* событие — это вызвано «кривой» архитектурой кода, и взаимодействие нужно на уровне выше.

Как правило события имеет смысл генерировать:

- Либо как явный и грубый хак, чтобы заставить работать сторонние библиотеки, в которых не предусмотрены другие средства взаимодействия.
- Либо для автоматического тестирования, чтобы скриптом «нажать на кнопку» и посмотреть, произошло ли нужное действие.
- Либо при создании своих «элементов интерфейса». Например, никто не мешает при помощи JavaScript создать из `<div class="calendar">` красивый календарь и генерировать на нём событие `change` при выборе даты. Эту тему мы разовьём позже.

События в деталях

В этом разделе мы разбираем конкретные события и особенности работы с ними.

Вы можете читать его в любом порядке или кратко просмотреть его и вернуться к конкретным событиям, когда они понадобятся.

Мышь: клики, кнопка, координаты

В этой главе мы глубже разберёмся 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.

Каждое событие обрабатывается независимо.

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

Получение информации о кнопке: which

При обработке событий, связанных с кликами мыши, бывает важно знать, какая кнопка нажата.

Для получения кнопки мыши в объекте event есть свойство which.

На практике оно используется редко, т.к. обычно обработчик вешается либо onclick — только на левую кнопку мыши, либо oncontextmenu — только на правую.

возможны следующие значения:

- `event.which == 1` – левая кнопка
- `event.which == 2` – средняя кнопка
- `event.which == 3` – правая кнопка

Это свойство не поддерживается IE8-, но его можно получить способом, описанным в конце главы.

Правый клик: `oncontextmenu`

Это событие срабатывает при клике правой кнопкой мыши:

```
<div>Правый клик на этой кнопке выведет "Клик".</div>
<button oncontextmenu="alert('Клик!');">Правый клик сюда</button>
```

Правый клик на этой кнопке выведет "Клик".

Правый клик сюда

При клике на кнопку выше после обработчика `oncontextmenu` будет показано обычное контекстное меню, которое браузер всегда показывает при клике правой кнопкой. Это является его действием по умолчанию.

Если мы не хотим, чтобы показывалось встроенное меню, например потому что показываем своё, специфичное для нашего приложения, то можно отменить действие по умолчанию.

В примере ниже встроенное меню показано 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 Alt+Shift+Клик:

```
<button>Alt+Shift+Кликни меня!</button>

<script>
  document.body.children[0].onclick = function(e) {
    if (!e.altKey || !e.shiftKey) return;
    alert('Ура!');
  }
</script>
```

Alt+Shift+Кликни меня!

⚠ Внимание: на Mac вместо Ctrl используется Cmd

На компьютерах под управлением Windows и Linux есть специальные клавиши `Alt`, `Shift` и `Ctrl`. На Mac есть ещё одна специальная клавиша: `Cmd`, которой соответствует свойство `metaKey`.

В большинстве случаев там, где под Windows/Linux используется `Ctrl`, на Mac используется `Cmd`. Там, где пользователь Windows нажимает `Ctrl+Enter` или `Ctrl+A`, пользователь Mac нажмёт `Cmd+Enter` или `Cmd+A`, и так далее, почти всегда `Cmd` вместо `Ctrl`.

Поэтому, если мы хотим поддерживать сочетание `Ctrl+click` или другие подобные, то под Mac имеет смысл использовать `Cmd+click`. Пользователям Mac это будет гораздо комфорльнее.

Более того, даже если бы мы хотели бы заставить пользователей Mac использовать именно `Ctrl+click` — это было бы затруднительно. Дело в том, что обычный клик с зажатым `Ctrl` под Mac работает как *правый клик* и генерирует событие `oncontextmenu`, а вовсе не `onclick`, как под Windows/Linux.

Решение — чтобы пользователи обоих операционных систем работали с комфортом, в паре с `ctrlKey` нужно обязательно использовать `metaKey`.

В JS-коде это означает, что для удобства пользователей Mac нужно проверять `if (event.ctrlKey || event.metaKey)`.

Координаты в окне: clientX/Y

Все мышиные события предоставляют текущие координаты курсора в двух видах: относительно окна и относительно документа.

Пара свойств `clientX/clientY` содержит координаты курсора относительно текущего окна.

При этом, например, если ваше окно размером 500x500, а мышь находится в центре, тогда и `clientX` и `clientY` будут равны 250.

Можно как угодно прокручивать страницу, но если не двигать при этом мышь, то координаты курсора `clientX/clientY` не изменятся, потому что они считаются относительно окна, а не документа.

В той же системе координат работает метод `element.getBoundingClientRect()`, возвращающий координаты элемента, а также `position:fixed`.

Относительно документа: pageX/Y

Координаты курсора относительно документа находятся в свойствах `pageX/pageY`.

Так как эти координаты — относительно левого-верхнего узла документа, а не окна, то они учитывают прокрутку. Если прокрутить страницу, а мышь не трогать, то координаты курсора `pageX/pageY` изменятся на величину прокрутки, они привязаны к конкретной точке в документе.

В IE8- этих свойств нет, но можно получить их способом, описанным в конце главы.

В той же системе координат работает `position:absolute`, если элемент позиционируется относительно документа.

Устарели: x, y, layerX, layerY

Некоторые браузеры поддерживают свойства `event.x/y`, `event.layerX/layerY`.

Эти свойства устарели, они нестандартные и не добавляют ничего к описанным выше. Использовать их не стоит.

Особенности IE8-

Двойной клик

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 второй раз, без события)
- `mouseup+dblclick` (отжал).

Поэтому отловить двойной клик в IE8-, отслеживая только `click`, нельзя, ведь при втором нажатии

его нет. Нужно именно событие `buttonclick`.

Свойство which/button

В старых IE8- не поддерживалось свойство `which`, а вместо него использовалось свойство `button`, которое является 3-х битным числом, в котором каждому биту соответствует кнопка мыши. Бит установлен в 1, только если соответствующая кнопка нажата.

Чтобы его расшифровать — нужна [побитовая операция &](#) («битовое И»):

- `!(button & 1) == true` (1й бит установлен), если нажата левая кнопка,
- `!(button & 2) == true` (2й бит установлен), если нажата правая кнопка,
- `!(button & 4) == true` (3й бит установлен), если нажата средняя кнопка.

Что интересно, при этом мы можем узнать, были ли две кнопки нажаты одновременно, в то время как стандартный `which` такой возможности не даёт. Так что, в некотором смысле, свойство `button` — более мощное.

Можно легко сделать функцию, которая будет ставить свойство `which` из `button`, если его нет:

```
function fixWhich(e) {
  if (!e.which && e.button) { // если which нет, но есть button... (IE8-)
    if (e.button & 1) e.which = 1; // левая кнопка
    else if (e.button & 4) e.which = 2; // средняя кнопка
    else if (e.button & 2) e.which = 3; // правая кнопка
  }
}
```

Свойства pageX/pageY

В IE до версии 9 не поддерживаются свойства `pageX/pageY`, но их можно получить, прибавив к `clientX/clientY` величину прокрутки страницы.

Более подробно о её вычислении вы можете прочитать в разделе [прокрутка страницы](#).

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

```
function fixPageXY(e) {
  if (e.pageX == null && e.clientX != null) { // если нет pageX...
    var html = document.documentElement;
    var body = document.body;

    e.pageX = e.clientX + (html.scrollLeft || body && body.scrollLeft || 0);
    e.pageX -= html.clientLeft || 0;

    e.pageY = e.clientY + (html.scrollTop || body && body.scrollTop || 0);
    e.pageY -= html.clientTop || 0;
  }
}
```

Итого

События мыши имеют следующие свойства:

- Кнопка мыши: `which` (для IE8+: нужно ставить из `button`)
- Элемент, вызвавший событие: `target`
- Координаты, относительно окна: `clientX/clientY`
- Координаты, относительно документа: `pageX/pageY` (для IE8+: нужно ставить по `clientX/Y` и прокрутке)
- Если зажата спец. клавиша, то стоит соответствующее свойство: `altKey`, `ctrlKey`, `shiftKey` или `metaKey` (Mac).
- Для поддержки `Ctrl+click` не забываем проверить `if (e.metaKey || e.ctrlKey)`, чтобы пользователи Mac тоже были довольны.

✓ Задачи

Список с выделением

важность: 5

Эта задача состоит из трёх частей.

1. Сделайте список, элементы которого можно выделять кликом.
2. Добавьте мульти-выделение. Если клик с нажатым `Ctrl` (Cmd под Mac), то элемент добавляется/удаляется из выделенных.
3. Добавьте выделение промежутков. Если происходит клик с нажатым `Shift`, то к выделению добавляется промежуток элементов от предыдущего кликнутого до этого. При этом не важно, какое именно действие делал предыдущий клик. Это похоже на то, как работает файловый менеджер в ряде ОС, но чуть проще, так как конкретная реализация выделений различается у разных ОС, и её точное воспроизведение не входит в эту задачу.

Демо:

Клик на элементе выделяет только его.

`Ctrl(Cmd)+Клик` добавляет/убирает элемент из выделенных.

`Shift+Клик` добавляет промежуток от последнего кликнутого к выделению.

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

P.S. В этой задаче можно считать, что в элементах списка может быть только текст, без вложенных тегов. P.P.S. Обработка одновременного нажатия Ctrl(Cmd) и Shift может быть любой.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Дерево: проверка клика на заголовке

важность: 3

Есть кликабельное JavaScript-дерево UL/LI (см. задачу [Раскрывающееся дерево](#)).

```
<ul>
  <li>Млекопитающие
    <ul>
      <li>Коровы</li>
      <li>Ослы</li>
      <li>Собаки</li>
      <li>Тигры</li>
    </ul>
  </li>
</ul>
```

При клике на заголовке его список его детей скрывается-раскрывается. Выглядит это так: (кликайте на заголовки)

- Животные
 - Млекопитающие
 - Коровы
 - Ослы
 - Собаки
 - Тигры
 - Другие
 - Змеи
 - Птицы
 - Ящерицы
- Рыбы
 - Аквариумные
 - Гуппи
 - Скалярии
 - Морские
 - Морская форель

Однако, проблема в том, что скрытие-раскрытие происходит даже при клике *вне* заголовка, на пустом пространстве справа от него.

Как скрывать/раскрывать детей только при клике на заголовок?

В задаче [Раскрывающееся дерево](#) это решено так: заголовки завёрнуты в элементы SPAN и проверяются клики только на них. Представим на минуту, что мы не хотим оборачивать текст в

SPAN, а хотим оставить как есть. Например, по соображениям производительности, если дерево и так очень большое, ведь оборачивание всех заголовков в SPAN увеличит количество DOM-узлов в 2 раза.

Решите задачу без обёртывания заголовков в SPAN, используя работу с координатами.

Исходный документ содержит кликабельное дерево.

P.S. Задача — скорее на сообразительность, однако подход может быть полезен в реальной жизни.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Мышь: отмена выделения, невыделяемые элементы

У кликов мыши есть неприятная особенность.

Двойной клик или нажатие с движением курсора как правило инициируют выделение текста.

Если мы хотим обрабатывать эти события сами, то такое выделение — некрасиво и неудобно. В этой главе мы рассмотрим основные способы, как делать элемент невыделяемым.

Для полноты картины, среди них будут и такие, которые применимы не только к событиям мыши.

Способ 1: отмена mousedown/selectstart

Проблема: браузер выделяет текст при движении мышью с зажатой левой кнопкой, а также при двойном клике на элемент. Даже там, где это не нужно.

Если сделать двойной клик на таком элементе, то обработчик сработает. Но побочным эффектом является *выделение текста браузером*.

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

Текст

Чтобы избежать выделения, мы должны предотвратить действие браузера по умолчанию для события [selectstart](#) ↗ 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 станут невыделяемыми:

Элементы списка не выделяются при клике:

```
<ul onmousedown="return false" onselectstart="return false">
  <li>Винни-Пух</li>
  <li>Ослик Иа</li>
  <li>Мудрая Сова</li>
  <li>Кролик. Просто кролик.</li>
</ul>
```

Элементы списка не выделяются при клике:

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

➊ Выделение, всё же, возможно

Отмена действия браузера при `mousedown/selectstart` отменяет выделение при клике, но не запрещает его полностью.

Если пользователь всё же хочет выделить текстовое содержимое элемента, то он может сделать это.

Достаточно начать выделение (зажать кнопку мыши) не на самом элементе, а рядом с ним. Ведь там отмены не произойдёт, выделение начнётся, и дальше можно передвинуть мышь уже на элемент.

Способ 2: снятие выделения пост-фактум

Вместо *предотвращения* выделения, можно его снять в обработчике события, *после* того, как оно уже произошло.

Для этого мы используем методы работы с выделением, которые описаны в отдельной главе [Выделение: Range, TextRange и Selection](#). Здесь нам понадобится всего лишь одна функция `clearSelection`, которая будет снимать выделение.

Пример со снятием выделения при двойном клике на элемент списка:

```

<ul>
  <li ondblclick="clearSelection()">Выделение отменяется при двойном клике.</li>
</ul>

<script>
function clearSelection() {
  if (window.getSelection) {
    window.getSelection().removeAllRanges();
  } else { // старый IE
    document.selection.empty();
  }
}
</script>

```

- Выделение отменяется при двойном клике.

У этого подхода есть две особенности, на которые стоит обратить внимание:

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

Способ 3: свойство user-select

Существует нестандартное CSS-свойство `user-select`, которое делает элемент невыделяемым.

Оно когда-то планировалось в стандарте CSS3, потом от него отказались, но поддержка в браузерах уже была сделана и потому осталась.

Это свойство работает (с префиксом) везде, кроме IE9-:

```

<style>
  b {
    -webkit-user-select: none;
    /* user-select -- это нестандартное свойство */

    -moz-user-select: none;
    /* поэтому нужны префиксы */

    -ms-user-select: none;
  }
</style>

```

Строка до..

```

<div ondblclick="alert('Тест')">
  <b>Этот текст нельзя выделить (кроме IE9-)</b>
</div>
.. Страна после

```

Строка до..
Этот текст нельзя выделить (кроме IE9-)
.. Страна после

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

IE9-: атрибут `unselectable="on"`

В IE9- нет `user-select`, но есть атрибут [unselectable](#).

Он отменяет выделение, но у него есть особенности:

1. Во-первых, невыделяемость не наследуется. То есть, невыделяемость родителя не делает невыделяемыми детей.
2. Во-вторых, текст, в отличие от `user-select`, всё равно можно выделить, если начать выделение не на самом элементе, а рядом с ним.

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

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

Итого

Для отмены выделения есть несколько способов:

1. CSS-свойство `user-select` — везде кроме IE9- (нужен префикс, нестандарт).
2. Атрибут `unselectable="on"` — работает для любых IE (должен быть у всех потомков)
3. Отмена действий на `mousedown` и `selectstart`:

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

4. Отмена выделения пост-фактум через функцию `clearSelection()`, описанную выше.

Какой же способ выбирать?

Это зависит от задач и вашего удобства, а также конкретного случая. Все описанные способы

работают.

Недостаток `user-select` — в том, что посетитель теряет возможность скопировать текст. А что, если он захочет именно это сделать?

В любом случае эти способы не предназначены для защиты от выделения-и-копирования.

Если уж хочется запретить копирование — можно использовать событие `oncopy`:

```
<div oncopy="alert('Копирование запрещено');return false">  
    Уважаемый копирователь,  
    почему-то автор хочет заставить вас покопаться в исходном коде этой страницы.  
    Если вы знаете JS или HTML, то скопировать текст не составит для вас проблем,  
    ну а если нет, то увы...  
</div>
```

Уважаемый копирователь, почему-то автор хочет заставить вас покопаться в исходном коде этой страницы. Если вы знаете JS или HTML, то скопировать текст не составит для вас проблем, ну а если нет, то увы...

Мышь: движение `mouseover/out, mouseenter/leave`

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.



При этом мы можем узнать, с какого элемента пришла (или на какой ушла) мышь, используя дополнительное свойство объекта события `relatedTarget`.

Например, в обработчике события `mouseover`:

- `event.target` — элемент, на который пришла мышь, то есть на котором возникло событие.
- `event.relatedTarget` — элемент, с которого пришла мышь.

Для `mouseout` всё наоборот:

- `event.target` — элемент, с которого ушла мышь, то есть на котором возникло событие.
- `event.relatedTarget` — элемент, на который перешла мышь.

В примере ниже, если у вас есть мышь, вы можете наглядно посмотреть события `mouseover/out, mouseenter/leave`, возникающие на всех элементах.

[Смотреть пример онлайн ↗](#)

⚠ relatedTarget может быть null

Свойство `relatedTarget` может быть равно `null`.

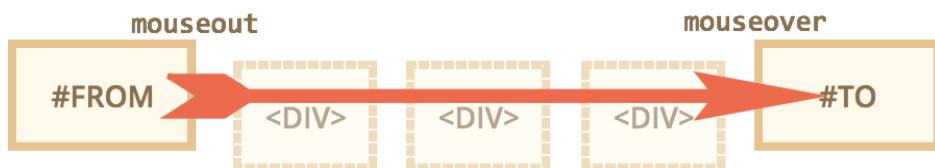
Это вполне нормально и означает, что мышь пришла не с другого элемента, а из-за пределов окна (или ушла за окно). Мы обязательно должны иметь в виду такую возможность, когда пишем код, который обращается к свойствам `event.relatedTarget`.

Частота событий

Событие `mousemove` срабатывает при передвижении мыши. Но это не значит, что каждый пиксель экрана порождает отдельное событие!

События `mousemove` и `mouseover/mouseout` срабатывают так часто, насколько это позволяет внутренняя система взаимодействия с мышью браузера.

Это означает, что если посетитель двигает мышью быстро, то DOM-элементы, через которые мышь проходит на большой скорости, могут быть пропущены.

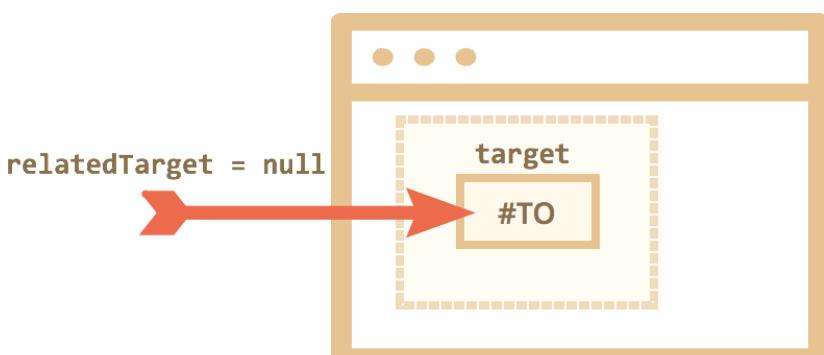


При быстром движении с элемента `#FROM` до элемента `#TO`, как изображено на картинке выше — промежуточные `<DIV>` будут пропущены. Сработает только событие `mouseout` на `#FROM` и `mouseover` на `#TO`.

На практике это полезно, потому что таких промежуточных элементов может быть много, и если обрабатывать заход и уход с каждого — дополнительные вычислительные затраты.

С другой стороны, мы должны это понимать и не рассчитывать на то, что мышь аккуратно пройдёт с одного элемента на другой и так далее. Нет, она «прыгает».

В частности, возможна ситуация, когда курсор прыгает в середину страницы, и при этом `relatedTarget=null`, то есть он пришёл «ниоткуда» (на самом деле извне окна):



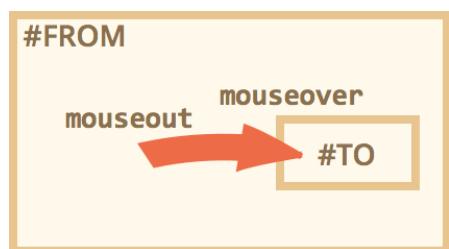
Обратим внимание еще на такую деталь. При быстром движении курсор окажется над #TO сразу, даже если этот элемент глубоко в DOM. Его родители при движении сквозь них события не поймают.

Важно иметь в виду эту особенность событий, чтобы не написать код, который рассчитан на последовательный проход над элементами. В остальном это вполне удобно.

«Лишний» mouseout при уходе на потомка

Представьте ситуацию — курсор зашёл 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.

При переходе на потомка срабатывает mouseout на родителе.



Это кажется странным, но легко объяснимо.

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 куда-нибудь, то автоматически ушёл с предыдущего элемента. Всё просто.

Самое забавное начинается чуть позже.

Ведь события mouseover и mouseout всплывают.

Получается, что если поставить обработчики mouseover и mouseout на #FROM и #TO, то последовательность срабатывания при переходе #FROM → #TO будет следующей:

1. mouseout на #FROM (с `event.target=#FROM, event.relatedTarget=#TO`).
2. mouseover на #TO (с `event.target=#TO, event.relatedTarget=#FROM`).
3. Событие mouseover после срабатывания на #TO всплывает выше, запуская обработчики mouseover на родителях. Ближайший родитель — как раз #FROM, то есть сработает обработчик mouseover на нём, с теми же значениями `target/relatedTarget`.

Если посмотреть на 1) и 3), то видно, что то видно, что на #FROM сработает сначала mouseout, а затем с #TO всплыёт mouseover.

Если по mouseover мы что-то показываем, а по mouseout — скрываем, то может получаться «мигание».

У обработчиков создаётся впечатление, что курсор ушёл mouseout с родителя, а затем тут же

перешел mouseover на него (за счет всплытия mouseover с потомка).

Если действия при наведении и уходе курсора с родителя простые, например скрытие/показ подсказки, то можно вообще ничего не заметить. Ведь события происходят сразу одно за другим, подсказка будет скрыта по mouseout и тут же показана по mouseover.

Если же происходит что-то более сложное, то бывает важно отследить момент «настоящего» ухода, то есть понять, когда элемент зашёл на родителя, а когда ушёл — без учёта переходов по дочерним элементам.

Для этого можно использовать 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. Не учитываются переходы внутри элемента.
2. События mouseenter/mouseleave не всплывают.

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.

Делегирование

События mouseenter/leave более наглядны и понятны, но они не всплывают, а значит с ними нельзя использовать делегирование.

Представим, что нам нужно обработать вход/выход мыши для ячеек таблицы. А в таблице таких ячеек тысяча.

Естественное решение — поставить обработчик на верхний элемент `<table>` и ловить все события в нём. Но события mouseenter/leave не всплывают, они срабатывают именно на том элементе, на котором стоит обработчик и только на нём.

Если обработчики mouseenter/leave стоят на `<table>`, то они сработают при входе-выходе из таблицы, но получить из них какую-то информацию о переходах по её ячейкам невозможно.

Не беда — воспользуемся mouseover/mouseout.

Простейший вариант обработчиков выглядит так:

```
table.onmouseover = function(event) {
  var target = event.target;
  target.style.background = 'pink';
};

table.onmouseout = function(event) {
  var target = event.target;
  target.style.background = '';
};
```

В таком виде они срабатывают при переходе с любого элемента на любой. Нас же интересуют переходы строго с одной ячейки `<td>` на другую.

Нужно фильтровать события.

Один из вариантов:

- Запоминать текущий подсвеченный `<td>` в переменной.
- При `mouseover` проверять, остались ли мы внутри того же `<td>`, если да — игнорировать.
- При `mouseout` проверять, если мы ушли с текущего `<td>`, а не перешли куда-то внутрь него, то игнорировать.

Детали кода вы можете посмотреть в [полном примере ↗](#).

Особенности IE8-

В IE8- нет свойства `relatedTarget`. Вместо него используется `fromElement` для `mouseover` и `toElement` для `mouseout`.

Можно «исправить» несовместимость с `relatedTarget` так:

```
function fixRelatedTarget(e) {
  if (e.relatedTarget === undefined) {
    if (e.type == 'mouseover') e.relatedTarget = e.fromElement;
    if (e.type == 'mouseout') e.relatedTarget = e.toElement;
  }
}
```

Итого

У `mouseover`, `mousemove`, `mouseout` есть следующие особенности:

- При быстром движении мыши события `mouseover`, `mousemove`, `mouseout` могут пропускать промежуточные элементы.
- События `mouseover` и `mouseout` — единственные, у которых есть вторая цель: `relatedTarget` (`toElement`/`fromElement` в IE).
- События `mouseover/mouseout` подразумевают, что курсор находится над одним, самым глубоким элементом. Они срабатывают при переходе с родительского элемента на дочерний.

События `mouseenter/mouseleave` не всплывают и не учитывают переходы внутри элемента.

Задачи

Поведение «вложенная подсказка»

важность: 5

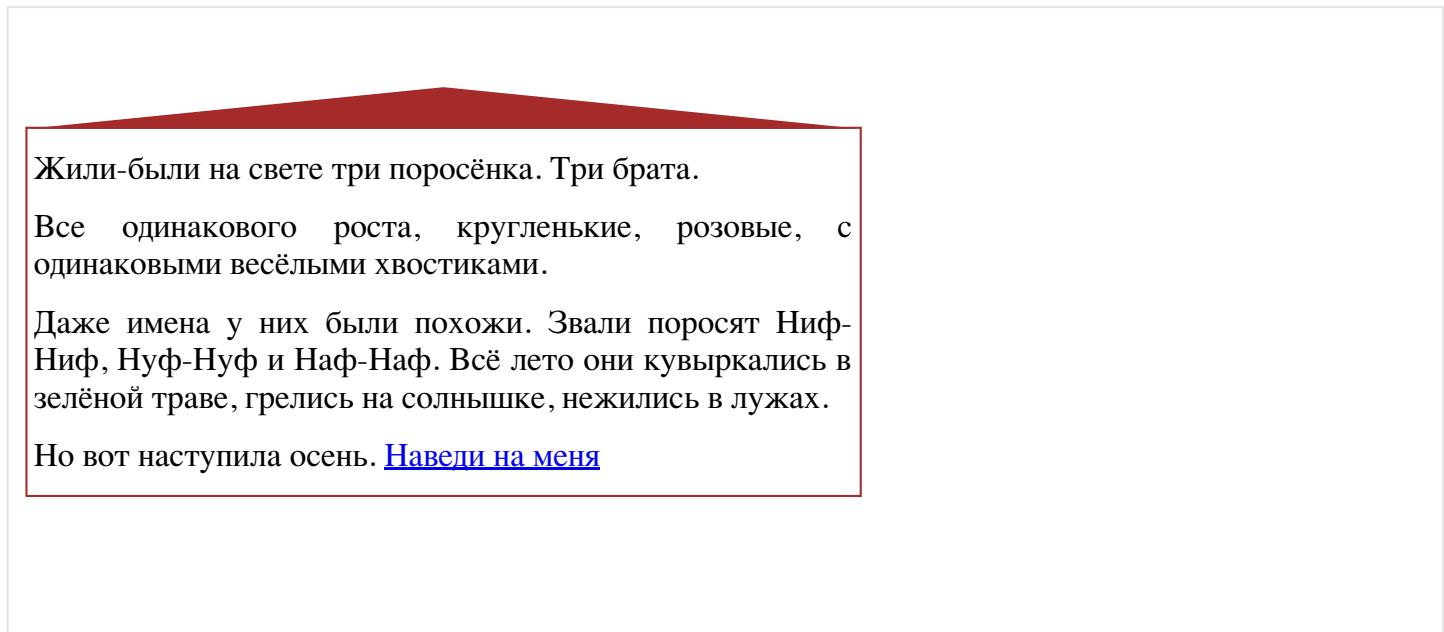
Напишите JS-код, который будет показывать всплывающую подсказку над элементом, если у него есть атрибут `data-tooltip`.

Условие аналогично задаче [Поведение «подсказка»](#), но здесь необходима поддержка вложенных элементов. При наведении показывается самая вложенная подсказка.

Например:

```
<div data-tooltip="Это – внутренность дома" id="house">
  <div data-tooltip="Это – крыша" id="roof"></div>
  ...
  <a href="http://ru.wikipedia.org/wiki/Три_поросёнка" data-tooltip="Читать дальше">Наведи на меня</a>
</div>
```

Результат в ифрейме с документом:



Жили-были на свете три поросёнка. Три брата. Все одинакового роста, кругленькие, розовые, с одинаковыми весёлыми хвостиками. Даже имена у них были похожи. Звали пороссят Ниф-Ниф, Нуф-Нуф и Наф-Наф. Всё лето они кувыркались в зелёной траве, грелись на солнышке, нежились в лужах. Но вот наступила осень. [Наведи на меня](#)

Вы можете использовать как заготовку решение задачи [Поведение «подсказка»](#).

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Подсказка при замедлении над элементом

важность: 5

Нужно написать функцию, которая показывает подсказку при *наведении* на элемент, но не при *быстром проходе* над ним.

То есть, если посетитель именно навёл курсор мыши на элемент и почти остановился — подсказку показать, а если быстро провёл над ним, то не надо, зачем излишнее мигание?

Технически — можно измерять скорость движения мыши над элементом, если она маленькая, то считаем, что это «наведение на элемент» (показать подсказку), если большая — «быстрый проход мимо элемента» (не показывать).

Реализуйте это через универсальный объект `new HoverIntent(options)`, с параметрами `options`:

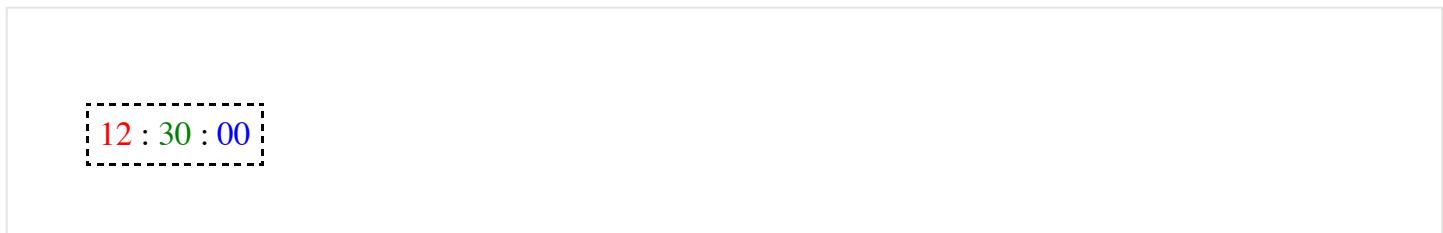
- `elem` — элемент, наведение на который нужно отслеживать.
- `over` — функция-обработчик наведения на элемент.
- `out` — функция-обработчик ухода с элемента (если было наведение).

Пример использования такого объекта для подсказки:

```
// образец подсказки
var tooltip = document.createElement('div');
tooltip.className = "tooltip";
tooltip.innerHTML = "Подсказка";

// при "наведении на элемент" показать подсказку
new HoverIntent({
  elem: elem,
  over: function() {
    tooltip.style.left = this.getBoundingClientRect().left + 'px';
    tooltip.style.top = this.getBoundingClientRect().bottom + 5 + 'px';
    document.body.appendChild(tooltip);
  },
  out: function() {
    document.body.removeChild(tooltip);
  }
});
```

Демо этого кода:



Если провести мышкой над «часиками» быстро, то ничего не будет, а если медленно или остановиться на них, то появится подсказка.

Обратите внимание — подсказка не «мигает» при проходе мыши внутри «часиков», по подэлементам.

[Открыть песочницу для задачи.](#) ↗

Мышь: 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.

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.

Отличия от HTML5 Drag'n'Drop

В современном стандарте HTML5 есть поддержка Drag'n'Drop при помощи [специальных событий ↗](#).

Эти события поддерживаются всеми современными браузерами, и у них есть свои интересные особенности, например, можно перетащить файл в браузер, так что JS получит доступ к его содержимому. Они заслуживают отдельного рассмотрения.

Но в плане именно Drag'n'Drop у них есть существенные ограничения. Например, нельзя организовать перенос «только по горизонтали» или «только по вертикали». Также нельзя ограничить перенос внутри заданной зоны. Есть и другие интерфейсные задачи, которые такими встроенными событиями нереализуемы.

Поэтому здесь мы будем рассматривать Drag'n'Drop при помощи событий мыши.

Рассматриваемые приёмы, вообще говоря, применяются не только в Drag'n'Drop, но и для любых интерфейсных взаимодействий вида «захватить – потянуть – отпустить».

Алгоритм Drag'n'Drop

Основной алгоритм Drag'n'Drop выглядит так:

1. Отслеживаем нажатие кнопки мыши на переносимом элементе при помощи события `mousedown`.
2. При нажатии — подготовить элемент к перемещению.
3. Далее отслеживаем движение мыши через `mousemove` и передвигаем переносимый элемент на новые координаты путём смены `left/top` и `position:absolute`.
4. 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 для переноса мяча:

```

var ball = document.getElementById('ball');

ball.onmousedown = function(e) { // 1. отследить нажатие

    // подготовить к перемещению
    // 2. разместить на том же месте, но в абсолютных координатах
    ball.style.position = 'absolute';
    moveAt(e);
    // переместим в body, чтобы мяч был точно не внутри position:relative
    document.body.appendChild(ball);

    ball.style.zIndex = 1000; // показывать мяч над другими элементами

    // передвинуть мяч под координаты курсора
    // и сдвинуть на половину ширины/высоты для центрирования
    function moveAt(e) {
        ball.style.left = e.pageX - ball.offsetWidth / 2 + 'px';
        ball.style.top = e.pageY - ball.offsetHeight / 2 + 'px';
    }

    // 3, перемещать по экрану
    document.onmousemove = function(e) {
        moveAt(e);
    }

    // 4. отследить окончание переноса
    ball.onmouseup = function() {
        document.onmousemove = null;
        ball.onmouseup = null;
    }
}

```

Если запустить этот код, то мы заметим нечто странное. При начале переноса мяч «раздваивается» и переносится не сам мяч, а его «клон».

Это потому, что браузер имеет свой собственный Drag'n'Drop, который автоматически запускается и вступает в конфликт с нашим. Это происходит именно для картинок и некоторых других элементов.

Его нужно отключить:

```

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

```

Теперь всё будет в порядке.

Ещё одна особенность правильного Drag'd'Drop — событие `mousemove` отслеживается на `document`, а не на `ball`.

С первого взгляда кажется, что мышь всегда над мячом и обработчик `mousemove` можно повесить на сам мяч, а не на документ.

Однако, на самом деле мышь во время переноса не всегда над мячом.

Вспомним, событие `mousemove` возникает хоть и 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 всем `document`.

Правильное позиционирование

В примерах выше мяч позиционируется в центре под курсором мыши:

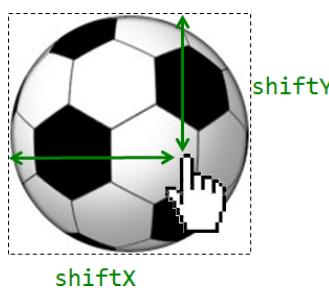
```
self.style.left = e.pageX - ball.offsetWidth / 2 + 'px';
self.style.top = e.pageY - ball.offsetHeight / 2 + 'px';
```

Если поставить `left/top` ровно в `pageX/pageY`, то мячик прилипнет верхним-левым углом к курсору мыши. Будет некрасиво. Поэтому мы сдвигаем его на половину высоты/ширины, чтобы был центром под мышью. Уже лучше.

Но не идеально. В частности, в самом начале переноса, особенно если мячик «взят» за край — он резко «прыгает» центром под курсор мыши.

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

Где захватили, за ту «часть элемента» и переносим:



1. Когда человек нажимает на мячик `mousedown` — курсор сдвинут относительно левого-верхнего угла мяча на расстояние, которое мы обозначим `shiftX/shiftY`. И нужно при переносе сохранить этот сдвиг.

Получить значения `shiftX/shiftY` легко: достаточно вычесть из координат курсора `pageX/pageY` левую-верхнюю границу мячика, полученную при помощи функции `getCoords`.

При Drag'n'Drop мы везде используем координаты относительно документа, так как они подходят в большем количестве ситуаций.

Конечно же, не проблема перейти к координатам относительно окна, если это понадобится. Достаточно использовать `position:fixed`, `elem.getBoundingClientRect()` для определения координат и `e.clientX/Y`.

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

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

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

Итоговый код с правильным позиционированием:

```
var ball = document.getElementById('ball');  
  
ball.onmousedown = function(e) {  
  
    var coords = getCoords(ball);  
    var shiftX = e.pageX - coords.left;  
    var shiftY = e.pageY - coords.top;  
  
    ball.style.position = 'absolute';  
    document.body.appendChild(ball);  
    moveAt(e);  
  
    ball.style.zIndex = 1000; // над другими элементами  
  
    function moveAt(e) {  
        ball.style.left = e.pageX - shiftX + 'px';  
        ball.style.top = e.pageY - shiftY + 'px';  
    }  
  
    document.onmousemove = function(e) {  
        moveAt(e);  
    };  
  
    ball.onmouseup = function() {  
        document.onmousemove = null;  
        ball.onmouseup = null;  
    };  
}  
  
ball.ondragstart = function() {  
    return false;  
};
```

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:

- События ball.mousedown → document.mousemove → ball.mouseup.
- Передвижение с учётом изначального сдвига shiftX/shiftY.
- 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.
- При обработке событий `mousedown` и `mouseup` можно использовать делегирование, так что одного обработчика достаточно для управления переносом в зоне с сотнями элементов.

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

✓ Задачи

Слайдер

важность: 5

Создайте слайдер:

Захватите мышкой синий бегунок и двигайте его, чтобы увидеть в работе.

Важно:

- Слайдер должен нормально работать при резком движении мыши влево или вправо, за пределы полосы. При этом бегунок должен останавливаться четко в нужном конце полосы.
- При нажатом бегунке мышь может выходить за пределы полосы слайдера, но слайдер пусть все равно работает (это удобно для пользователя).

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Расставить супергероев по полю

важность: 5

В этой задаче вы можете проверить своё понимание сразу нескольких аспектов Drag'n'Drop.

Сделайте так, чтобы элементы с классом `draggable` можно было переносить мышкой. По окончании переноса элемент остаётся на том месте в документе, где его положили.

Требования к реализации:

- Должен быть 1 обработчик на `document`, использующий делегирование.
- Если элементы подносят к вертикальным краям окна — оно должно прокручиваться вниз/вверх.
- Горизонтальной прокрутки в этой задаче не существует.

- Элемент при переносе, даже при резких движениях мышкой, не должен попасть вне окна.

Футбольное поле в этой задаче слишком большое, чтобы показывать его здесь, поэтому откройте его, кликнув по ссылке ниже. Там же и подробное описание задачи (осторожно, винни-пух и супергерои!).

[Демо в новом окне ↗](#)

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Мышь: Drag'n'Drop более глубоко

В [предыдущей статье](#) мы рассмотрели основы Drag'n'Drop. Здесь мы разберём дополнительные «тонкие места» и приёмы реализации, которые возникают на практике.

Почти все javascript-библиотеки реализуют Drag'n'Drop так, как написано (хотя бывает что и менее эффективно).

Зная, что и как, вы сможете легко написать свой код переноса или поправить, адаптировать существующую библиотеку под себя.

Этот материал не строго обязательен для изучения, он специфичен именно для Drag'n'Drop.

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 классом `draggable`.
- Элементы, на которые можно положить (компьютер), имеют класс `droppable`.

```






<p>Браузер переносить сюда:</p>


```

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».
- Продвинутые 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, конечно, делегирование.

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.

Поэтому 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 будем брать `document`, хотя это может быть и любой элемент.

Найденный `draggable`-элемент сохраним в свойстве `dragObject.elem` и начнём двигать.

Код обработчика `mousedown`:

```
var dragObject = {};  
  
document.onmousedown = function(e) {  
  
    if (e.which != 1) { // если клик правой кнопкой мыши  
        return; // то он не запускает перенос  
    }  
  
    var elem = e.target.closest('.draggable');  
  
    if (!elem) return; // не нашли, клик вне draggable-объекта  
  
    // запомнить переносимый объект  
    dragObject.elem = elem;  
  
    // запомнить координаты, с которых начал перенос объекта  
    dragObject.downX = e.pageX;  
    dragObject.downY = e.pageY;  
}  
}
```

⚠ Не начинаем перенос по mousedown

Ранее мы по `mousedown` начинали перенос.

Но на самом деле нажатие на элемент вовсе не означает, что его собираются куда-то двигать. Возможно, на нём просто кликают.

Это важное различие. Снимать элемент со своего места и куда-то двигать нужно только при переносе.

Чтобы отличить перенос от клика, в том числе — от клика, который сопровождается нечаянным перемещением на пару пикселей (рука дрогнула), мы будем запоминать в `dragObject`, какой элемент (`elem`) и где (`downX/downY`) 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 будет иметь такой вид:

```
document.onmousemove = function(e) {  
    if (!dragObject.elem) return; // элемент не зажат  
  
    if (!dragObject.avatar) { // элемент нажат, но пока не начали его двигать  
        ...начать перенос, присвоить dragObject.avatar = переносимый элемент  
    }  
  
    ...отобразить перенос элемента...  
}
```

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 потому, что он слишком большой. А удобно создать некоторое визуальное представление элемента, и его уже переносить. Аватор 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:

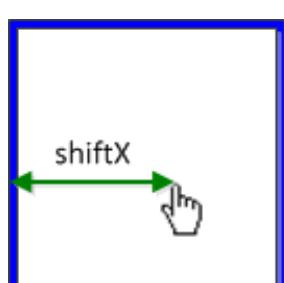
```
// в начале переноса:  
if (avatar.parentNode != document.body) {  
    document.body.appendChild(avatar); // переместить в BODY, если надо  
}  
avatar.style.zIndex = 9999; // сделать, чтобы элемент был над другими  
avatar.style.position = 'absolute';
```

... А затем его можно двигать:

```
// при каждом движении мыши  
avatar.style.left = новая координата + 'px';  
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 [Мышь: Drag'n'Drop](#).

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:

```
document.onmousemove = function(e) {
  if (!dragObject.elem) return; // элемент не зажат

  if ( !dragObject.avatar ) { // если перенос не начал...

    // посчитать дистанцию, на которую переместился курсор мыши
    var moveX = e.pageX - dragObject.downX;
    var moveY = e.pageY - dragObject.downY;
    if ( Math.abs(moveX) < 3 && Math.abs(moveY) < 3 ) {
      return; // ничего не делать, мышь не передвинулась достаточно далеко
    }

    dragObject.avatar = createAvatar(e); // захватить элемент
    if (!dragObject.avatar) {
      dragObject = {}; // аватар создать не удалось, отмена переноса
      return; // возможно, нельзя захватить за эту часть элемента
    }

    // аватар создан успешно
    // создать вспомогательные свойства shiftX/shiftY
    var coords = getCoords/avatar);
    dragObject.shiftX = dragObject.downX - coords.left;
    dragObject.shiftY = dragObject.downY - coords.top;

    startDrag(e); // отобразить начало переноса
  }

  // отобразить перенос объекта при каждом движении мыши
  dragObject.avatar.style.left = e.pageX - dragObject.shiftX + 'px';
  dragObject.avatar.style.top = e.pageY - dragObject.shiftY + 'px';

  return false;
}
```

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.

```
function createAvatar(e) {  
  
    // запомнить старые свойства, чтобы вернуться к ним при отмене переноса  
    var avatar = dragObject.elem;  
    var old = {  
        parent: avatar.parentNode,  
        nextSibling: avatar.nextSibling,  
        position: avatar.position || '',  
        left: avatar.left || '',  
        top: avatar.top || '',  
        zIndex: avatar.zIndex || ''  
    };  
  
    // функция для отмены переноса  
    avatar.rollback = function() {  
        old.parent.insertBefore(avatar, old.nextSibling);  
        avatar.style.position = old.position;  
        avatar.style.left = old.left;  
        avatar.style.top = old.top;  
        avatar.style.zIndex = old.zIndex  
    };  
  
    return avatar;  
}
```

Функция `startDrag(e)`, которую вызывает `mousemove`, если видит, что элемент в «зажатом» состоянии перенесли достаточно далеко, инициирует начало переноса и позиционирует аватар на странице:

```
function startDrag(e) {  
    var avatar = dragObject.avatar;  
  
    document.body.appendChild(avatar);  
    avatar.style.zIndex = 9999;  
    avatar.style.position = 'absolute';  
}
```

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:

```
document.onmouseup = function(e) {
  // (1) обработать перенос, если он идет
  if (dragObject.avatar) {
    finishDrag(e);
  }

  // в конце mouseup перенос либо завершился, либо даже не начался
  // (2) в любом случае очистим "состояние переноса" dragObject
  dragObject = {};
}
```

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

```
function finishDrag(e) {
  var dropElem = findDroppable(e);

  if (dropElem) {
    ... успешный перенос ...
  } else {
    ... отмена переноса ...
  }
}
```

Определяем элемент под курсором

Чтобы понять, над каким элементом мы остановились — используем метод `document.elementFromPoint(clientX, clientY)`, который мы обсуждали в разделе [координаты](#). 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 `droppable` под курсором мыши:

```
// возвратит ближайший droppable или null
// это предварительный вариант findDroppable, исправлен ниже!
function findDroppable(event) {

  // взять элемент на данных координатах
  var elem = document.elementFromPoint(event.clientX, event.clientY);

  // найти ближайший сверху droppable
  return elem.closest('.droppable');
}
```

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 дать аватару стиль `pointer-events:none` (кроме IE10-), либо:

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

Напишем функцию `findDroppable(event)`, которая это делает:

```
function findDroppable(event) {  
    // спрячем переносимый элемент  
    dragObject.avatar.hidden = true;  
  
    // получить самый вложенный элемент под курсором мыши  
    var elem = document.elementFromPoint(event.clientX, event.clientY);  
  
    // показать переносимый элемент обратно  
    dragObject.avatar.hidden = false;  
  
    if (elem == null) {  
        // такое возможно, если курсор мыши "вылетел" за границу окна  
        return null;  
    }  
  
    return target.closest('.droppable');  
}
```

DragManager

Из фрагментов кода, разобранных выше, можно собрать мини-фреймворк.

Объект `DragManager` будет запоминать текущий переносимый объект и отслеживать его перенос.

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:

```
var DragManager = new function() {  
  
    var dragObject = {};  
  
    var self = this; // для доступа к себе из обработчиков  
  
    function onMouseDown(e) { ... }  
    function onMouseMove(e) { ... }  
    function onMouseUp(e) { ... }  
  
    document.onmousedown = onMouseDown;  
    document.onmousemove = onMouseMove;  
    document.onmouseup = onMouseUp;  
  
    this.onDragEnd = function(dragObject, dropElem) { };  
    this.onDragCancel = function(dragObject) { };  
}
```

всю работу будут выполнять обработчики `onmouse*`, которые оформлены как локальные функции. В данном случае они ставятся на `document` через `on...`, но это легко поменять это на `addEventListener`.

Код функция `onMouse*` мы подробно рассмотрели ранее, но вы сможете увидеть их в полном примере ниже.

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

У него будут следующие свойства, которые также разобраны выше:

elem

Текущий зажатый 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. Что же касается действий при его окончании — их должен назначить внешний код, который использует `DragManager`.

Можно сделать это через вспомогательные методы `onDrag*`, которые устанавливаются внешним кодом и затем вызываются фреймворком. Разработчик, подключив `DragManager`, описывает в этих методах, что делать при начале и завершении переноса. Конечно же, можно заменить методы `onDrag*` на генерацию «своих» событий.

С использованием `DragManager` пример, с которого начиналась эта глава — перенос иконок браузеров в компьютер, реализуется совсем просто:

```
DragManager.onDragEnd = function(dragObject, dropElem) {
    // скрыть/удалить переносимый объект
    dragObject.elem.hidden = true;

    // успешный перенос, показать улыбку классом computer-smile
    dropElem.className = 'computer computer-smile';

    // убрать улыбку через 0.2 сек
    setTimeout(function() {
        dropElem.classList.remove('computer-smile');
    }, 200);
};

DragManager.onDragCancel = function(dragObject) {
    // откат переноса
    dragObject.avatar.rollback();
};
```

Полный пример с кодом:

[Смотреть пример онлайн ↗](#)

Расширения

Существует масса возможных применений Drag'n'Drop. Здесь мы не будем реализовывать их все, поскольку не стоит цель создать фреймворк-монстр.

Однако, мы рассмотрим их, чтобы, при необходимости, легко было написать то, что нужно.

Захватывать элемент можно только за «ручку»

Часто бывает, что перенос должен быть инициирован только при захвате за определённую зону элемента. К примеру, модальное окно можно «взять», только захватив его за заголовок.

Для этого достаточно добавить необходимую проверку, к примеру, в функцию `createAvatar` или перед её запуском.

Если `mousedown` был внутри элемента, помеченного, к примеру, классом `draghandle`, то начинаем перенос, иначе — нет.

Проверка прав на `droppable`

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

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» (`droppable`), а узел «пользователи» — нельзя. Но и то и другое можно поместить в «корзину».

Здесь решение: «можно или нет» переносить или нельзя зависит от «типа» переносимого объекта.

Есть и более сложные варианты, когда решение зависит от конкретного места в `droppable`, над которым посетитель отпустил кнопку мыши. К примеру, переносить в верхнюю часть можно, а в нижнюю — нет.

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

Однако, на практике бывают ситуации, когда решение «прямо сейчас» принять невозможно. Например, нужно сделать запрос на сервер: «А разрешено ли текущему посетителю производить такую операцию?»

Как при этом должен вести себя интерфейс? Можно, конечно сделать, чтобы элемент после отпускания кнопки мыши «завис» над `droppable`, ожидая ответа. Однако, такое решение неудобно в реализации и странновато выглядит для посетителя.

Как правило, применяют «оптимистичный» алгоритм, по которому мы считаем, что перенос обычно успешен, но при необходимости можем отменить его.

При нём посетитель кладет объект туда, куда он хочет, а 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()`. Автар 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.

Подсветка текущего `droppable`

Удобно, когда пользователь во время переноса наглядно видит, куда он сейчас положит `draggable`. Например, текущий `droppable` (или его часть) подсвечиваются.

Для этого в `DragManager` можно добавить дополнительные методы интеграции с внешним кодом:

- `onDragEnter` — будет вызываться при заходе на `droppable`, из `onMouseMove`.
- `onDragMove` — при каждом передвижении внутри `droppable`, из `onMouseMove`.
- `onDragLeave` — при выходе с `droppable`, из `onMouseMove` и `onMouseUp`.

Возможен более сложный вариант, когда нужно поддерживать не только перенос в элемент, но и перенос между элементами, например вставку одной статьи между двумя другими.

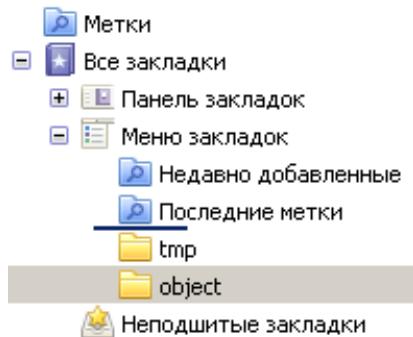
Для этого код, который обрабатывает перенос, может «делить на части» `droppable`, к примеру, в отношении 25% – 50% – 25%, и смотреть:

- Если перенос в верхнюю четверть, то это — «над».
- Если перенос в середину, то это «внутрь».

- Если перенос в нижнюю четверть, то это — «под».

Текущий `draggable` и позиция относительно него при этом могут помечаться подсветкой и жирной чертой над/под, если требуется.

Пример индикации из Firefox:



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.rollback()`.

Итого

Уточнённый алгоритм Drag'n'Drop:

1. При `mousedown` запомнить координаты нажатия.
2. При `mousemove` инициировать перенос, как только зажатый элемент передвинули на 3 пикселя или больше. Сообщить во внешний код вызовом `onDragStart`.
 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. Сообщать во внешний код о текущем `draggable` под курсором и позиции над ним вызовами `onDragEnter, onDragMove, onDragLeave`.
3. При `mouseup` обработать завершение переноса. Элемент под аватаром получить по координатам, предварительно спрятав аватар. Сообщить во внешний код вызовом `onDragEnd`.

Получившаяся реализация Drag'n'Drop проста, эффективна, изящна.

Её очень легко поменять или адаптировать под «особые» потребности.

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

Мышь: колёсико, событие wheel

колёсико мыши используется редко. Оно есть даже не у всех мышей. Поэтому существуют пользователи, которые в принципе не могут генерировать 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](#) — 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 генерировать `onscroll`.
- А событие `wheel` является чисто «мышиным». 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 ↗](#) не так давно. Оно поддерживается Chrome 31+, IE9+, Firefox 17+.

До него браузеры обрабатывали прокрутку при помощи событий [mousewheel ↗](#) (все кроме Firefox) и [DOMMouseScroll ↗](#), [MozMousePixelScroll ↗](#) (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`.

`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, включающий поддержку IE8-:

```
if (elem.addEventListener) {
  if ('onwheel' in document) {
    // IE9+, FF17+, Ch31+
    elem.addEventListener("wheel", onWheel);
  } else if ('onmousewheel' in document) {
    // устаревший вариант события
    elem.addEventListener("mousewheel", onWheel);
  } else {
    // Firefox < 17
    elem.addEventListener("MozMousePixelScroll", onWheel);
  }
} else { // IE8-
  elem.attachEvent("onmousewheel", onWheel);
}

function onWheel(e) {
  e = e || window.event;

  // wheelDelta не дает возможность узнать количество пикселей
  var delta = e.deltaY || e.detail || e.wheelDelta;

  var info = document.getElementById('delta');

  info.innerHTML = +info.innerHTML + delta;

  e.preventDefault ? e.preventDefault() : (e.returnValue = false);
}
```

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.

Задачи

Масштабирование колёсиком мыши

важность: 5

Сделайте так, чтобы при прокрутке колёсиком мыши над элементом, он масштабировался.

Масштабирование обеспечивайте при помощи свойства CSS transform:

```
// увеличение в 1.5 раза  
elem.style.transform = elem.style.WebkitTransform = elem.style.MsTransform = 'scale(1.5)';
```

Результат в iframe:

При прокрутке колёсика мыши над этим элементом, он будет масштабироваться.

[К решению](#)

Прокрутка без влияния на страницу

важность: 4

В большинстве браузеров (кроме Firefox) если в процессе прокрутки `textarea` мышкой (или жестами) мы достигаем границы, то прокрутка продолжается уже на уровне страницы.

Иными словами, если в примере ниже вы попробуете прокрутить `textarea` вниз, то когда прокрутка дойдёт до конца — начнёт прокручиваться документ:

Начало документа

```
прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня
```

То же самое происходит при прокрутке вверх.

В интерфейсах редактирования, когда большая `textarea` является основным элементом страницы, такое поведение может быть неудобно.

Для редактирования более оптимально, чтобы при прокрутке до конца `textarea` страница не «улетала» вверх и вниз.

Вот тот же документ, но с желаемым поведением `textarea`:

Начало документа

```
прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня прокрути меня прокрути меня прокрути меня прокрути меня прокрути  
меня
```

Задача:

- Создать скрипт, который при подключении к документу исправлял бы поведение всех `textarea`, чтобы при прокрутке они не трогали документ.
- Направление прокрутки — только вверх или вниз.
- Редактор прокручивает только мышкой или жестами (на мобильных устройствах), прокрутку клавиатурой здесь рассматривать не нужно (хотя это и возможно).

[К решению](#)

Мышь: IE8-, исправление события

Ранее мы говорили о различных несовместимостях при работе с событиями для IE8-. Самая главная — это, конечно, назначение событий при помощи `attachEvent/detachEvent` вместо `addEventListener/removeEventListener` и отсутствие фазы перехвата. Но и в самом объекте события есть различия.

Что касается событий мыши, различия в свойствах можно легко исправить при помощи функции `fixEvent`, которая описана в этой главе.

Только IE8-

Эта глава и описанная далее функция `fixEvent` нужны только для поддержки IE8-.

Если IE8- для Вас неактуален, то пролистывайте дальше, это читать Вам не надо.

Функция `fixEvent` предназначена для запуска в начале обработчика, вот так:

```
elem.onclick = function(event) {
    // если IE8-, то получить объект события window.event и исправить его
    event = event || fixEvent.call(this, window.event);
    ...
}
```

Она добавлит объекту события в IE8- следующие стандартные свойства:

- `target`
- `currentTarget` — если обработчик назначен не через `attachEvent`.
- `relatedTarget` — для `mouseover/mouseout` и `mouseenter/mouseleave`.
- `pageX/pageY`
- `which`

Код функции:

```

function fixEvent(e) {
    e.currentTarget = this;
    e.target = e.srcElement;

    if (e.type == 'mouseover' || e.type == 'mouseenter') e.relatedTarget = e.fromElement;
    if (e.type == 'mouseout' || e.type == 'mouseleave') e.relatedTarget = e.toElement;

    if (e.pageX == null && e.clientX != null) {
        var html = document.documentElement;
        var body = document.body;

        e.pageX = e.clientX + (html.scrollLeft || body && body.scrollLeft || 0);
        e.pageX -= html.clientLeft || 0;

        e.pageY = e.clientY + (html.scrollTop || body && body.scrollTop || 0);
        e.pageY -= html.clientTop || 0;
    }

    if (!e.which && e.button) {
        e.which = e.button & 1 ? 1 : (e.button & 2 ? 3 : (e.button & 4 ? 2 : 0));
    }
}

return e;
}

```

Эта функция может быть полена, если не используются JavaScript-фреймворки, в которых есть свои средства сглаживания кросс-браузерных различий.

Прокрутка: событие scroll

Событие `onscroll` происходит, когда элемент прокручивается.

В отличие от события `onwheel` (колесико мыши), его могут генерировать только прокручиваемые элементы или окно `window`. Но зато оно генерируется всегда, при любой прокрутке, не обязательно «мышиной».

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

```

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

```

В действии: Текущая прокрутка = **прокрутите окно**

Каких-либо особенностей события здесь нет, разве что для его использования нужно отлично представлять, как получить текущее значение прокрутки или прокрутить документ. Об этом мы говорили ранее, в главе [Размеры и прокрутка элементов](#).

Некоторые области применения `onscroll`:

- Показ дополнительных элементов навигации при прокрутке.
- Подгрузка и инициализация элементов интерфейса, ставших видимыми после прокрутки.

вашему вниманию предлагаются несколько задач, которые вы можете решить сами или посмотреть использование onscroll на их примере.

✓ Задачи

Аватар наверху при прокрутке

важность: 5

Сделайте так, чтобы при прокрутке ниже элемента #avatar (картинка с Винни-Пухом) — он продолжал показываться в левом-верхнем углу.

При прокрутке вверх — должен возвращаться на обычное место.

Прокрутите вниз, чтобы увидеть:

Шапка

Персонажи:

- Винни-Пух
- Ослик Иа
- Сова
- Кролик

Винни-Пух



Винни-Пух (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (шикль не имеет общего

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Кнопка вверх-вниз

важность: 3

Создайте кнопку навигации, которая помогает при прокрутке страницы.

Работать должна следующим образом:

- Пока страница промотана меньше чем на высоту экрана вниз — кнопка не видна.
- При промотке страницы вниз больше, чем на высоту экрана, появляется кнопка «стрелка

вверх».

- При нажатии на нее страница прыгает вверх, но не только. Дополнительно, кнопка меняется на «стрелка вниз» и при клике возвратит на старое место. Если же в этом состоянии посетитель сам прокрутит вниз больше, чем один экран, то она вновь изменится на «стрелка вверх».

Должен получиться удобный навигационный помощник.

Посмотрите, как оно должно работать, в ифрейме ниже. Прокрутите ифрейм, навигационная стрелка появится слева-сверху.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43  
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63  
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83  
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102  
103 104 105 106 107 108 109 110 111 112 113 114 115 116  
117 118 119 120 121 122 123 124 125 126 127 128 129 130  
131 132 133 134 135 136 137 138 139 140 141 142 143 144  
145 146 147 148 149 150 151 152 153 154 155 156 157 158  
159 160 161 162 163 164 165 166 167 168 169 170 171 172  
173 174 175 176 177 178 179 180 181 182 183 184 185 186
```

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Загрузка видимых изображений

важность: 4

Задача, которая описана ниже, демонстрирует результативный метод оптимизации страницы.

С целью экономии трафика и более быстрой загрузки страницы изображения на ней заменяются на «макеты».

Вместо такого изображения:

```

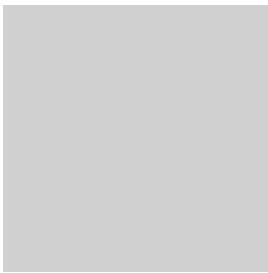
```



Стоит вот такое:

```

```



То есть настоящий URL находится в атрибуте `realsrc` (название атрибута можно выбрать любое). А в `src` поставлен серый GIF размера 1x1, и так как `width/height` правильные, то он растягивается, так что вместо изображения виден серый прямоугольник.

При этом, чтобы браузер загрузил изображение, нужно заменить значение `src` на то, которое находится в `realsrc`.

Если страница большая, то замена больших изображений на такие макеты существенно убыстряет полную загрузку страницы. Это особенно заметно в случае, когда на странице много анонсов новостей с картинками или изображений товаров, из которых многие находятся за пределами прокрутки.

Кроме того, для мобильных устройств JavaScript может подставлять URL уменьшенного варианта картинки.

Напишите код, который при прокрутке окна загружает ставшие видимыми изображения.

То есть, как только изображение попало в видимую часть документа — в `src` нужно прописать правильный URL из `realsrc`.

Пример работы вы можете увидеть в `iframe` ниже, если прокрутите его:

Тексты и картинки взяты с сайта <http://etoday.ru>.

Все изображения с `realsrc` загружаются, когда становятся видимыми.

Космопорт Америка \ Architecture

Будущее уже сейчас! Скоро фраза из фантастического фильма "флипнуть до космопорта" станет реальностью. По крайней мере вторую ее часть человечество обеспечило. В октябре состоялась официальная церемония открытия космопорта «Америка», первой в мире коммерческой площадки частных космических полетов. Космопорт открылся в пустыне штата Нью-Мексико. Проект реализован английским бюро Foster and Partners. Космопорт включает в себя зал ожидания подготовки к полетам, диспетчерский пункт и ангар. Также обеспечена взлетно-посадочная полоса длиной в три километра.



Особенности реализации:

- При начальной загрузке некоторые изображения должны быть видны сразу, до прокрутки. код должен это учитывать.
- Некоторые изображения могут быть обычными, без `realsrc`. Их код не должен трогать вообще.
- Также код не должен перегружать уже показанное изображение.
- Желательно предусмотреть загрузку изображений не только видимых сейчас, но и на страницу вперед и назад от текущего места.

P.S. Горизонтальной прокрутки нет.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Клавиатура: `keyup`, `keydown`, `keypress`

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.

[Смотреть пример онлайн](#) ↗

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

События `keydown` и `keyup`

События `keydown`/`keyup` происходят при нажатии/отпусканье 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.

Скан-коды

Для буквенно-цифровых клавиш есть очень простое правило: 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)`.

Для других символов, в частности, знаков пунктуации, есть таблица кодов, которую можно взять, например, из статьи Джона Уолтерса: [JavaScript Madness: Keyboard Events](#) ↗, или же можно нажать на нужную клавишу на [тестовом стенде](#) и получить код.

Когда-то в этих кодах была масса кросс-браузерных несовместимостей. Сейчас все 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.

Таблица несовместимостей:

Клавиша	Firefox	Остальные браузеры
;	59	186
=	107	187
-	109	189

Остальные коды одинаковы, код для нужного символа будет в тестовом стенде.

Событие 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:

```
// event.type должен быть keypress
function getChar(event) {
  if (event.which == null) { // IE
    if (event.keyCode < 32) return null; // спец. символ
    return String.fromCharCode(event.keyCode)
  }

  if (event.which != 0 && event.charCode != 0) { // все кроме IE
    if (event.which < 32) return null; // спец. символ
    return String.fromCharCode(event.which); // остальные
  }

  return null; // спец. символ
}
```

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

1. Во всех браузерах, кроме IE, у события keypress есть свойство keyCode, которое содержит код символа.
2. Браузер IE для keypress не устанавливает keyCode, а вместо этого он записывает код символа в keyCode (в keydown/keyup там хранится скан-код).
3. Также 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`.

⚠ 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'. Лучше использовать ту, что приведена выше.

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">
```

При тестировании на стенде вы можете заметить, что отмена действия браузера при keydown также предотвращает само событие keypress.

⚠ При keydown/keypress значение ещё старое

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

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

Вводите символы

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

Отмена любых действий

Отменять можно не только символ, а любое действие клавиш.

Например:

- При отмене `Backspace` — символ не удалится.
- При отмене `PageDown` — страница не прокрутится.
- При отмене `Tab` — курсор не перейдёт на следующее поле.

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:

```
<input id="only-upper" type="text" size="2">
<script>
  document.getElementById('only-upper').onkeypress = function(e) {
    // спец. сочетание - не обрабатываем
    if (e.ctrlKey || e.altKey || e.metaKey) return;

    var char = getChar(e);

    if (!char) return; // спец. символ - не обрабатываем

    this.value = char.toUpperCase();

    return false;
  };
</script>
```

Невосместимости

Некоторые несовместимости в порядке срабатывания клавиатурных событий (когда что) ещё

существуют.

Стоит иметь в виду три основных категории клавиш, работа с которыми отличается.

Категория	События	Описание
Печатные клавиши S 1 ,	keydown keypress keyup	Нажатие вызывает keydown и keypress. Когда клавишу отпускают, срабатывает keyup. Исключение — CapsLock под MacOS, с ним есть проблемы: <ul style="list-style-type: none">• В Safari/Chrome/Opera: при включении только keydown, при отключении только keyup.• В Firefox: при включении и отключении только keydown.
Специальные клавиши Alt Esc ↑	keydown keyup	Нажатие вызывает keydown. Когда клавишу отпускают, срабатывает keyup. Некоторые браузеры могут дополнительно генерировать и keypress, например IE для Esc.
Сочетания с печатной клавишей Alt+E Ctrl+Y Cmd+1	keydown keypress? keyup	На практике это не доставляет проблем, так как для специальных клавиш мы всегда используем keydown/keyup. Браузеры под Windows — не генерируют keypress, браузеры под MacOS — генерируют. Кроме того, если сочетание вызвало браузерное действие или диалог («Сохранить файл», «Открыть» и т.п., ряд диалогов можно отменить при keydown), то может быть только keydown.

Общий вывод можно сделать такой:

- Обычные символы работают везде корректно.
- CapsLock под MacOS ведёт себя плохо, не стоит ставить на него обработчики вообще.
- Для других специальных клавиш и сочетаний с ними следует использовать только keydown.

Автоповтор

При долгом нажатии клавиши возникает *автоповтор*. По стандарту, должны генерироваться многоократные события keydown (+keypress), и вдобавок стоять свойство `repeat=true` у объекта события.

То есть поток событий должен быть такой:

```
keydown  
keypress  
keydown  
keypress  
..повторяется, пока клавиша не отжата...  
keyup
```

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

```
keydown  
keypress  
keyup  
keydown  
keypress  
keyup  
..повторяется, пока клавиша не отжата...  
keyup
```

...А Chrome под MacOS не генерирует keypress. В общем, «зоопарк».

Полагаться можно только на keydown при каждом автонажатии и keyup по отпусканью клавиши.

Итого

Ряд рецептов по итогу этой главы:

1. Для реализации горячих клавиш, включая сочетания — используем keydown. Скан-код будет в keyCode, почти все скан-коды кросс-браузерны, кроме нескольких пунктуационных, перечисленных в таблице выше.
2. Если нужен именно символ — используем keypress. При этом функция getChar позволит получить символ и отфильтровать лишние срабатывания. Гарантированно получать символ можно только при нажатии обычных клавиш, если речь о сочетаниях с модификаторами, то keypress не всегда генерируется.
3. Ловля CapsLock глючит под MacOS. Её можно организовать при помощи проверки navigator.userAgent и navigator.platform, а лучше вообще не трогать эту клавишу.

Распространённая ошибка — использовать события клавиатуры для работы с полями ввода в формах.

Это нежелательно. События клавиатуры предназначены именно для работы с клавиатурой. Да, их можно использовать для проверки ввода в <input>, но будут недочёты. Например, текст может быть вставлен мышкой, при помощи правого клика и меню, без единого нажатия клавиши. И как нам помогут события клавиатуры?

Некоторые мобильные устройства также не генерируют keypress/keydown, а сразу вставляют текст в поле. Обработать ввод на них при помощи клавиатурных событий нельзя.

Далее мы разберём [события для элементов форм](#), которые позволяют работать с вводом в формы правильно.

их можно использовать как отдельно от событий клавиатуры, так и вместе с ними.

✓ Задачи

Поле только для цифр

важность: 5

При помощи событий клавиатуры сделайте так, чтобы в поле можно было вводить только цифры. Пример ниже.

Введите ваш возраст:

В поле должны нормально работать специальные клавиши `Delete` / `Backspace` и сочетания с `Ctrl` / `Alt` / `Cmd`.

P.S. Конечно, при помощи альтернативных способов ввода (например, вставки мышью), посетитель всё же может ввести что угодно.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Отследить одновременное нажатие

важность: 3

Создайте функцию `runOnKeys(func, code1, code2, ... code_n)`, которая запускает `func` при одновременном нажатии клавиш со скан-кодами `code1, code2, ..., code_n`.

Например, код ниже выведет `alert` при одновременном нажатии клавиш "Q" и "W" (в любом

реестре, в любои раскладке)

```
runOnKeys(  
  function() { alert("Привет!") },  
  "Q".charCodeAt(0),  
  "W".charCodeAt(0)  
)
```

[Демо в новом окне ↗](#)

[К решению](#)

Загрузка документа: **DOMContentLoaded, load, beforeunload, unload**

Процесс загрузки HTML-документа, условно, состоит из трёх стадий:

- **DOMContentLoaded** — браузер полностью загрузил HTML, и построил DOM-дерево.
- **load** — браузер загрузил все ресурсы.
- **beforeunload/unload** — уход со страницы.

Все эти стадии очень важны. На каждую можно повесить обработчик, чтобы совершить полезные действия:

- **DOMContentLoaded** — означает, что все DOM-элементы разметки уже созданы, можно их искать, вешать обработчики, создавать интерфейс, но при этом, возможно, ещё не додгрузились какие-то картинки или стили.
- **load** — страница и все ресурсы загружены, используется редко, обычно нет нужды ждать этого момента.
- **beforeunload/unload** — можно проверить, сохранил ли посетитель изменения, уточнить, действительно ли он хочет покинуть страницу.

Далее мы рассмотрим важные детали этих событий.

DOMContentLoaded

Событие **DOMContentLoaded** поддерживается во всех браузерах, кроме IE8-. Про поддержку аналогичного функционала в старых IE мы поговорим в конце главы.

Обработчик на него вешается только через `addEventListener`:

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

Пример:

```
<script>
  function ready() {
    alert( 'DOM готов' );
    alert( "Размеры картинки: " + img.offsetWidth + "x" + img.offsetHeight );
  }

  document.addEventListener("DOMContentLoaded", ready);
</script>


```

В примере выше размеры обработчик `DOMContentLoaded` сработает сразу после загрузки документа, не дожидаясь получения картинки.

Поэтому на момент вывода `alert` и сама картинка будет невидна и её размеры — неизвестны (кроме случая, когда картинка взята из кеша браузера).

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.

Исключением являются скрипты с атрибутами `async` и `defer`, которые подгружаются асинхронно.

Побочный эффект: если на странице подключается скрипт с внешнего ресурса (к примеру, реклама), и он тормозит, то событие `DOMContentLoaded` и связанные с ним действия могут сильно задержаться.

Современные системы рекламы используют атрибут `async`, либо вставляют скрипты через DOM: `document.createElement('script')...`, что работает так же как `async`: такой скрипт выполняется полностью независимо от страницы и от других скриптов — сам ничего не ждёт и ничего не блокирует.

DOMContentLoaded и стили

Внешние стили никак не влияют на событие `DOMContentLoaded`. Но есть один нюанс.

Если после стиля идёт скрипт, то этот скрипт обязан дождаться, пока стиль загрузится:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // сработает после загрузки style.css
</script>
```

Такое поведение прописано в стандарте. Его причина — скрипт может захотеть получить информацию со страницы, зависящую от стилей, например, ширину элемента, и поэтому обязан дождаться загрузки `style.css`.

Побочный эффект — так как событие DOMContentLoaded будет ждать выполнения скрипта, то оно подождёт и загрузки стилей, которые идут перед <script>.

Автозаполнение

Firefox/Chrome/Opera 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 автозаполнение не сработает до полной их загрузки.

Конечно, это довод в пользу того, чтобы не задерживать DOMContentLoaded, в частности — использовать у скриптов атрибуты `async` и `defer`.

window.onload

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

Пример ниже выведет `alert` лишь после полной загрузки окна, включая IFRAME и картинку:

```
<script>
  window.onload = function() {
    alert( 'Документ и все ресурсы загружены' );
  };
</script>
<iframe src="https://example.com/" style="height:60px"></iframe>

```

window.onunload

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

Это позволяет другое событие — `window.onbeforeunload`, которое поэтому используется гораздо чаще.

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 большей безопасности посетителя, чтобы его нельзя было ввести в заблуждение сообщением.

Эмуляция DOMContentLoaded для IE8-

Прежде чем что-то эмулировать, заметим, что альтернативой событию onDOMContentLoaded является вызов функции `init` из скрипта в самом конце BODY, когда основная часть DOM уже готова:

```
<body>
...
<script>
  init();
</script>
</body>
```

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

Мини-скрипт documentReady

Если вы всё же хотите использовать onDOMContentLoaded кросс-браузерно, то нужно либо подключить какой-нибудь фреймворк — почти все предоставляют такой функционал, либо использовать функцию из мини-библиотеки [jquery.documentReady.js ↗](#).

Несмотря на то, что в названии содержится слово «jquery», эта библиотечка не требует [jQuery ↗](#). Наоборот, она представляет собой единственную функцию с названием `$`, вызов которой `$(callback)` добавляет обработчик `callback` на DOMContentLoaded (можно вызывать много раз), либо, если документ уже загружен — выполняет его тут же.

Пример использования:

```
<script src="https://js.cx/script/jquery.documentReady.js"></script>

<script>
$(function() {
  alert( "DOMContentLoaded" );
});
</script>


<div>Текст страницы</div>
```

Здесь `alert` сработает до загрузки картинки, но после создания DOM, в частности, после появления

текста. И так оудет для всех браузеров, включая даже очень старые IE.

❶ Как именно эмулируется DOMContentLoaded?

Технически, эмуляция DOMContentLoaded для старых IE осуществляется очень забавно.

Основной приём — это попытка прокрутить документ вызовом:

```
document.documentElement.doScroll("left");
```

Метод `doScroll` работает только в IE и «методом тыка» было обнаружено, что он бросает исключение, если DOM не полностью создан.

Поэтому библиотека пытается вызвать прокрутку, если не получается — через `setTimeout(..., 1)` пытается прокрутить его ещё раз, и так до тех пор, пока действие не перестанет вызывать ошибку. На этом этапе документ считается загрузившимся.

Внутри фреймов и в очень старых браузерах такой подход может ошибаться, поэтому дополнительно ставится резервный обработчик на `onload`, чтобы уж точно сработал.

Итого

- Самое востребованное событие из описанных — без сомнения, `DOMContentLoaded`. Многие страницы сделаны так, что инициализируют интерфейсы именно по этому событию.

Это удобно, ведь можно в `<head>` написать скрипт, который будет запущен в момент, когда все DOM-элементы доступны.

С другой стороны, следует иметь в виду, что событие `DOMContentLoaded` будет ждать не только, собственно, HTML-страницу, но и внешние скрипты, подключенные тегом `<script>` без атрибутов `defer`/`async`, а также стили перед такими скриптами.

Событие `DOMContentLoaded` не поддерживается в IE8-, но почти все фреймворки умеют его эмулировать. Если нужна отдельная функция только для кросс-браузерного аналога `DOMContentLoaded` — можно использовать [jquery.documentReady.js ↗](#).

- Событие `window.onload` используют редко, поскольку обычно нет нужды ждать подгрузки всех ресурсов. Если же нужен конкретный ресурс (картинка или iframe), то можно поставить событие `onload` непосредственно на нём, мы посмотрим, как это сделать, далее.
- Событие `window.onunload` почти не используется, как правило, оно бесполезно — мало что можно сделать, зная, что окно браузера прямо сейчас закроется.
- Гораздо чаще применяется `window.onbeforeunload` — это де-факто стандарт для того, чтобы проверить, сохранил ли посетитель данные, действительно ли он хочет покинуть страницу. В системах редактирования документов оно используется повсеместно.

Загрузка скриптов, картинок, фреймов: `onload` и `onerror`

Браузер позволяет откладывать загрузку внешних ресурсов — скриптов, изображений, картинок и других.

Для этого есть два события:

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

Загрузка SCRIPT

Рассмотрим следующую задачу.

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

Подгрузить внешний скрипт — достаточно просто:

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

document.body.appendChild(script);
```

...Но как после подгрузки выполнить функцию, которая объявлена в этом скрипте? Для этого нужно отловить момент окончания загрузки и выполнения тега `<script>`.

script.onload

Главным помощником станет событие `onload`. Оно сработает, когда скрипт загрузился и выполнился.

Например:

```
var script = document.createElement('script');
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"
document.body.appendChild(script);

script.onload = function() {
    // после выполнения скрипта становится доступна функция _
    alert(_); // её код
}
```

Это даёт возможность, как в примере выше, получить переменные из скрипта и выполнять объявленные в нём функции.

...А что, если загрузка скрипта не удалась? Например, такого скрипта на сервере нет (ошибка 404) или сервер «упал» (ошибка 500).

Такую ситуацию тоже нужно как-то обрабатывать, хотя бы сообщить посетителю о возникшей проблеме.

script.onerror

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

Например, сделаем запрос заведомо отсутствующего скрипта:

```
var script = document.createElement('script');
script.src = "https://example.com/404.js"
document.body.appendChild(script);

script.onerror = function() {
  alert( "Ошибка: " + this.src );
};
```

IE8: script.onreadystatechange

Примеры выше работают во всех браузерах, кроме IE8-.

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:

```
var script = document.createElement('script');
script.src = "https://code.jquery.com/jquery.js";
document.documentElement.appendChild(script);

script.onreadystatechange = function() {
  alert(this.readyState); // loading -> loaded -> complete
};
```

Скрипт с ошибкой:

```
var script = document.createElement('script');
script.src = "http://ajax.googleapis.com/404.js";
document.documentElement.appendChild(script);

script.onreadystatechange = function() {
  alert(this.readyState); // loading -> loaded
}
```

Обратим внимание на две особенности:

- **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 IE8- поймать загрузку (или ошибку загрузки) — это повесить обработчик на событие `onreadystatechange`, который будет срабатывать и на стадии `complete` и на стадии `loaded`. Так как скрипт может быть ещё не выполнен к этому моменту, то вызов функции лучше сделать через `setTimeout(..., 0)`.

Пример ниже вызывает `afterLoad` после загрузки скрипта и работает только в IE:

```
var script = document.createElement('script');
script.src = "https://code.jquery.com/jquery.js";
document.documentElement.appendChild(script);

function afterLoad() {
  alert("Загрузка завершена: " + typeof(jQuery));
}

script.onreadystatechange = function() {
  if (this.readyState == "complete") { // на случай пропуска loaded
    afterLoad(); // (2)
  }

  if (this.readyState == "loaded") {
    setTimeout(afterLoad, 0); // (1)

    // убираем обработчик, чтобы не сработал на complete
    this.onreadystatechange = null;
  }
}
```

Вызов (1) выполнится при первой загрузке скрипта, а (2) — при второй, когда он уже будет в кеше, 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-браузерной обработки загрузки скрипта или ее ошибки поставим обработчик на все три события: `onload`, `onerror`, `onreadystatechange`.

Пример ниже выполняет функцию `afterLoad` после загрузки скрипта или при ошибке.

Работает во всех браузерах:

```
var script = document.createElement('script');
script.src = "https://code.jquery.com/jquery.js";
document.documentElement.appendChild(script);

function afterLoad() {
  alert("Загрузка завершена: " + typeof(jQuery));
}

script.onload = script.onerror = function() {
  if (!this.executed) { // выполнится только один раз
    this.executed = true;
    afterLoad();
  }
};

script.onreadystatechange = function() {
  var self = this;
  if (this.readyState == "complete" || this.readyState == "loaded") {
    setTimeout(function() {
      self.onload()
    }, 0); // сохранить "this" для onload
  }
};
```

Загрузка других ресурсов

Поддержка этих событий для других типов ресурсов различна:

``, `<link>` (стили)

Поддерживает `onload/onerror` во всех браузерах.

`<iframe>`

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

Обратим внимание, что если `<iframe>` загружается с того же домена, то можно, используя `iframe.contentWindow.document` получить ссылку на документ и поставить обработчик `DOMContentLoaded`. А вот если `<iframe>` — с другого домена, то так не получится, однако сработает `onload`.

Итого

В этой статье мы рассмотрели события `onload/onerror` для ресурсов.

Их можно обобщить, разделив на рецепты:

Отловить загрузку скрипта (включая ошибку)

Ставим обработчики на onload + onerror + (для IE8-) onreadystatechange, как указано в рецепте выше

Отловить загрузку картинки или стиля <link>

Ставим обработчики на onload + onerror

```
var img = document.createElement('img');
img.onload = function() { alert("Успех "+this.src); };
img.onerror = function() { alert("Ошибка "+this.src); };
img.src = ...;
```

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

Чтобы работало в IE8-, 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.

✓ Задачи

Красивый «ALT»

важность: 5

Обычно, до того как изображение загрузится (или при отключенных картинках), посетитель видит пустое место с текстом из «ALT». Но этот атрибут не допускает HTML-форматирования.

При мобильном доступе скорость небольшая, и хочется, чтобы посетитель сразу видел красивый текст.

Реализуйте «красивый» (HTML) аналог alt при помощи CSS/JavaScript, который затем будет заменён картинкой сразу же как только она загрузится. А если загрузка не состоится — то не заменён.

Демо: (нажмите «перезагрузить», чтобы увидеть процесс загрузки и замены)

Перезагрузить ифрейм



Картинки для bing специально нет, так что текст остается «как есть».

Исходный документ содержит разметку текста и ссылки на изображения.

[Открыть песочницу для задачи.](#)

[К решению](#)

Загрузить изображения с коллбэком

важность: 4

Создайте функцию `preloadImages(sources, callback)`, которая предзагружает изображения из массива `sources`, и после загрузки вызывает функцию `callback`.

Пример использования:

```
addScripts(["1.jpg", "2.jpg", "3.jpg"], callback);
```

Если вдруг возникает ошибка при загрузке — считаем такое изображение загруженным, чтобы не ломать поток выполнения.

Такая функция может полезна, например, для фоновой загрузки картинок в онлайн-галерею.

В исходном документе содержатся ссылки на картинки, а также код для проверки, действительно ли изображения загрузились. Он должен выводить «0», затем «300».

[Открыть песочницу для задачи.](#)

[К решению](#)

Скрипт с коллбэком

важность: 4

Создайте функцию `addScript(src, callback)`, которая загружает скрипт с данным `src`, и после его загрузки и выполнения вызывает функцию `callback`.

Скрипт может быть любым, работа функции не должна зависеть от его содержимого.

Пример использования:

```
// go.js содержит функцию go()
addScript("go.js", function() {
  go();
});
```

Ошибки загрузки обрабатывать не нужно.

[Открыть песочницу для задачи.](#)

Скрипты с коллбэком

важность: 5

Создайте функцию `addScripts(scripts, callback)`, которая загружает скрипты из массива `scripts`, и после загрузки и выполнения их всех вызывает функцию `callback`.

Скрипт может быть любым, работа функции не должна зависеть от его содержимого.

Пример использования:

```
addScripts(["a.js", "b.js", "c.js"], function() { a() });
/* функция a() описана в a.js и использует b.js,c.js */
```

- Ошибки загрузки обрабатывать не нужно.
- Один скрипт не ждёт другого. Они все загружаются, а по окончании вызывается обработчик `callback`.

Исходный содержит скрипты `a.js`, `b.js`, `c.js`:

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Формы, элементы управления

Особые свойства, методы и события для работы с формами `<form>` и элементами ввода: `<input>`, `<select>` и другими.

Навигация и свойства элементов формы

Элементы управления, такие как `<form>`, `<input>` и другие имеют большое количество своих важных свойств и ссылок.

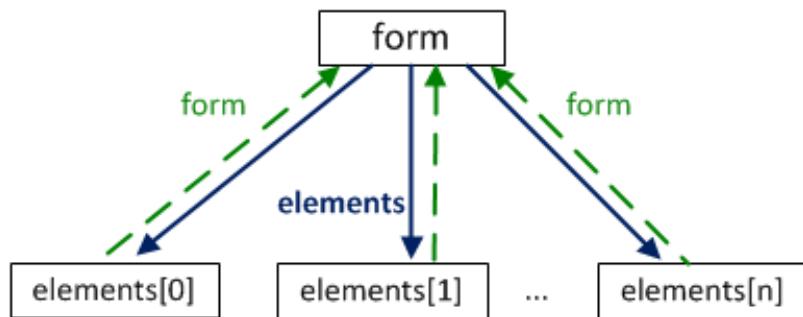
Псевдомассив `form.elements`

Элементы FORM можно получить по имени или номеру, используя свойство `document.forms[name/index]`.

Например:

```
document.forms.my -- форма с именем 'my'  
document.forms[0] -- первая форма в документе
```

Любой элемент формы form можно получить аналогичным образом, используя свойство form.elements.



Например:

```
<body>  
  <form name="my">  
    <input name="one" value="1">  
    <input name="two" value="2">  
  </form>  
  
<script>  
  var form = document.forms.my; // можно document.forms[0]  
  
  var elem = form.elements.one; // можно form.elements[0]  
  
  alert( elem.value ); // "один"  
</script>  
</body>
```

Может быть несколько элементов с одинаковым именем. В таком случае form.elements[name] вернет коллекцию элементов, например:

```
<body>  
  <form>  
    <input type="radio" name="age" value="10">  
    <input type="radio" name="age" value="20">  
  </form>  
  
<script>  
  var form = document.forms[0];  
  
  var elems = form.elements.age;  
  
  alert(elems[0].value); // 10, первый input  
</script>  
</body>
```

Эти ссылки не зависят от окружающих тегов. Элемент может быть «зарыт» где-то глубоко в форме, но он всё равно доступен через form.elements.

Свойство elements также есть у элементов <fieldset>. Вот пример:

```
<body>
  <form>
    <fieldset name="set">
      <legend>fieldset</legend>
      <input name="text" type="text">
    </fieldset>
  </form>

  <script>
    var form = document.forms[0];

    alert( form.elements.text ); // INPUT
    alert( form.elements.set.elements.text ); // INPUT
  </script>
</body>
```

Спецификация: [HTML5 Forms](#).

⚠ Доступ form.name тоже работает, но с особенностями

Получить доступ к элементам формы можно не только через `form.elements[name/index]`, но и проще: `form[index/name]`.

Этот способ короче, так как обладает одной неприятной особенностью: если к элементу обратиться по его `name`, а потом свойство `name` изменить, то он по-прежнему будет доступен под старым именем.

Звучит странно, поэтому посмотрим на примере.

```
<form name="myform">
  <input name="text">
</form>

<script>
  var form = document.forms.myform;

  alert( form.elements.text == form.text ); // true, это тот самый INPUT

  form.text.name = "new-name"; // меняем name ему

  // нет больше элемента с таким именем
  alert( form.elements.text ); // undefined

  alert( form.text ); // INPUT (а должно быть undefined!)
</script>
```

Ссылка на форму element.form

По элементу можно получить его форму, используя свойство `element.form`.

Пример:

```
<body>
<form>
  <input type="text" name="surname">
</form>

<script>
var form = document.forms[0];

var elem = form.elements.surname;

alert(elem.form == form); // true
</script>
</body>
```

Познакомиться с другими свойствами элементов можно в спецификации [HTML5 Forms](#).

Элемент label

Элемент `label` — один из самых важных в формах.

Клик на `label` засчитывается как фокусировка или 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.

Есть два способа показать, какой элемент относится к `label`:

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

```
<table>
  <tr>
    <td>
      <label for="agree">Согласен с правилами</label>
    </td>
    <td>
      <input id="agree" type="checkbox">
    </td>
  </tr>
  <tr>
    <td>
      <label for="not-a-robot">Я не робот</label>
    </td>
    <td>
      <input id="not-a-robot" type="checkbox">
    </td>
  </tr>
</table>
```

Согласен с правилами

Я не робот

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

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

Кликни меня

Элементы `input` и `textarea`

Для большинства типов `input` значение ставится/читается через свойство `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, и не меняется при изменении значения.

Исключения — `input type="checkbox"` и `input type="radio"`

Текущее «отмеченное» состояние для checkbox и radio находится в свойстве `checked` (true/false).

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

Элементы select и option

Селект в JavaScript можно установить двумя путями: поставив значение `select.value`, либо установив свойство `select.selectedIndex` в номер нужной опции.:

```
select.selectedIndex = 0; // первая опция
```

Установка `selectedIndex = -1` очистит выбор.

Список элементов-опций доступен через `select.options`.

Если `select` допускает множественный выбор (атрибут `multiple`), то значения можно получить/установить, сделав цикл по `select.options`. При этом выбранные опции будут иметь свойство `option.selected = true`.

Пример:

```
<form name="form">  
  <select name="genre" multiple>  
    <option value="blues" selected>Мягкий блюз</option>  
    <option value="rock" selected>Жёсткий рок</option>  
    <option value="classic">Классика</option>  
  </select>  
</form>  
  
<script>  
var form = document.forms[0];  
var select = form.elements.genre;  
  
for (var i = 0; i < select.options.length; i++) {  
  var option = select.options[i];  
  if(option.selected) {  
    alert( option.value );  
  }  
}  
</script>
```

Спецификация: [the select element ↗](#).

new Option

В стандарте [the option element ↗](#) 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 ↗](#)):

selected

выбрана ли опция

index

номер опции в списке селекта

text

Текстовое содержимое опции (то, что видит посетитель).

Итого

Свойства для навигации по формам:

document.forms

Форму можно получить как `document.forms[name/index]`.

form.elements

Элементы в форме: `form.elements[name/index]`. Каждый элемент имеет ссылку на форму в свойстве `form`. Свойство `elements` также есть у `<fieldset>`.

Значение элементов читается/ставится через `value` или `checked`.

Для элемента `select` можно задать опцию по номеру через `select.selectedIndex` и перебрать опции через `select.options`. При этом выбранные опции (в том числе при мультиселекте) будут иметь свойство `option.selected = true`.

✓ Задачи

Добавьте опцию к селекту

важность: 5

Есть селект:

```
<select>
  <option value="Rock">Рок</option>
  <option value="Blues" selected>Блюз</option>
</select>
```

При помощи JavaScript:

1. Выведите значение и текст текущей выбранной опции.
2. Добавьте опцию: `<option value="Classic">Классика</option>`.
3. Сделайте её выбранной.

[К решению](#)

Фокусировка: focus/blur

Говорят, что элемент «получает фокус», когда посетитель фокусируется на нём. Обычно фокусировка автоматически происходит при нажатии на элементе мышкой, но также можно перейти на нужный элемент клавиатурой — через клавишу `Tab`, нажатие пальцем на планшете и так далее.

Момент получения фокуса и потери очень важен.

При получении фокуса мы можем подгрузить данные для автодополнения, начать отслеживать изменения. При потере — проверить данные, которые ввёл посетитель.

Кроме того, иногда полезно «вручную», из языка перевести фокус на нужный элемент, например, на поле в динамически созданной форме.

События focus/blur

Событие `focus` вызывается тогда, когда пользователь фокусируется на элементе, а `blur` — когда фокус исчезает, например посетитель кликает на другом месте экрана.

Давайте сразу посмотрим на них в деле, используем для проверки («валидации») введённых в форму значений.

В примере ниже:

- Обработчик `onblur` проверяет, что в поле введено число, если нет — показывает ошибку.
- Обработчик `onfocus`, если текущее состояние поля ввода — «ошибка» — скрывает её (потом при `onblur` будет повторная проверка).

В примере ниже, если набрать что-нибудь в поле «возраст» и завершить ввод, нажав `Tab` или кликнув в другое место страницы, то введённое значение будет автоматически проверено:

```
<style> .error { border-color: red; } </style>

Введите ваш возраст: <input type="text" id="input">

<div id="error"></div>

<script>
input.onblur = function() {
  if (isNaN(this.value)) { // введено не число
    // показать ошибку
    this.className = "error";
    error.innerHTML = 'Вы ввели не число. Исправьте, пожалуйста.';
  }
};

input.onfocus = function() {
  if (this.className == 'error') { // сбросить состояние "ошибка", если оно есть
    this.className = "";
    error.innerHTML = "";
  }
};
</script>
```

Введите ваш возраст:

Методы focus/blur

Методы с теми же названиями переводят/уводят фокус с элемента.

Для примера модифицируем пример выше, чтобы при неверном вводе посетитель просто не мог уйти с элемента:

```

<style>
  .error {
    background: red;
  }
</style>

<div>Возраст:
  <input type="text" id="age">
</div>

<div>Имя:
  <input type="text">
</div>

<script>
  age.onblur = function() {
    if (isNaN(this.value)) { // введено не число
      // показать ошибку
      this.classList.add("error");
      //... и вернуть фокус обратно
      age.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>

```

Возраст:

Имя:

Этот пример работает во всех браузерах, кроме Firefox ([ошибка ↗](#)).

Если ввести что-то нецифровое в поле «возраст», и потом попытаться табом или мышкой перейти на другой `<input>`, то обработчик `onblur` вернёт фокус обратно.

Обратим внимание — если из `onblur` сделать `event.preventDefault()`, то такого же эффекта не будет, потому что `onblur` срабатывает уже *после* того, как элемент потерял фокус.

HTML5 и CSS3 вместо focus/blur

Прежде чем переходить к более сложным примерам, использующим JavaScript, мы рассмотрим три примера, когда его использовать не надо, а достаточно современного HTML/CSS.

Подсветка при фокусировке

Стилизация полей ввода может быть решена средствами CSS (CSS2.1), а именно — селектором `:focus`:

```
<style>
input:focus {
    background: #FA6;
    outline: none; /* убрать рамку */
}
</style>
<input type="text">
```

<p>Селектор :focus выделит элемент при фокусировке на нем и уберёт рамку, которой браузер выделяет этот



Селектор :focus выделит элемент при фокусировке на нем и уберёт рамку, которой браузер выделяет этот элемент по умолчанию.

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

Автофокус

При загрузке страницы, если на ней существует элемент с атрибутом autofocus — браузер автоматически фокусируется на этом элементе. Работает во всех браузерах, кроме IE9-.

```
<input type="text" name="search" autofocus>
```

Если нужны старые IE, то же самое может сделать JavaScript:

```
<input type="text" name="search">
<script>
    document.getElementsByName('search')[0].focus();
</script>
```

Как правило, этот атрибут используется при изначальной загрузке, для страниц поиска и так далее, где главный элемент очевиден.

Плейсхолдер

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

Во всех браузерах, кроме IE9-, это реализуется специальным атрибутом placeholder:

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



В некоторых браузерах этот текст можно стилизовать.

```
<style>
.my::-webkit-input-placeholder {
  color: red;
  font-style: italic;
}
.my::-moz-input-placeholder {
  color: red;
  font-style: italic;
}
.my::-ms-input-placeholder {
  color: red;
  font-style: italic;
}
</style>

<input class="my" type="text" placeholder="E-mail">
Стилизованный плейсхолдер
```

 Стилизованный плейсхолдер

Разрешаем фокус на любом элементе: tabindex

По умолчанию не все элементы поддерживают фокусировку.

Перечень элементов немного рознится от браузера к браузеру, например, список для IE описан в [MSDN](#), одно лишь верно всегда — заведомо поддерживают focus/blur те элементы, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д.

С другой стороны, на элементах для форматирования, таких как <div>, , <table> — по умолчанию сфокусироваться нельзя. Впрочем, существует способ включить фокусировку и для них.

В HTML есть атрибут tabindex.

Его основной смысл — это указать номер элемента при переборе клавишей Tab.

То есть, если есть два элемента, первый имеет `tabindex="1"`, а второй `tabindex="2"`, то нажатие Tab при фокусе на первом элементе — переведёт его на второй.

Исключением являются специальные значения:

- `tabindex="0"` делает элемент всегда последним.
- `tabindex="-1"` означает, что клавиша Tab будет элемент игнорировать.

Любой элемент поддерживает фокусировку, если у него есть `tabindex`.

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

```

Кликните на первый элемент списка и нажмите Tab. Внимание! Дальнейшие нажатия Tab могут вывести за границы

<ul>
  <li tabindex="1">Один</li>
  <li tabindex="0">Ноль</li>
  <li tabindex="2">Два</li>
  <li tabindex="-1">Минус один</li>
</ul>

<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</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 :)

Обычно `` не поддерживает фокусировку, но здесь есть `tabindex`.

Делегирование с focus/blur

События `focus` и `blur` не всплывают.

Это грустно, поскольку мы не можем использовать делегирование с ними. Например, мы не можем сделать так, чтобы при фокусировке в форме она вся подсвечивалась:

```

<!-- при фокусировке на форме ставим ей класс -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

```

Пример выше не работает, т.к. при фокусировке на любом `<input>` событие `focus` срабатывает только на этом элементе и не всплывает наверх. Так что обработчик `onfocus` на форме никогда не сработает.

Что делать? Печально мы должны присваивать обработчик каждому полю <input>?

Это забавно, но хотя focus/blur не всплывают, они могут быть пойманы на фазе перехвата.

Вот так сработает:

```
<form id="form">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
</form>

<style>
  .focused {
    outline: 1px solid red;
  }
</style>

<script>
  // ставим обработчики на фазе перехвата, последний аргумент true
  form.addEventListener("focus", function() {
    this.classList.add('focused');
  }, true);

  form.addEventListener("blur", function() {
    this.classList.remove('focused');
  }, true);
</script>
```

Ваше имя Ваша фамилия

События focusin/focusout

События focusin/focusout — то же самое, что и focus/blur, только они всплывают.

У них две особенности:

- Не поддерживаются Firefox (хотя поддерживаются даже старейшими IE), см. [https://bugzilla.mozilla.org/show_bug.cgi?id=687787 ↗](https://bugzilla.mozilla.org/show_bug.cgi?id=687787).
- Должны быть назначены не через on-свойство, а при помощи elem.addEventListener.

Из-за отсутствия поддержки Firefox эти события используют редко. Получается, что во всех браузерах можно использовать focus на стадии перехвата, но a focusin/focusout — в IE8-, где стадии перехвата нет.

Подсветка формы в примере ниже работает во всех браузерах.

```

<form name="form">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
</form>
<style>
  .focused {
    outline: 1px solid red;
  }
</style>

<script>
  function onFormFocus() {
    this.className = 'focused';
  }

  function onFormBlur() {
    this.className = '';
  }

  var form = document.forms.form;

  if (form.addEventListener) {
    // focus/blur на стадии перехвата срабатывают во всех браузерах
    // поэтому используем их
    form.addEventListener('focus', onFormFocus, true);
    form.addEventListener('blur', onFormBlur, true);
  } else {
    // ветка для IE8-, где нет стадии перехвата, но есть focusin/focusout
    form.onfocusin = onFormFocus;
    form.onfocusout = onFormBlur;
  }
</script>

```

Итого

События focus/blur происходят при получении и снятии фокуса с элемента.

У них есть особенности:

- Они не всплывают. Но на фазе перехвата их можно перехватить. Это странно, но это так, не спрашивайте почему.
- Везде, кроме Firefox, поддерживаются всплывающие альтернативы focusin/focusout.
- 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.

Текущий элемент: document.activeElement

Кстати, текущий элемент, на котором фокус, доступен как document.activeElement.

Задачи

Улучшенный плейсхолдер

важность: 5

Реализуйте более удобный плейсхолдер-подсказку на JavaScript через атрибут `data-placeholder`.

Правила работы плейсхолдера:

- Элемент изначально содержит плейсхолдер. Специальный класс `placeholder` придает ему синий цвет.
- При фокусировке плейсхолдер показывается уже над полем, становясь «подсказкой».
- При снятии фокуса, подсказка убирается, если поле пустое — плейсхолдер возвращается в него.

Демо:

Красивый placeholder:

[E-mail](#)

В этой задаче плейсхолдер должен работать на одном конкретном `input`. Подумайте, если `input` много, как здесь применить делегирование?

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Мышонок на «клавиатурном» приводе

важность: 4

Кликните по мышонку. Затем нажимайте клавиши со стрелками, и он будет двигаться.

[Демо в новом окне](#) ↗

В этой задаче запрещается ставить обработчики куда-либо, кроме элемента `#mouse`.

Можно изменять атрибуты и классы в HTML.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Горячие клавиши

важность: 5

Создайте `<div>`, который при нажатии `Ctrl+E` превращается в `<textarea>`.

Изменения, внесенные в поле, можно сохранить обратно в `<div>` сочетанием клавиш `Ctrl+S`, при этом `<div>` получит в виде HTML содержимое `<textarea>`.

Если же нажать `Esc`, то `<textarea>` снова превращается в `<div>`, изменения не сохраняются.

[Демо в новом окне ↗](#).

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Редактирование TD по клику

важность: 5

Сделать ячейки таблицы `td` редактируемыми по клику.

- При клике — ячейка `<td>` превращается в редактируемую, можно менять HTML. Размеры ячеек при этом не должны меняться.
- В один момент может редактироваться одна ячейка.
- При редактировании под ячейкой появляются кнопки для приема и отмена редактирования, только клик на них заканчивает редактирование.

Демо:

Кликните на ячейке для начала редактирования. Когда закончите -- нажмите OK или CANCEL.

Bagua Chart: Direction, Element, Color, Meaning

Northwest	North	Northeast
Metal	Water	Earth
Silver	Blue	Yellow
Elders	Change	Direction
West	Center	East
Metal	All	Wood
Gold	Purple	Blue
Youth	Harmony	Future
Southwest	South	Southeast
Earth	Fire	Wood

[Открыть песочницу для задачи.](#)

[К решению](#)

Красивый плейсхолдер для INPUT

важность: 5

Создайте для `<input type="password">` красивый, стилизованный плейсхолдер, например (кликните на тексте):

Добро пожаловать
Скажи пароль, друг
.. и заходи

При клике плейсхолдер просто исчезает и дальше не показывается.

[Открыть песочницу для задачи.](#)

[К решению](#)

Поле, предупреждающее о включенном CapsLock

важность: 3

Создайте поле, которое будет предупреждать пользователя, если включен `CapsLock`. Выключение `CapsLock` уберёт предупреждение.

Такое поле может помочь избежать ошибок при вводе пароля.

Введите текст(например, пароль) с нажатым CapsLock:

[Открыть песочницу для задачи.](#)

[К решению](#)

Изменение: change, input, cut, copy, paste

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

Событие change

Событие [change ↗](#) происходит по окончании изменения значения элемента формы, когда это изменение зафиксировано.

Для текстовых элементов это означает, что событие произойдёт не при каждом вводе, а при потере фокуса.

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` оно срабатывает сразу при выборе значения.

Поздний `onchange` в IE8-

В IE8- `checkbox/radio` при изменении мышью не инициируют событие сразу, а ждут потери фокуса.

Для того, чтобы видеть изменения `checkbox/radio` тут же — в IE8- нужно повесить обработчик на событие `click` (оно произойдет и при изменении значения с клавиатуры) или воспользоваться событием `propertychange`, описанным далее.

Событие input

Событие `input` срабатывает *тут же* при изменении значения текстового элемента и поддерживается всеми браузерами, кроме IE8-.

В IE9 оно поддерживается частично, а именно — *не возникает при удалении символов* (как и `onpropertychange`).

Пример использования (не работает в IE8-):

```
<input type="text"> oninput: <span id="result"></span>
<script>
  var input = document.body.children[0];

  input.oninput = function() {
    document.getElementById('result').innerHTML = input.value;
  };
</script>
```

oninput:

В современных браузерах `oninput` — самое главное событие для работы с элементом формы. Именно его, а не `keydown`/`keypress` следует использовать.

Если бы ещё не проблемы со старыми IE... Впрочем, их можно решить при помощи события `propertychange`.

IE10-, событие `propertychange`

Это событие происходит только в IE10-, при любом изменении свойства. Оно позволяет отлавливать изменение тут же. Оно нестандартное, и его основная область использования — исправление недочётов обработки событий в старых IE.

Если поставить его на `checkbox` в IE8-, то получится «правильное» событие `change`:

```
<input type="checkbox"> Чекбокс с "onchange", работающим везде одинаково
<script>
  var checkbox = document.body.children[0];

  if ("onpropertychange" in checkbox) {
    // старый IE
    checkbox.onpropertychange = function() {
      // проверим имя изменённого свойства
      if (event.propertyName == "checked") {
        alert( checkbox.checked );
      }
    };
  } else {
    // остальные браузеры
    checkbox.onchange = function() {
      alert( checkbox.checked );
    };
  }
</script>
```

Чекбокс с "onchange", работающим везде одинаково

Это событие также срабатывает при изменении значения текстового элемента. Поэтому его можно использовать в старых IE вместо `oninput`.

К сожалению, в IE9 у него недочёт: оно не срабатывает при удалении символов. Поэтому сочетания `onpropertychange + oninput` недостаточно, чтобы поймать любое изменение поля в старых IE.

Далее мы рассмотрим пример, как это можно сделать иначе.

События `cut`, `copy`, `paste`

Эти события используются редко. Они происходят при вырезании/вставке/копировании значения.

К сожалению, кросс-браузерного способа получить данные, которые вставляются/копируются, не существует, поэтому их основное применение — это отмена соответствующей операции.

Например, вот так.

```
<input type="text" id="input"> event: <span id="result"></span>
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    result.innerHTML = event.type + ' ' + input.value;
    return false;
  };
</script>
```

event:

Пример: поле с контролем СМС

Как видим, событий несколько и они взаимно дополняют друг друга.

Посмотрим, как их использовать, на примере.

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

Как такое реализовать?

Событие `input` идеально решит задачу во всех браузерах, кроме IE9-. Собственно, если IE9- нам не нужен, то на этом можно и остановиться.

IE9-

В IE8- событие `input` не поддерживается, но, как мы видели ранее, есть `onpropertychange`, которое может заменить его.

Что же касается IE9 — там поддерживаются и `input` и `onpropertychange`, но они оба не работают при удалении символов. Поэтому мы будем отслеживать удаление при помощи `keyup` на `Delete` и `BackSpace`. А вот удаление командой «вырезать» из меню — сможет отловить лишь `oncut`.

Получается вот такая комбинация:

```
<input type="text" id="sms"> символов: <span id="result"></span>
<script>
  function showCount() {
    result.innerHTML = sms.value.length;
  }

  sms.onkeyup = sms.oninput = showCount;
  sms.onpropertychange = function() {
    if (event.propertyName == "value") showCount();
  }
  sms.oncut = function() {
    setTimeout(showCount, 0); // на момент oncut значение еще старое
  };
</script>
```

символов:

Здесь мы добавили вызов `showCount` на все события, которые могут приводить к изменению значения. Да, иногда изменение будет обрабатываться несколько раз, но зато с гарантией. А лишние вызовы легко убрать, например, при помощи `throttle`-декоратора, описанного в задаче [Тормозилка](#).

Есть и совсем другой простой, но единственный вариант: через `setInterval` регулярно проверять значение и, если оно слишком длинное, обрезать его.

Чтобы сэкономить ресурсы браузера, мы можем начинать отслеживание по `onfocus`, а прекращать — по `onblur`, вот так:

```
<input type="text" id="sms"> символов: <span id="result"></span>

<script>
  var timerId;

  sms.onfocus = function() {

    var lastValue = sms.value;
    timerId = setInterval(function() {
      if (sms.value != lastValue) {
        showCount();
        lastValue = sms.value;
      }
    }, 20);
  };

  sms.onblur = function() {
    clearInterval(timerId);
  };

  function showCount() {
    result.innerHTML = sms.value.length;
  }
</script>
```

символов:

Обратим внимание — весь этот «танец с бубном» нужен только для поддержки IE8-, в которых не поддерживается `oninput` и IE9, где `oninput` не работает при удалении.

Итого

События изменения данных:

Событие	Описание	Особенности
change	Изменение значения любого элемента формы. Для текстовых элементов срабатывает при потере фокуса.	В IE8- на чекбоксах ждет потери фокуса, поэтому для мгновенной реакции ставят также <code>onclick</code> -обработчик

<code>input</code>	Событие срабатывает только на текстовых элементах. Оно не ждет потери фокуса, в отличие от <code>change</code> .	В IE8- не поддерживается, в IE9 не работает при удалении символов.
<code>propertychange</code>	Только для IE10-. Универсальное событие для отслеживания изменения свойств элементов. Имя изменённого свойства содержится в <code>event.propertyName</code> . Используют для мгновенной реакции на изменение значения в старых IE.	В IE9 не срабатывает при удалении символов.
<code>cut/copy/paste</code>	Срабатывают при вставке/копировании/удалении текста. Если в их обработчиках отменить действие браузера, то вставки/копирования/удаления не произойдёт.	Вставляемое значение получить нельзя: на момент срабатывания события в элементе всё ещё <i>старое</i> значение, а новое недоступно.

Ещё особенность: в IE8- события `change`, `propertychange`, `cut` и аналогичные не всплывают. То есть, обработчики нужно назначать на сам элемент, без делегирования.

✓ Задачи

Автовычисление процентов по вкладу

важность: 5

Создайте интерфейс для автоматического вычисления процентов по вкладу.

Ставка фиксирована: 12% годовых. При включённом поле «капитализация» — проценты приплюсовываются к сумме вклада каждый месяц ([сложный процент ↗](#)).

Пример:

Калькулятор процентов, из расчёта 12% годовых.

Сумма	10000
Срок в месяцах	12 (год)
С капитализацией	<input type="checkbox"/>

Было: Станет:
10000 **11200**

Технические требования:

- В поле с суммой должно быть нельзя ввести не-цифру. При этом пусть в нём работают специальные клавиши и сочетания Ctrl-X/Ctrl-V.
- Изменения в форме отражаются в результатах сразу.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Формы: отправка, событие и метод submit

Событие `submit` возникает при отправке формы. Наиболее частое его применение — это *валидация* (проверка) формы перед отправкой.

Метод `submit` позволяет инициировать отправку формы из JavaScript, без участия пользователя. Далее мы рассмотрим детали их использования.

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, находясь на каком-нибудь поле.**

Какой бы способ ни выбрал посетитель — будет сгенерировано событие `submit`. Обработчик в нём может проверить данные и, если они неверны, то вывести ошибку и сделать `event.preventDefault()` — тогда форма не отправится на сервер.

Например, в таком случае она просто выведет alert, форма не будет отправлена.

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

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 .

i Взаимосвязь событий submit и click

При отправке формы путём нажатия Enter на текстовом поле, на элементе везде, кроме IE8-, генерируется событие click.

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

```
<form onsubmit="alert('submit');return false">
    <input type="text" size="30" value="При нажатии Enter будет click">
    <input type="submit" value="Submit" onclick="alert('click')">
</form>
```

! В IE8- событие submit не всплывает

В IE8- событие submit не всплывает. Нужно вешать обработчик submit на сам элемент формы, без использования делегирования.

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.

✓ Задачи

Модальное диалоговое окно

важность: 5

Создайте функцию `showPrompt(text, callback)`, которая выводит форму для ввода с сообщением `text` и кнопками ОК/Отмена.

- При отправке формы (OK/ввод в текстовом поле) — должна вызываться функция `callback` со значением поля.
- При нажатии на Отмена или на клавишу `Esc` — должна вызываться функция `callback(null)`. Клавиша `Esc` должна закрывать форму всегда, даже если поле для ввода сообщения не в фокусе.

Особенности реализации:

- Форма должна показываться в центре окна (и оставаться в центре при изменении его размеров, а также при прокрутке окна!).
- Текст может состоять из нескольких строк, возможен любой HTML
- При показе формы остальные элементы страницы использовать нельзя, не работают другие кнопки и т.п., это окно — *модальное*.
- При показе формы — сразу фокус на INPUT для ввода.
- Нажатия `Tab` / `Shift+Tab` переключают в цикле только по полям формы, они не позволяют переключиться на другие элементы страницы.

Пример использования:

```
showPrompt("Введите что-нибудь<br>... умное :)", function(value) {
  alert( value );
});
```

Демо в iframe:

Нажмите на кнопку ниже

Нажмите для показа формы ввода

Исходный HTML/CSS для формы с готовым fixed-позиционированием – в песочнице.

[Открыть песочницу для задачи.](#)

[К решению](#)

Валидация формы

важность: 3

Напишите функцию `validate(form)`, которая проверяет содержимое формы по клику на кнопку «Проверить».

Ошибки:

1. Одно из полей не заполнено.
2. Пароли не совпадают.

Ошибка должна сопровождаться сообщением у поля. Например:

The form consists of several input fields and a dropdown menu. The fields are labeled: 'От кого' (Recipient), 'Ваш пароль' (Your password), 'Повторите пароль' (Repeat password), and 'Куда' (Where). Below these is a large text area labeled 'Сообщение:' (Message). A dropdown arrow is positioned next to the 'Куда' field. At the bottom left is a button labeled 'Проверить' (Check).

[Открыть песочницу для задачи.](#)

[К решению](#)

Создание графических компонентов

В этом разделе мы обсуждаем, как вместо «простыни кода» писать объектно-ориентированные компоненты на JavaScript.

Введение

Для надёжности необходима простота.

“ Эдсгер Вибе
Дейкстра

В современной JavaScript-разработке используются фреймворки, которые дают готовые библиотеки и правила для написания кода. Эта глава не ставит своей целью их все заменить, а равно как и научить какому-нибудь фреймворку.

Вместо этого мы разберём основные средства для построения архитектуры, и при помощи чистого JavaScript построим компоненты интерфейса с их использованием.

Это во-первых покажет, что и без фреймворков есть жизнь, а во-вторых даст фундамент для освоения любого фреймворка, какой бы вы ни выбрали.

Графические компоненты

Первый и главный шаг в наведении порядка — это оформить код в объекты, каждый из которых будет решать свою задачу.

Здесь мы сосредоточимся на графических компонентах, которые также называют «виджетами».

В браузерах есть встроенные виджеты, например `<select>`, `<input>` и другие элементы, о которых мы даже и не думаем, «как они работают». Они «просто работают»: показывают значение, вызывают события...

Наша задача — сделать то же самое на уровне выше. Мы будем создавать объекты, которые генерируют меню, диалог или другие компоненты интерфейса, и дают возможность удобно работать с ними.

Виджет Menu

Мы начнём работу с виджета, который предусматривает уже готовую разметку.

То есть, в нужном месте HTML находится DOM-структура для меню — заголовок и список опций:

```
<div class="menu" id="sweets-menu">
  <span class="title">Сладости</span>
  <ul>
    <li>Торт</li>
    <li>Пончик</li>
    <li>...</li>
  </ul>
</div>
```

Далее она может дополняться, изменяться, но в начале — она такая.

Обратим внимание на важные соглашения виджета:

Вся разметка заключена в корневой элемент `<div class="menu" id="sweeties-menu">`.

Это очень удобно. Выпусти этот элемент из меню — нет меню, вытащи в другое место — переместил меню. Кроме того, можно удобно искать подэлементы.

Внутри корневого элемента — только классы, не id.

Документ вполне может содержать много различных меню. Они не должны конфликтовать между собой, поэтому для разметки везде используются классы.

Иключение — корневой элемент. В данном случае мы предполагаем, что данное конкретное «меню сладостей» в документе только одно, поэтому даём ему id.

Класс виджета

Для работы с разметкой будем создавать объект new Menu и передавать ему корневой элемент. В конструкторе он поставит необходимые обработчики:

```
function Menu(options) {
  var elem = options.elem;

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

  elem.onclick = function(event) {
    if (event.target.closest('.title')) {
      elem.classList.toggle('open');
    }
  };
}

// использование
var menu = new Menu({
  elem: document.getElementById('sweets-menu')
});
```

Меню:

[Смотреть пример онлайн ↗](#)

Это, конечно, только первый шаг, но уже здесь видны некоторые важные соглашения в коде.

У конструктора только один аргумент — объект options.

Это удобно, так как у графических компонентов обычно много настроек, большинство из которых имеют разумные значения «по умолчанию». Если передавать аргументы через запятую — их будет слишком много.

Обработчики назначаются через делегирование.

Вместо того, чтобы найти элемент и поставить обработчик на него:

```
var titleElem = elem.querySelector('.title');

titleElem.onclick = function() {
  elem.classList.toggle('open');
}
```

...Мы ставим обработчик на корневой elem и используем делегирование:

```
elem.onclick = function(event) {
  if (event.target.closest('.title')) {
    elem.classList.toggle('open');
  }
};
```

Это ускоряет инициализацию, так как не надо искать элементы, и даёт возможность в любой момент менять DOM внутри, в том числе через innerHTML, без необходимости переставлять обработчик.

В этот код лучше добавить дополнительную проверку на то, что найденный .title находится внутри elem:

```
elem.onclick = function(event) {
  var closestTitle = event.target.closest('.title');
  if (closestTitle && elem.contains(closestTitle)) {
    elem.classList.toggle('open');
  }
};
```

Публичные методы

Уважающий себя компонент обычно имеет публичные методы, которые позволяют управлять им снаружи.

Рассмотрим повнимательнее этот фрагмент:

```
if (event.target.closest('.title')) {
  elem.classList.toggle('open');
}
```

Здесь в обработчике события сразу код работы с элементом. Пока одна строка — всё понятно, но если их будет много, то при чтении понадобится долго и упорно вникать: «А что же, всё-таки, такое делается при клике?»

Для улучшения читаемости выделим обработчик в отдельную функцию toggle, которая к тому же станет полезным публичным методом:

```
function Menu(options) {
  var elem = options.elem;

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

  elem.onclick = function(event) {
    if (event.target.closest('.title')) {
      toggle();
    }
  };

  function toggle() {
    elem.classList.toggle('open');
  }

  this.toggle = toggle;
}
```

Теперь метод `toggle` можно использовать и снаружи:

```
var menu = new Menu(...);
menu.toggle();
```

Генерация DOM-элемента

До этого момента меню «оживляло» уже существующий HTML.

Но далеко не всегда в HTML уже есть готовая разметка. В сложных интерфейсах намного чаще её нет, а есть данные, на основе которых компонент генерирует разметку.

В случае меню, данные — это набор пунктов меню, которые передаются конструктору.

Для генерации DOM добавим меню три метода:

- `render()` — генерирует корневой DOM-элемент и заголовок меню.
- `renderItems()` — генерирует DOM для списка опций `ul/li`.
- `getElem()` — возвращает DOM-элемент меню, при необходимости запуская генерацию, публичный метод.

Функция генерации корневого элемента с заголовком `render` отделена от генерации списка `renderItems`. Почему — будет видно чуть далее.

Новый способ использования меню:

```
// создать объект меню с данным заголовком и опциями
var menu = new Menu({
  title: "Сладости",
  items: [
    "Торт",
    "Пончик",
    "Пирожное",
    "Шоколадка",
    "Мороженое"
  ]
});

// получить сгенерированный DOM-элемент меню
var elem = menu.getElem();

// вставить меню в нужное место страницы
document.body.appendChild(elem);
```

Код Menu с новыми методами:

```

function Menu(options) {
  var elem;

  function getElem() {
    if (!elem) render();
    return elem;
  }

  function render() {
    elem = document.createElement('div');
    elem.className = "menu";

    var titleElem = document.createElement('span');
    elem.appendChild(titleElem);
    titleElem.className = "title";
    titleElem.textContent = options.title;

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

    elem.onclick = function(event) {
      if (event.target.closest('.title')) {
        toggle();
      }
    }
  }

  function renderItems() {
    var items = options.items || [];
    var list = document.createElement('ul');
    items.forEach(function(item) {
      var li = document.createElement('li');
      li.textContent = item;
      list.appendChild(li);
    });
    elem.appendChild(list);
  }

  function open() {
    if (!elem.querySelector('ul')) {
      renderItems();
    }
    elem.classList.add('open');
  };

  function close() {
    elem.classList.remove('open');
  };

  function toggle() {
    if (elem.classList.contains('open')) close();
    else open();
  };

  this.getElem = getElem;
  this.toggle = toggle;
  this.close = close;
  this.open = open;
}

```

Отметим некоторые особенности этого кода.

Обработчики отделяются от реальных действий.

В обработчике onclick мы «ловим» событие и выясняем, что именно произошло. Возможно, нужно проверить event.target, координаты, клавиши-модификаторы, и т.п. Это всё можно делать здесь

же.

Выяснив, что нужно сделать, обработчик `onclick` не делает это сам, а вызывает для этого соответствующий метод. Этот метод уже не знает ничего о событии, он просто делает что-то с виджетом. Его можно вызвать и отдельно, не из обработчика.

Здесь есть ряд важных плюсов:

- Обработчик `onclick` не «распухает» чрезмерно.
- Код гораздо лучше читается.
- Метод можно повторно использовать, в том числе и сделать публичным, как в коде выше.

Генерация DOM, по возможности, должна быть «ленивой».

Мы стараемся откладывать работу до момента, когда она реально нужна. Например, когда `new Menu` создается, то переменная `elem` лишь объявляется. DOM-дерево будет сгенерировано только при вызове `getElem()` функцией `render()`.

Более того! Пока меню закрыто — достаточно заголовка. Кроме того, возможно, посетитель вообще никогда не раскроет это меню, так зачем генерировать список раньше времени? А при первом открытии `open()` вызовет функцию `renderItems()`, которая специально для этого выделена отдельно от `render()`.

Фаза инициализации очень чувствительна к производительности, так как обычно в сложном интерфейсе создаётся многое всего.

Если изначально подходить к оптимизации на этой фазе «спустя рукава», то потом поправить долгий старт может быть сложно. Тем более, что инициализация — это фундамент, начало работы виджета, её оптимизация в будущем может потребовать сильных изменений кода.

Конечно, здесь, как и везде в оптимизации — без фанатизма. Бывают ситуации, когда гораздо удобнее что-то сделать сразу. Если это один элемент, то оптимизация здесь ни к чему. А если большой фрагмент DOM, который, как в случае с меню, прямо сейчас не нужен — то лучше отложить.

В действии:

[Смотреть пример онлайн ↗](#)

Итого

Мы начали создавать компонент «с чистого листа», пока без дополнительных библиотек.

Основные принципы:

- Виджет — это объект, который либо контролирует готовое дерево DOM, либо создаёт своё.
- В конструктор виджета передаётся объект аргументов `options`.
- Виджет при необходимости создаёт элемент или «оживляет» готовый. Внутре в разметке не используются `id`.
- Обработчики назначаются через делегирование — для производительности и упрощения

Виджеты.

- Обработчики событий вызывают соответствующий метод, не пытаются делать всё сами.
- При инициализации, если существенный участок работы можно отложить до реального задействования виджета — откладываем его.

✓ Задачи

Часики

важность: 5

Создайте компонент «Часы» (Clock).

Интерфейс:

```
var clock = new Clock({  
    elem: элемент  
});  
  
clock.start(); // старт  
clock.stop(); // остановка
```

Остальные методы, если нужны, должны быть приватными.

При нажатии на alert часы должны приостанавливаться, а затем продолжать идти с правильным временем.

Пример результата:

00:00:00
Старт Стоп alert для проверки корректного возобновления

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Слайдер-компонент

важность: 5

Перепишите слайдер в виде компонента:

Исходный документ возьмите из решения задачи [Слайдер](#).

[К решению](#)

Компонент: список с выделением

важность: 5

Перепишите решение задачи [Список с выделением](#) в виде компонента.

У компонента должен быть единственный публичный метод `getSelected()`, который возвращает выбранные значения в виде массива.

Использование:

```
var listSelect = new ListSelect({  
  elem: document.querySelector('ul')  
});  
// listSelect.getSelected()
```

Демо:

Клик на элементе выделяет только его.

Ctrl(Cmd)+Клик добавляет/убирает элемент из выделенных.

Shift+Клик добавляет промежуток от последнего кликнутого к выделению.

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

`listSelect.getSelected()`

[К решению](#)

Голосовалка

важность: 5

Напишите функцию-конструктор `new Voter(options)` для голосовалки. Она должна получать элемент в `options.elem`, в следующей разметке:

```
<div id="voter" class="voter">
  <span class="down">-</span>
  <span class="vote">0</span>
  <span class="up">+</span>
</div>
```

По клику на + и – число должно увеличиваться или уменьшаться.

Публичный метод `voter.setVote(vote)` должен устанавливать текущее число — значение голоса.

Все остальные методы и свойства пусть будут приватными.

Результат:

– 1 +

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Голосовалка в прототипном стиле ООП

важность: 5

Поменяйте стиль ООП в голосовалке, созданной в задаче [Голосовалка](#) на прототипный.

Внешний код, использующий класс `Voter`, не должен измениться.

В качестве исходного кода возьмите решение задачи [Голосовалка](#).

[К решению](#)

Добавить двойной голос в голосовалку

важность: 5

Создайте функцию-конструктор `StepVoter`, которая наследует от голосовалки, созданной в задаче [Голосовалка в прототипном стиле ООП](#) и добавляет голосовалке опцию `options.step`, которая задаёт «шаг» голоса.

пример.

```
var voter = new StepVoter({
  elem: document.getElementById('voter'),
  step: 2 // увеличивать/уменьшать сразу на 2 пункта
});
```

Результат:

- 0 +

В реальном проекте влияние клика на голосовалку может зависеть от полномочий или репутации посетителя.

В качестве исходного кода используйте решение задачи [Голосовалка в прототипном стиле ООП](#).

P.S. Код voter.js изменять нельзя, нужно не переписать Voter, а отнаследовать от него.

[К решению](#)

Вёрстка графических компонентов

При создании графических компонент («виджетов») в первую очередь придумывается их HTML/CSS-структура.

Как будет выглядеть виджет в обычном состоянии? Как будет меняться в процессе взаимодействия с посетителем?

Чтобы разработка виджета была удобной, при вёрстке полезно соблюдать несколько простых, но очень важных соглашений.

Семантическая вёрстка

HTML-разметка и названия CSS-классов должны отражать не оформление, а смысл.

Например, сообщение об ошибке можно сверстать так:

```
<div style="color:red; border: 1px solid red">
  Плохая вёрстка сообщения об ошибке: атрибут style!
</div>
```

...Или так:

```
<div class="red red-border">  
  Плохая вёрстка сообщения об ошибке: несемантический class!  
</div>
```

В обоих случаях вёрстка не является семантической. В первом случае — стиль, а во втором — класс содержит информацию об оформлении.

При семантической вёрстке классы описывают смысл («что это?» — меню, кнопка...) и состояние (открыто, закрыто, отключено...) компонента.

Например:

```
<div class="error">  
  Сообщение об ошибке (error), правильная вёрстка!  
</div>
```

У предупреждения будет класс `message` и так далее, по смыслу.

```
<div class="warning">  
  Предупреждение (warning), правильная вёрстка!  
</div>
```

Семантическая верстка упрощает поддержку и развитие CSS, упрощает взаимодействие между членами команды.

Такая верстка удобна для организации JS-кода. В коде мы просто ставим нужный класс, остальное делает CSS.

Состояние виджета — класс на элементе

Зачастую компонент может иметь несколько состояний. Например, меню может быть открыто или закрыто.

Состояние должно добавляться CSS-классом не на тот элемент, который нужно скрыть/показать/..., а на тот, к которому оно «по смыслу» относится, обычно — на корневой элемент.

Например, меню в закрытом состоянии скрывает свой список элементов. Класс `open` нужно добавлять не к списку опций ``, который скрывается-показывается, а к корневому элементу виджета, поскольку это состояние касается всего меню:

```
<div class="menu open">  
  <span class="title">Заголовок меню</span>  
  <ul>  
    <li>Список элементов</li>  
  </ul>  
</div>
```

Или, к примеру, разметка для индикатора загрузки может выглядеть так:

```
<div class="indicator loading">
  <span class="progress">Тут показывается прогресс</span>
</div>
```

Состояние индикатора может быть «в процессе» (loading) или «загрузка завершена» (complete). С точки зрения оформления оно может влиять только на показ внутреннего span, но ставить его нужно всё равно на внешний элемент, ведь это — состояние всего компонента.

Из примеров выше можно подумать, что классы, описывающие состояние, всегда ставятся на корневой элемент. Но это не так.

Возможно и такое, что состояние относится к внутреннему элементу. Например, для дерева состояние открыт/закрыт относится к узлу, соответственно, класс должен быть на узле.

Например:

```
<ul class="tree">
  <li class="closed">
    Закрытый узел дерева
  </li>
  <li class="open">
    Открытый узел дерева
  </li>
  ...
</ul>
```

Префиксы компонента у классов

Рассмотрим пример вёрстки «диалогового окна»:

```

<div class="dialog">
  <h2 class="title">Заголовок</h2>
  <div class="content">
    HTML-содержимое.
  </div>
  <div class="close">Закрыть</div>
</div>

<style>
.dialog {
  background: lightgreen;
  border: lime 2px solid;
  border-radius: 10px;
  padding: 4px;
  position: relative;
}

.dialog .title {
  margin: 0;
  font-size: 24px;
  color: darkgreen;
}

.dialog .content {
  padding: 10px 0 0 0;
}

.dialog .close {
  position: absolute;
  right: 4px;
  top: 4px;
  font-size: 10px;
}
</style>

```

Заголовок

Закрыть

HTML-содержимое.

Диалоговое окно может иметь любое HTML-содержимое.

А что будет, если в этом содержимом окажется меню — да-да, то самое, которое рассмотрели выше, со `` ?

Правило `.dialog .title` применяется ко всем `.title` внутри `.dialog`, а значит — и к нашему меню тоже. Будет конфликт стилей с непредсказуемыми последствиями.

Конечно, можно попытаться бороться с этим. Например, жёстко задать вложенность — использовать класс `.dialog > .title`. Это сработает в данном конкретном примере, но как быть в тех местах, где между `.dialog` и `.title` есть другие элементы? Длинные цепочки вида `.dialog > ... > .title` страшновато выглядят и делают вёрстку ужасно негибкой. К счастью, есть альтернативный путь.

Чтобы избежать возможных проблем, все классы внутри виджета начинают с его имени.

Здесь имя `dialog`, так что все, относящиеся к диалогу, будем начинать с `dialog_`

Получится так:

```
<div class="dialog">
  <h2 class="dialog_title">Заголовок</h2>
  <div class="dialog_content">
    HTML-содержимое.
  </div>
  <div class="dialog_close">Закрыть</div>
</div>

<style>
  .dialog { ... }
  .dialog_title { стиль заголовка }
  .dialog_content { стиль содержимого }
  ...
</style>
```

Здесь двойное подчёркивание __ служит «стандартным» разделителем. Можно выбрать и другой разделитель, но при этом стоит иметь в виду, что иногда имя класса может состоять из нескольких слов. Например title-picture. С двойным подчёркиванием: dialog_title-picture, очень наглядно видно где что.

Есть ещё одно полезное правило, которое заключается в том, что стили должны вешаться на класс, а не на тег. То есть, не h2 { ... }, а .dialog_title { ... }, где .dialog_title — класс на соответствующем заголовке.

Это позволяет и избежать конфликтов на вложенных h2, и использовать всегда те теги, которые имеют правильный смысл, не оглядываясь на встроенные стили (которые можно обнулить своими).

❶ Без фанатизма

На практике из этих правил зачастую делают исключения. Можно «вешать» стили на теги и использовать CSS-каскады без префиксов, если мы при этом твёрдо понимаем, что конфликты заведомо исключены.

Например, когда мы точно знаем, что никакого произвольного HTML внутри элемента (или внутри данного поддерева DOM) не будет.

БЭМ

Описанное выше правило именования элементов является частью более общей концепции «БЭМ», которая разработана в Яндексе.

БЭМ предлагает способ организации HTML/CSS/JS в виде независимых «блоков» — компонент, которые можно легко перемещать по файловой системе и между проектами.

Можно как взять часть идеологии, например систему именования классов, так и полностью перейти на инструментарий БЭМ, который даёт инструменты сборки для HTML/JS/CSS, описанных по БЭМ-методу.

Более подробное описание основ БЭМ можно почитать в статье <https://ru.bem.info/articles/bem-for-small-projects/>, а о системе вообще — на сайте <http://ru.bem.info>.

- Вёрстка должна быть семантической, использовать соответствующие смыслу информации теги и классы.
- Класс, описывающий состояние всего компонента, нужно ставить на его корневом элементе, а не на том, который нужно «украсить» в этом состоянии. Если состояние относится не ко всему компоненту, а к его части — то на соответствующем «по смыслу» DOM-узле.
- Классы внутри компонента должны начинаться с префикса — имени компонента.

Это не всегда строго необходимо, но позволяет избежать проблем в случаях, когда компонент может содержать произвольный DOM, как например диалоговое окно с произвольным HTML-текстом.

Использование `.dialog__title` вместо `.dialog .title` гарантирует, что CSS не применится по ошибке к какому-нибудь другому `.title` внутри диалога.

Задачи

Семантическое меню

важность: 5

Посмотрите на вёрстку горизонтального меню.

```
<div class="rounded-horizontal-blocks">
  <div class="item">Главная</div>
  <div class="vertical-splitter">|</div>
  <div class="item">Товары</div>
  <div class="item">Фотографии</div>
  <div class="item">Контакты</div>
</div>
```

```
.rounded-horizontal-blocks .item {
  float: left;
  padding: 6px;
  margin: 0 2px;
  border: 1px solid gray;
  border-radius: 10px;
  cursor: pointer;
  font-size: 90%;
  background: #FFF5EE;
}

.vertical-splitter {
  float: left;
  padding: 6px;
  margin: 0 2px;
}

.item:hover {
  text-decoration: underline;
}
```

[Главная](#)[Товары](#)[Фотографии](#)[Контакты](#)

Что делает эту вёрстку несемантичной? Найдите 3 ошибки (или больше).

Как бы вы сверстали меню правильно?

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Шаблонизатор LoDash

В этой главе мы рассмотрим *шаблонизацию* — удобный способ генерации HTML по «шаблону» и *данным*.

Большинство виджетов, которые мы видели ранее, получают готовый HTML/DOM и «оживляют» его. Это типичный случай в сайтах, где JavaScript — на ролях «второго помощника». Разметка, CSS уже есть, от JavaScript, условно говоря, требуются лишь обработчики, чтобы менюшки заработали.

Но в сложных интерфейсах разметка изначально отсутствует на странице. Компоненты генерируют свой DOM сами, динамически, на основе данных, полученных с сервера или из других источников.

Зачем нужны шаблоны?

Ранее мы уже видели код `Menu`, который сам создаёт свой элемент:

```
function Menu(options) {
  // ... приведены только методы для генерации DOM ...

  function render() {
    elem = document.createElement('div');
    elem.className = "menu";

    var titleElem = document.createElement('span');
    elem.appendChild(titleElem);
    titleElem.className = "title";
    titleElem.textContent = options.title;

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

    elem.onclick = function(event) {
      if (event.target.closest('.title')) {
        toggle();
      }
    }
  }

  function renderItems() {
    var items = options.items || [];
    var list = document.createElement('ul');
    items.forEach(function(item) {
      var li = document.createElement('li');
      li.textContent = item;
      list.appendChild(li);
    });
    elem.appendChild(list);
  }
  // ...
}
```

Понятен ли этот код? Очевидно ли, какой HTML он генерируют методы `render`, `renderItems`?

С первого взгляда — вряд ли. Нужно как минимум внимательно посмотреть и продумать код, чтобы разобраться, какая именно DOM-структура создаётся.

...А что, если нужно изменить создаваемый HTML? ...А что, если эта задача досталась не программисту, который написал этот код, а верстальщику, который с HTML/CSS проекта знаком отлично, но этот JS-код видит впервые? Вероятность ошибок при этом зашкаливает за все разумные пределы.

К счастью, генерацию HTML можно упростить. Для этого воспользуемся библиотекой шаблонизации.

Пример шаблона

Шаблон — это строка в специальном формате, которая путём подстановки значений (текст сообщения, цена и т.п.) и выполнения встроенных фрагментов кода превращается в DOM/HTML.

Пример шаблона для меню:

```
<div class="menu">
  <span class="title"><%-title%></span>
  <ul>
    <% items.forEach(function(item) { %>
      <li><%-item%></li>
    <% } ); %>
  </ul>
</div>
```

Как видно, это обычный HTML, с вставками вида `<% ... %>`.

Для работы с таким шаблоном используется специальная функция `_.template`, которая предоставляется фреймворком [LoDash ↗](#), её синтаксис мы подробно посмотрим далее.

Пример использования `_.template` для генерации HTML с шаблоном выше:

```
// сгенерировать HTML, используя шаблон tmpl (см. выше)
// с данными title и items
var html = _.template(tmpl)({
  title: "Сладости",
  items: [
    "Торт",
    "Печенье",
    "Пирожное"
  ]
});
```

Значение `html` в результате:

```
<div class="menu">
  <span class="title">Сладости</span>
  <ul>
    <li>Торт</li>
    <li>Печенье</li>
    <li>Сладости</li>
  </ul>
</div>
```

Этот гораздо проще, чем JS-код, не правда ли? Шаблон очень наглядно показывает, что в итоге должно получиться. В отличие от кода, в шаблоне первичен текст, а вставок кода обычно мало.

Давайте подробнее познакомимся с `_.template` и синтаксисом шаблонов.

Holy war detected!

Способов шаблонизации и, в особенности, синтаксисов шаблонов, примерно столько же, сколько способов [поймать льва в пустыне](#). Иначе говоря... много.

Эта глава — совершенно не место для священных войн на эту тему.

Далее будет более полный обзор типов шаблонных систем, применяемых в JavaScript, но начнём мы с `_.template`, поскольку эта функция проста, быстра и демонстрирует приёмы, используемые в целом классе шаблонных систем, активно используемых в самых разных JS-проектах.

Синтаксис шаблона

Шаблон представляет собой строку со специальными разделителями, которых всего три:

<% code %> — код

Код между разделителями `<% ... %>` будет выполнен «как есть»

<%= expr %> — для вставки expr как HTML

Переменная или выражение внутри `<%= ... %>` будет вставлено «как есть». Например: `<%=title %>` вставит значение переменной `title`, а `<%=2+2%>` вставит 4.

<%- expr %> — для вставки expr как текста

Переменная или выражение внутри `<%- ... %>` будет вставлено «как текст», то есть с заменой символов `<` `>` `&` `"` `'` на соответствующие HTML-entities.

Например, если `expr` содержит текст `
`, то при `<%-expr%>` в результат попадёт, в отличие от `<%=expr%>`, не HTML-тег `
`, а текст `
`.

Функция `_.template`

Для работы с шаблоном в библиотеке [LoDash](#) есть функция `_.template tmpl, data, options`.

Её аргументы:

tmpl

Шаблон.

options

Необязательные настройки, например можно поменять разделители.

Эта функция запускает «компиляцию» шаблона `tmpl` и возвращает результат в виде функции, которую далее можно запустить с данными и получить строку-результат.

Вот так:

```
// Шаблон
var tmpl = _.template('<span class="title"><%=title%></span>');

// Данные
var data = {
  title: "Заголовок"
};

// Результат подстановки
alert( tmpl(data) ); // <span class="title">Заголовок</span>
```

Пример выше похож на операцию «поиск-и-замена»: шаблон просто заменил `<%=title%>` на значение свойства `data.title`.

Но возможность вставки JS-кода делает шаблоны сильно мощнее.

Например, вот шаблон для генерации списка от 1 до `count`:

```
// используется \, чтобы объявить многострочную переменную-текст шаблона
var tmpl = '<ul>
<% for (var i=1; i<=count; i++) { %> \
<li><%=i%></li> \
<% } %> \
</ul>';
alert( _.template(tmpl)({count: 5}) ); // <ul><li>1</li><li>2</li>...</ul>
```

Здесь в результат попал сначала текст ``, потом выполнился код `for`, который последовательно сгенерировал элементы списка, и затем список был закрыт ``.

Хранение шаблона в документе

Шаблон — это многострочный HTML-текст. Записывать его прямо в скрипте — неудобно.

Один из альтернативных способов объявления шаблона — записать его в HTML, в тег `<script>` с нестандартным `type`, например "text/template":

```
<script type="text/template" id="menu-template">
<div class="menu">
  <span class="title"><%-title%></span>
</div>
</script>
```

Если `type` не знаком браузеру, то содержимое такого скрипта игнорируется, однако оно доступно при помощи `innerHTML`:

```
var template = document.getElementById('menu-template').innerHTML;
```

В данном случае выбран `type="text/template"`, однако подошёл бы и любой другой нестандартный, например `text/html`. Главное, что браузер такой скрипт никак не обработает. То есть, это всего лишь способ передать строку шаблона в HTML.

```
<!-- библиотека LoDash -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>

<!-- шаблон для списка от 1 до count -->
<script type="text/template" id="list-template">
<ul>
  <% for (var i=1; i<=count; i++) { %>
    <li><%=i%></li>
  <% } %>
</ul>
</script>

<script>
  var tmpl = _.template(document.getElementById('list-template').innerHTML);

  // ..а вот и результат
  var result = tmpl({count: 5});
  document.write( result );
</script>
```

Как работает функция `_.template`?

Понимание того, как работает `_.template`, очень важно для отладки ошибок в шаблонах.

Как обработка шаблонов устроена внутри? За счёт чего организована возможность перемежать с текстом произвольный JS-код?

Оказывается, очень просто.

Вызов `_.template(str)` разбивает строку `str` по разделителям и, при помощи `new Function` создаёт на её основе JavaScript-функцию. Тело этой функции создаётся таким образом, что код, который в шаблоне оформлен как `<% ... %>` — попадает в неё «как есть», а переменные и текст прибавляются к специальному временному «буферу», который в итоге возвращается.

Взглянем на пример:

```
var compiled = _.template("<h1><%=title%></h1>");

alert( compiled );
```

Функция `compiled`, которую вернул вызов `_template` из этого примера, выглядит примерно так:

```
function(obj) {
  obj || (obj = {});
  var __t, __p = '', __e = _.escape;
  with(obj) {
    __p += '<h1>' +
    ((__t = (title)) == null ? '' : __t) +
    '</h1>';
  }
  return __p
}
```

Сталился результатом вызова new Function('obj', 'код'), где код написан таким образом генерируется на основе шаблона:

1. Вначале в коде идёт «шапка» — стандартное начало функции, в котором объявляется переменная __p. В неё будет записываться результат.
2. Затем добавляется блок `with(obj) { ... }`, внутри которого в __p добавляются фрагменты HTML из шаблона, а также переменные из выражений `<%=...%>`. Код из `<%=...%>` копируется в функцию «как есть».
3. Затем функция завершается, и `return __p` возвращает результат.

При вызове этой функции, например `compiled({title: "Заголовок"})`, она получает объект данных как `obj`, здесь это `{title: "Заголовок"}`, и если внутри `with(obj) { .. }` обратиться к `title`, то по правилам [конструкции with](#) это свойство будет получено из объекта.

Можно и без with

Конструкция `with` является устаревшей, но в данном случае она полезна.

Так как функция создаётся через `new Function("obj", "код")` то:

- Она работает в глобальной области видимости, не имеет доступа к внешним локальным переменным.
- Внешний `use strict` на такую функцию не влияет, то есть даже в строгом режиме шаблон продолжит работать.

Если мы всё же не хотим использовать `with` — нужно поставить второй параметр — `options`, указав параметр `variable` (название переменной с данными).

Например:

```
alert(_.template("<h1><%=menu.title%></h1>", {variable: "menu"}));
```

Результат:

```
function(menu) {
  var __t, __p = '';
  __p += '<h1>' +
    ((__t = (menu.title)) == null ? '' : __t) +
    '</h1>';
  return __p
}
```

При таком подходе переменная `title` уже не будет искаться в объекте данных автоматически, поэтому нужно будет обращаться к ней как `<%=menu.title%>`.

Кеширование скомпилированных шаблонов

Чтобы не компилировать один и тот же шаблон много раз, результаты обычно кешируют.

Например, глобальная функция `getTemplate("menu-template")` может доставать шаблон из HTML, компилировать, результат запоминать и сразу отдавать при последующих обращениях к тому же шаблону.

Меню на шаблонах

Рассмотрим для наглядности полный пример меню на шаблонах.

HTML (шаблоны):

```
<script type="text/template" id="menu-template">
<div class="menu">
  <span class="title"><%-title%></span>
</div>
</script>

<script type="text/template" id="menu-list-template">
<ul>
  <% items.forEach(function(item) { %>
    <li><%-item%></li>
  <%}); %>
</ul>
</script>
```

JS для создания меню:

```
var menu = new Menu({
  title: "Сладости",
  // передаём также шаблоны
  template: _.template(document.getElementById('menu-template').innerHTML),
  listTemplate: _.template(document.getElementById('menu-list-template').innerHTML),
  items: [
    "Торт",
    "Пончик",
    "Пирожное",
    "Шоколадка",
    "Мороженое"
  ],
  document.body.appendChild(menu.getElement());
```

JS код Menu:

```
function Menu(options) {
  var elem;

  function getElem() {
    if (!elem) render();
    return elem;
  }

  function render() {
    var html = options.template({
      title: options.title
    });

    elem = document.createElement('div');
    elem.innerHTML = html;
    elem = elem.firstChild;

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

    elem.onclick = function(event) {
      if (event.target.closest('.title')) {
        toggle();
      }
    }
  }

  function renderItems() {
    if (elem.querySelector('ul')) return;

    var listHtml = options.listTemplate({
      items: options.items
    });
    elem.insertAdjacentHTML("beforeEnd", listHtml);
  }

  function open() {
    renderItems();
    elem.classList.add('open');
  };

  function close() {
    elem.classList.remove('open');
  };

  function toggle() {
    if (elem.classList.contains('open')) close();
    else open();
  };

  this.getElem = getElem;
  this.toggle = toggle;
  this.close = close;
  this.open = open;
}
}
```

Результат:

► Сладости

Здесь два шаблона. Первый мы уже разобрали, посмотрим теперь на список ul/li:

```
<ul>
  <% items.forEach(function(item) { %>
    <li><%-item%></li>
  <% }); %>
</ul>
```

Если разбить шаблон для списка элементов по разделителям, то он будет таким:

- — текст
- <% items.forEach(function(item) { %> — код
- — текст
- <%-item%> — вставить значение item с экранированием
- — текст
- <% }); %> — код
- — текст

Вот функция, которую возвратит `_.template tmpl` для этого шаблона:

```
function(obj) {
  obj || (obj = {});
  var __t, __p = '', __e = _.escape;
  with(obj) {
    __p += '\n<ul>\n ';
    items.forEach(function(item) {
      __p += '\n  <li>' +
        __e(item) + // <%-item%> экранирование функцией __.escape
        '</li>\n ';
    });
    __p += '\n</ul>\n';
  }
  return __p
}
```

Как видно, она один-в-один повторяет код и вставляет текст в переменную `__p`. При этом выражение в `<%-...%>` обёрнуто в вызов `__.escape`, который заменяет спецсимволы HTML на их текстовые варианты.

Отладка шаблонов

Что, если в шаблоне ошибка? Например, синтаксическая. Конечно, ошибки будут возникать, куда

Шаблон компилируется в функцию, ошибка будет либо при компиляции, либо позже, в процессе её выполнения. В различных шаблонных системах есть свои средства отладки, `_.template` тут не блистает.

Но и здесь можно кое-что отладить. При ошибке, если она не синтаксическая, отладчик при этом останавливается где-то посередине «страшной» функции.

Попробуйте сами запустить пример с открытыми инструментами разработчика и включённой опцией «остановка при ошибке»:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>

<script type="text/template" id="menu-template">


<%-title%>


<% items.forEach(function(item) { %>
  <li><%-item%></li>
<% }); %>
</ul>

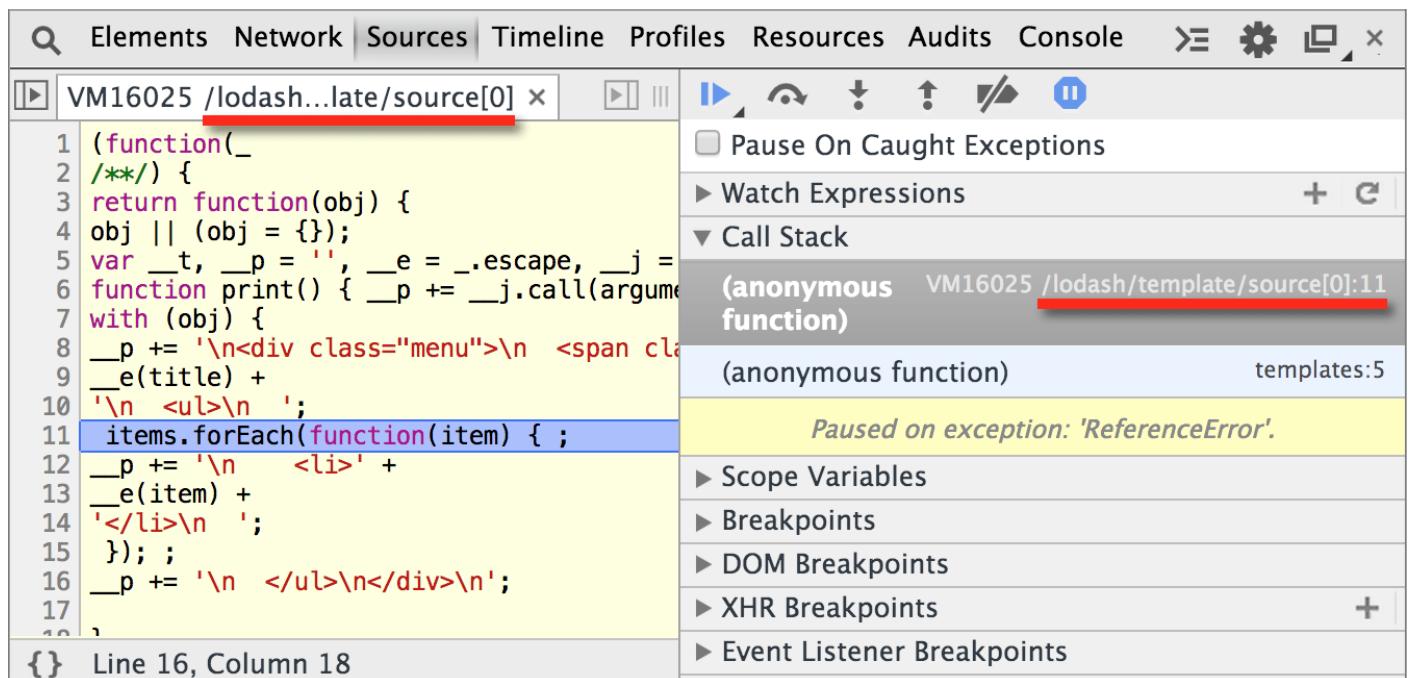

</script>
var tmpl = _.template(document.getElementById('menu-template').innerHTML);

var result = tmpl({ title: "Заголовок" });

document.write(result);
</script>
```

В шаблоне допущена ошибка, поэтому отладчик остановит выполнение.

В Chrome картина будет примерно такой:



библиотека LoDash пытается помочь, подсказав, в каком именно шаблоне произошла ошибка. Ведь из функции это может быть неочевидно.

Для этого она добавляет к шаблонам специальный идентификатор `sourceURL` ↗, который служит аналогом «имени файла». На картинке он отмечен красным.

По умолчанию `sourceURL` имеет вид `/lodash/template/source[N]`, где `N` — постоянно увеличивающийся номер шаблона. В данном случае мы можем понять, что эта функция получена при самой первой компиляции.

Это, конечно, лучше чем ничего, но, как правило, его имеет смысл заменить `sourceURL` на свой, указав при компиляции дополнительный параметр `sourceURL`:

```
...
var compiled = _.template(template, {sourceURL: '/template/menu-template'});
...
```

Попробуйте запустить [исправленный пример](#) ↗ и вы увидите в качестве имени файла `/template/menu-template`.

⚠ Не определена переменная — ошибка

Кстати говоря, а в чём же здесь ошибка?

...А в том, что переменная `items` не передана в шаблон. При доступе к неизвестной переменной JavaScript генерирует ошибку.

Самый простой способ это обойти — обращаться к необязательным переменным через `obj`, например `<%=obj.items%>`. Тогда в случае `undefined` просто ничего не будет выведено. Но в данном случае реакция совершенно адекватна, так как для меню список опций `items` является обязательным.

Итого

Шаблоны полезны для того, чтобы отделить HTML от кода. Это упрощает разработку и поддержку.

В этой главе подробно разобрана система шаблонизации из библиотеки [LoDash](#) ↗:

- Шаблон — это строка со специальными вставками кода `<% ... %>` или переменных `<%- expr ->`, `<%= expr ->`.
- Вызов `_.template(template)` превращает шаблон `template` в функцию, которой в дальнейшем передаются данные — и она генерирует HTML с ними.

В этой главе мы рассмотрели хранение шаблонов в документе, при помощи `<script>` с нестандартным `type`. Конечно, есть и другие способы, можно хранить шаблоны и в отдельном файле, если шаблонная система или система сборки проектов это позволяют.

шаблонных систем много. Одна из основных на склоне принципов — генерации функций из строк, например:

- EJS ↗
- Jade ↗
- Handlebars ↗

Есть и альтернативный подход — шаблонная система получает «образец» DOM-узла и клонирует его вызовом `cloneNode(true)`, каждый раз изменяя что-то внутри. В отличие от подхода, описанного выше, это будет работать не с произвольной строкой текста, а только и именно с DOM-узлами. Но в некоторых ситуациях у него есть преимущество.

Такой подход используется во фреймворках:

- AngularJS ↗
- Knockout.js ↗

✓ Задачи

Шаблон для таблицы с пользователями

важность: 5

Есть данные:

```
var users = [  
  {name: "Вася", age: 10},  
  {name: "Петя", age: 15},  
  {name: "Женя", age: 20},  
  {name: "Маша", age: 25},  
  {name: "Даша", age: 30},  
];
```

Выведите их в виде таблицы TABLE/TR/TD при помощи шаблона.

Результат:

Имя	Возраст
Вася	10
Петя	15
Женя	20
Маша	25
Даша	30

Открыть песочницу для задачи. ↗

Шаблон в div с display:none?

важность: 5

Что лучше для размещения шаблона — `<script>` с нестандартным `type` или `<div>` с `display:none`?

Почему? Есть ли какие-то другие хорошие варианты?

[К решению](#)

Сделайте меню ссылками

важность: 5

Возьмите в качестве исходного кода меню на шаблонах и модифицируйте его, чтобы оно выводило не просто список, а список ссылок.

1. Вместо массива `items` меню должно принимать *объект* `items`, вот так:

```
var menu = new Menu({
  title: "Сладости",
  template: _.template(document.getElementById('menu-template').innerHTML),
  listTemplate: _.template(document.getElementById('menu-list-template').innerHTML),
  items: {
    "donut": "Пончик",
    "cake": "Пирожное",
    "chocolate": "Шоколадка"
  }
});
```

- Вывод в шаблоне пусть будет не просто `Пончик`, а через ссылку: `Пончик`. При клике на ссылку должно выводиться название из её `href`.

Демо:

▶ Сладости

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

КОЛЛБЭКИ И СОБЫТИЯ НА КОМПОНЕНТАХ

Компоненты, хоть и каждый сам по себе, обычно как-то общаются с остальной частью страницы

Есть несколько способов, при помощи которых компоненты сообщают друг другу о важных событиях, которые в них произошли.

Коллбэки

Коллбэк (от англ. callback) — это функция, которую мы передаём куда-либо и ожидаем, что она будет вызвана при наступлении события.

Например, мы можем добавить в `options` для `Menu` новый параметр — функцию `onselect`, которая будет вызываться при выборе пункта меню:

```
var menu = new Menu({
  title: "Сладости",
  template: _.template(document.getElementById('menu-template').innerHTML),
  listTemplate: _.template(document.getElementById('menu-list-template').innerHTML),
  items: {
    "donut": "Пончик",
    "cake": "Пирожное",
    "chocolate": "Шоколадка"
  },
  onselect: showSelected
});

function showSelected(href) {
  alert(href);
}
```

В коде меню нужно будет вызывать её, например так:

```
...
function select(link) {
  options.onselect(link.getAttribute('href').slice(1));
}
...
...
```

Полный пример:

[Смотреть пример онлайн ↗](#)

Свои события

Как мы уже знаем, в современных браузерах DOM-элементы могут [генерировать произвольные события](#) при помощи встроенных методов, а в IE8- это возможно с использованием фреймворка, к примеру, jQuery.

Воспользуемся ими, чтобы корневой элемент меню генерировал событие, которое мы назовём `select`, при выборе элемента, и передавал в объект события выбранное значение.

Для этого мы должны настроить обработчик события select.

```
function Menu(options) {  
  ...  
  
  function select(link) {  
    var widgetEvent = new CustomEvent("select", {  
      bubbles: true,  
      // detail - стандартное свойство CustomEvent для произвольных данных  
      detail: link.getAttribute('href').slice(1)  
    });  
    elem.dispatchEvent(widgetEvent);  
  }  
  ...  
}
```

Код, который заинтересован в том, чтобы узнавать, что выбрано в меню, подписывается на событие `select` его корневого элемента:

```
var menu = new Menu(...);  
  
var elem = menu.getElem();  
  
elem.addEventListener('select', function(event) {  
  alert( event.detail );  
});
```

Вместо `detail` можно было бы выбрать и другое название свойства, но тогда нужно позаботиться о том, чтобы оно не конфликтовало со стандартными. Кроме того, в конструкторе `CustomEvent` разрешено только `detail`, другое свойство понадобилось бы присваивать в отдельной строке.

Полный пример:

[Смотреть пример онлайн ↗](#)

Внимание, инкапсуляция!

Очень важно, что внешний код ставит обработчик на корневой элемент, но не на внутренние элементы меню.

Строго говоря, он вообще не знает про то, как устроено меню, есть ли там ссылки и какие, или там вообще всё реализовано через кнопки.

Меню для него — «чёрный ящик». Корневой элемент — точка доступа к его функционалу. Событие — не то, которое произошло на ссылке, а «переработанный вариант», интерпретация действия со стороны меню.

Такое правило позволяет нам не опасаться проблем при оптимизации, расширении и даже полной переделке DOM-структуры меню. Коль скоро события и методы сохраняются, внешний код будет работать как прежде.

Ещё раз — внешний код не имеет права залезать внутрь DOM-структуры меню, ставить там обработчики и так далее.

Итого

Для того, чтобы внешний код мог узнавать о важных событиях, произошедших внутри компоненты, используются:

- Коллбэки — функции, которые передаются «снаружи» при создании компонента, и которые он обязуется вызвать при наступлении событий.
- События — компонент генерирует их на корневом элементе при помощи `dispatchEvent`, а внешний код ставит обработчики при помощи `addEventListener`. Такие события всплывают, если указан флаг `bubbles`, поэтому можно использовать с ними делегирование.

Задачи

Голосовалка «на событиях»

важность: 5

Добавьте событие в голосовалку, созданную в задаче [Голосовалка](#), используя механизм генерации событий на объекте.

Пусть каждое изменение голоса сопровождается событием `change` со свойством `detail`, содержащим обновлённое значение:

```
var voter = new Voter({
  elem: document.getElementById('voter')
});

voter.setVote(5);

document.getElementById('voter').addEventListener('change', function(e) {
  alert( e.detail ); // новое значение голоса
});
```

Все изменения голоса должны производиться централизованно, через метод `setVote`, который и генерирует событие.

Результат использования кода выше (планируемый):

— 5 +

Исходный документ возьмите из решения задачи [Голосовалка](#).

[К решению](#)

Список с выделением и событием

важность: 5

Добавьте в решение задачи [Компонент: список с выделением](#) событие `select`.

Оно должно срабатывать при каждом изменении выбора и в свойстве `detail` содержать список выбранных строк.

Во внешнем коде добавьте обработчик к списку, который при изменениях выводит список значений.

Клик на элементе выделяет только его.

Ctrl(Cmd)+Клик добавляет/убирает элемент из выделенных.

Shift+Клик добавляет промежуток от последнего кликнутого к выделению.

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

В качестве исходного кода возьмите решение задачи [Компонент: список с выделением](#).

[К решению](#)

Свой селект

важность: 5

Напишите свой, самостоятельно оформленный, селект.

Требования:

- Открытие и закрытие по клику на заголовок.
- Закрытие селекта происходит при выборе или клике на любое другое место документа, в том числе на другой аналогичный селект.
- Событие "select" при выборе опции возникает на элементе селекта и всплывает.
- Значение опции хранится в атрибуте data-value (HTML-структура есть в исходном документе).

Например:

Последний результат: ...

Выберите

Выберите

В примере выше два селекта, чтобы можно было проверить процесс открытия-закрытия.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Слайдер с событиями

важность: 5

На основе слайдера из задачи [Слайдер-компонент](#) создайте графический компонент, который умеет возвращать/получать значение.

Синтаксис:

```
var slider = new Slider({
  elem: document.getElementById('slider'),
  max: 100 // слайдер на самой правой позиции соответствует 100
});
```

```
slider.setValue(50);
```

У слайдера должно быть два события: `slide` при каждом передвижении и `change` при отпускании мыши (установке значения).

Пример использования:

```
var sliderElem = document.getElementById('slider');

sliderElem.addEventListener('slide', function(event) {
  document.getElementById('slide').innerHTML = event.detail;
});

sliderElem.addEventListener('change', function(event) {
  document.getElementById('change').innerHTML = event.detail;
});
```

В действии:

Slide: Change:

- Ширина/высота слайдера может быть любой, JS-код это должен учитывать.
- Центр бегунка должен располагаться в точности над выбранным значением. Например, он должен быть в центре для 50 при `max=100`.

Исходный документ — возьмите решение задачи [Слайдер-компонент](#).

[К решению](#)

Что изучать дальше

Если вы прочитали весь учебник и сделали задачи, то на текущий момент вы обладаете важнейшими фундаментальными знаниями и навыками JavaScript.

В этом разделе мы изучали основы создания компонентов на JavaScript. Если проект большой и сложный, то понадобятся дополнительные инструменты для связывания компонент между собой, для привязки к ним данных и так далее.

Сейчас существует много фреймворков. Всё активно развивается, меняется, кипит и булькает, может быть из этого получится «общепринятая» архитектура, а может и нет. Сейчас явного победителя нет, выбор фреймворка зависит от проекта и личных предпочтений разработчиков.

Примеры удачных фреймворков, которые можно изучить:

- [AngularJS](#)
- [React.JS](#) + [Flux](#)
- [Backbone.JS](#) + [Marionette](#)

Также для работы с браузерами понадобятся различные [API](#), в частности:

- Работу с окнами и фреймами.
- Регулярные выражения, класс RegExp.
- Объекты XMLHttpRequest и WebSocket для работы с сервером.
- Другие возможности современных браузеров.

В дополнительных разделах учебника мы обязательно разберём что-то из этого.

...И, конечно, понадобится система сборки проектов, например [WebPack](#).

Успехов вам!

Решения

Дерево DOM

Что выведет этот alert?

Выведет null, так как на момент выполнения скрипта тег <body> ещё не обработан браузером.

Попробуйте в действии:

```
<html>
  <head>
    <script>
      alert( document.body ); // null
    </script>
  </head>

  <body>
    Привет, мир!
  </body>
</html>
```

[К условию](#)

Навигация по DOM-элементам

DOM children

HEAD

Два способа:

```
document.documentElement.children[0]  
document.documentElement.firstChild
```

Второй способ работает, так как пробелы перед `<head>` игнорируются.

Также в современных браузерах доступен `document.head`.

UL

Например, так:

```
document.body.children[1]
```

LI

Можно так:

```
document.body.children[1].children[1]; // LI
```

Может возникнуть проблема с комментарием в IE8-, так как он станет одним из `children`, в результате последний код станет работать некорректно.

В последующих разделах учебника мы рассмотрим другие методы поиска по DOM, которые позволят эту проблему обойти.

[К условию](#)

Проверка существования детей

Вначале нерабочие способы, которые могут прийти на ум:

```
if (!elem) { .. }
```

Это не работает, так как `elem` всегда есть, и является объектом. Так что проверка `if (elem)` всегда верна, вне зависимости от того, есть ли у `elem` потомки.

```
if (!elem.childNodes) { ... }
```

Тоже не работает, так как псевдо-массив `childNodes` всегда существует. Он может быть пуст или непуст, но он всегда является объектом, так что проверка `if (elem.childNodes)` всегда верна.

Несколько рабочих способов:

```
if (!elem.childNodes.length) { ... }  
if (!elem.firstChild) { ... }  
if (!elem.lastChild) { ... }
```

Также существует метод [hasChildNodes](#), который позволяет вызовом `elem.hasChildNodes()` определить наличие детей. Он работает так же, как проверка `elem.childNodes.length != 0`.

[К условию](#)

Вопрос по навигационным ссылкам

1. Да, верно, с оговоркой. Элемент `elem.lastChild` последний, у него нет правого соседа.

Оговорка: `elem.lastChild.nextSibling` выдаст ошибку если `elem` не имеет детей.

2. Нет, неверно, это может быть текстовый узел. Значением `elem.children[0]` является первый узел-элемент, перед ним может быть текст.

Аналогично предыдущему случаю, если у `elem` нет детей-элементов — будет ошибка.

[К условию](#)

Выделите ячейки по диагонали

Для удобства работы с таблицей используем специальные свойства `rows` и `cells`.

[Открыть решение в песочнице](#)

[К условию](#)

ПОИСК ЭЛЕМЕНТОВ

Есть много вариантов решения, вот некоторые из них:

```
// 1
document.getElementById('age-table').getElementsByTagName('label');

// 2
document.getElementById('age-table').getElementsByTagName('td')[0];
// в современных браузерах можно одним запросом:
var result = document.querySelector('#age-table td');

// 3
document.getElementsByName('form')[1];

// 4
document.querySelector('form[name="search"]');

// 5
document.querySelector('form[name="search-person"] input')

// 6
document.getElementsByName("info[0]")[0];

// 7
document.querySelector('form[name="search-person"] [name="info[0]"]');
```

[К условию](#)

Дерево

Сделаем цикл по узлам ``:

```
var lis = document.getElementsByTagName('li');
for (i = 0; i < lis.length; i++) {
  ...
}
```

В цикле для каждого `lis[i]` можно получить текст, используя свойство `firstChild`. Ведь первым в `` является как раз текстовый узел, содержащий текст названия.

Также можно получить количество потомков, используя `lis[i].getElementsByTagName('li')`.

Напишите код с этой подсказкой.

Если уж не выйдет — тогда откройте решение.

[К условию](#)

Внутреннее устройство поисковых методов

Длина коллекции после удаления элементов

Ответ на первый вопрос

Ответ: 0, пустая коллекция.

```
<ul id="menu">
  <li>Главная страница</li>
  <li>Форум</li>
  <li>Магазин</li>
</ul>
<script>
  var lis = document.body.getElementsByTagName('li');

  document.body.innerHTML = "";

  alert( lis.length );
</script>
```

Это потому, что все элементы из BODY удаляются, а коллекция – живая.

Ответ на второй вопрос

Ответ на второй вопрос зависит от браузера. В большинстве браузеров будет 3, коллекция не изменилась, так как она теперь привязана не к BODY, а к элементу, на котором идёт поиск, т.е. к menu.

Но элемент menu находится в переменной, и поэтому должен быть жив, а значит и его дети тоже. Но некоторые браузеры (IE10) используют агрессивный подход при работе с памятью и очищают все элементы, кроме тех, которые непосредственно хранятся в переменных.

Поэтому результат кода ниже в большинстве браузеров: 3, а в IE10: 0.

```
<ul id="menu">
  <li>Главная страница</li>
  <li>Форум</li>
  <li>Магазин</li>
</ul>
<script>
  var menu = document.getElementById('menu');
  var lis = menu.getElementsByTagName('li');

  document.body.innerHTML = "";

  alert( lis.length );
</script>
```

[К условию](#)

Сравнение количества элементов

Значение `aList1` изменится, потому что `getElementsByTagName` – живая коллекция. Она автоматически дополнится новым элементом а и ее длина увеличится на 1.

А вот `querySelector`, наоборот, возвращает статичный список узлов. Он ссылается на те же самые элементы, что бы не происходило с документом. Поэтому длина `aList2.length` останется неизменной.

[К условию](#)

Бенчмаркинг методов поиска в DOM

Для бенчмаркинга будем использовать функцию `bench(f, times)`, которая запускает функцию `f` `times` раз и возвращает разницу во времени:

```
function bench(f, times) {
  var d = new Date();
  for (var i = 0; i < times; i++) f();
  return new Date() - d;
}
```

Первый вариант (неверный) — замерять разницу между функциями `runGet/runQuery`, вот так:

```
function runGet() {
  var results = document.getElementsByTagName('p');
}

function runQuery() {
  var results = document.querySelectorAll('p');
}

alert( bench(runGet, 10000) ); // вывести время 1000*runGet
```

Он даст неверные результаты, т.к. `getElementsByTagName` является «живым поисковым запросом». Если не обратиться к его результатам, то поиска не произойдет вообще, т.е. `runGet` ничего по сути не ищет.

...А `querySelectorAll` всегда производит поиск и формирует список элементов.

Более правильный тест — это не только запустить поиск, но и получить все элементы, как это делается в реальной жизни.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Получить второй LI

Можно так:

```
var li = ul.getElementsByTagName('li')[1];
```

Или так:

```
var li = ul.querySelector('li:nth-child(2)');
```

Оба этих вызова будут перебирать детей UL и остановят перебор на найденном элементе.

А вот так — браузер найдет все элементы, а затем выберет второй. Это дольше:

```
var li = ul.querySelectorAll('li')[1];
```

На практике разница в производительности будет видна только для действительно больших списков, либо при частом выполнении запроса. Браузер перебирает элементы весьма шустро.

[К условию](#)

Что выведет код в консоли?

Однозначно правильный ответ невозможен.

В консоли не выводятся пробельные узлы. Если перед `<h1>` находится пробельный узел, то будет `undefined`, а если нет — то текст внутри `<h1>`.

Пример с `undefined`:

```
<body>
  <h1>Привет, мир!</h1>

  <script>
    alert( document.body.firstChild.innerHTML ); // undefined
  </script>
</body>
```

Если убрать из него перевод строки перед `<h1>`, то было бы "Привет, мир!".

[К условию](#)

В инлайн скрипте `lastChild.nodeType`

Небольшой подвох — в том, что во время выполнения скрипта последним тегом является `SCRIPT`. Браузер не может обработать страницу дальше, пока не выполнит скрипт.

Так что результат будет 1 (узел-элемент).

```
<!DOCTYPE HTML>
<html>

  <body>
    <script>
      alert( document.body.lastChild.nodeType );
    </script>
  </body>

</html>
```

[К условию](#)

Тег в комментарии

Ответ: **BODY**.

```
<script>
  var body = document.body;

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // BODY
</script>
```

Происходящее по шагам:

1. Заменяем содержимое `<body>` на комментарий. Он будет иметь вид `<!--BODY-->`, так как `body.tagName == "BODY"`. Как мы помним, свойство `tagName` в HTML всегда находится в верхнем регистре.
2. Этот комментарий теперь является первым и единственным потомком `body.firstChild`.
3. Получим значение `data` для комментария `body.firstChild`. Оно равно содержимому узла для всех узлов, кроме элементов. Содержимое комментария: "BODY".

[К условию](#)

Где в DOM-иерархии `document`?

Объектом какого класса является `document`, можно выяснить так:

```
alert(document); // [object HTMLDocument]
```

Или так:

```
alert(document.constructor); // function HTMLDocument() { ... }
```

Итак, `document` — объект класса `HTMLDocument`.

Какое место `HTMLDocument` занимает в иерархии?

Можно поискать в документации. Но попробуем выяснить это самостоятельно.

Вопрос не такой простой и требует хорошего понимания [прототипного наследования](#).

Вспомним, как оно устроено:

- Методы объекта document находятся в структуре конструктора, в данном случае `HTMLDocument.prototype`.
- У `HTMLDocument.prototype` есть ссылка `__proto__` на прототип-родитель.
- У прототипа-родителя может быть ссылка `__proto__` на его родитель, и так далее.

При поиске свойства в `document`, если его там нет, оно ищется в `document.__proto__`, затем в `document.__proto__.__proto__` и так далее, пока не найдём, или пока цепочка `__proto__` не закончится. Это обычное устройство класса, без наследования.

Нам нужно лишь узнать, что находится в этих самых `__proto__`.

Строго говоря, там могут быть любые объекты. Вовсе не обязательно, чтобы объектам из цепочки прототипов соответствовали какие-то конструкторы.

Вполне может быть цепочка, где родители — просто обычные JS-объекты:

```
document -> HTMLDocument.prototype -> obj1 -> obj2 -> ...
```

Однако, здесь мы знаем, что наследование — «на классах», то есть, эти объекты `obj1`, `obj2` являются `prototype` неких функций-конструкторов:

```
document -> HTMLDocument.prototype -> F1.prototype -> F2.prototype -> ...
```

Что стоит на месте `F1`, `F2`?

Опять же, если говорить про некие абстрактные объекты, то откуда нам знать, какие функции на них ссылаются через `prototype`? Ниоткуда. Один объект может быть в `prototype` хоть у десятка функций.

Но в стандартном прототипном наследовании один объект является `prototype` ровно у одной функции. Причём при создании функции в её `prototype` уже есть объект со свойством `constructor`, которое ссылается обратно на функцию:

```
F.prototype = { constructor: F }
```

Это свойство `constructor`, если конечно его не удалить или не перезаписать нечаянно (чего делать не следует), и позволяет из прототипа узнать соответствующий ему конструктор.

```
// цепочка наследования:  
alert(HTMLDocument.prototype.constructor); // function HTMLDocument  
alert(HTMLDocument.prototype.__proto__.constructor); // function Document  
alert(HTMLDocument.prototype.__proto__.__proto__.constructor); // function Node
```

При выводе объекта через `console.dir(document)` в Google Chrome, мы тоже можем, раскрывая `__proto__`, увидеть эти названия (`HTMLDocument`, `Document`, `Node`).

[К условию](#)

Современный DOM: полифиллы

Полифилл для matches

Код для полифилла здесь особенно прост.

Реализовывать ничего не надо, просто записать нужный метод в `Element.prototype.matches`, если его там нет:

```
(function() {  
    // проверяем поддержку  
    if (!Element.prototype.matches) {  
  
        // определяем свойство  
        Element.prototype.matches = Element.prototype.matchesSelector ||  
            Element.prototype.webkitMatchesSelector ||  
            Element.prototype.mozMatchesSelector ||  
            Element.prototype.msMatchesSelector,  
  
    }  
})();
```

[К условию](#)

Полифилл для closest

Код для этого полифилла имеет стандартный вид:

```
(function() {  
    // проверяем поддержку  
    if (!Element.prototype.closest) {  
  
        // реализуем  
        Element.prototype.closest = function(css) {  
            var node = this;  
  
            while (node) {  
                if (node.matches(css)) return node;  
                else node = node.parentElement;  
            }  
            return null;  
        };  
    }  
})();
```

Обратим внимание, что код этого полифилла использует `node.matches`, то есть может в свою очередь потребовать полифилла для него. Это типичная ситуация — один полифилл тянет за собой другой. Именно поэтому сервисы и библиотеки полифиллов очень полезны.

[К условию](#)

Полифилл для `textContent`

Код для полифилла здесь имеет стандартный вид:

```
(function() {  
    // проверяем поддержку  
    if (document.documentElement.textContent === undefined) {  
  
        // определяем свойство  
        Object.defineProperty(HTMLElement.prototype, "textContent", {  
            get: function() {  
                return this.innerText;  
            },  
            set: function(value) {  
                this.innerText = value;  
            }  
        });  
    }  
})();
```

Единственный тонкий момент — в проверке поддержки.

Мы часто можем использовать уже существующий элемент. В частности, при проверке `firstElementChild` мы можем проверить его наличие в `document.documentElement`.

Однако, в данном случае попытка получить `document.documentElement.textContent` при поддержке этого свойства приведёт к совершенно лишним затратам времени и памяти,

так как браузер будет динамически пересортировать строку из содержимого документа.

Поэтому лучше бы использовать пустой DOM-элемент. Но это лишь оптимизация, общий подход верен.

[К условию](#)

Атрибуты и DOM-свойства

Получите пользовательский атрибут

```
<body>

  <div id="widget" data-widget-name="menu">Выберите жанр</div>

  <script>
    var div = document.getElementById('widget');

    var widgetName = div.getAttribute('data-widget-name');
    // или так, кроме IE10-
    var widgetName = div.dataset.widgetName;

    alert( widgetName ); // "menu"
  </script>
</body>
```

[Открыть решение в песочнице ↗](#)

[К условию](#)

Поставьте класс ссылкам

Сначала можно найти ссылки, например, при помощи `document.querySelectorAll('a')`, а затем выбрать из них нужные.

Затем определимся — что использовать для проверки адреса ссылки: свойство `href` или атрибут `getAttribute('href')`?

Различие между ними заключается в том, что свойство будет содержать полный путь ссылки, а атрибут — значение, указанное в HTML.

Если открыть страницу локально, на диске, то для `` значения будут такими:

- `a.getAttribute('href') == "/tutorial".`

Здесь нужен именно атрибут, хотя бы потому, что в свойстве все ссылки уже с хостом и протоколом, а нам надо понять, был ли протокол в href или нет.

Правила определения:

- Ссылки без href и без протокола :// являются заведомо внутренними.
- Там, где протокол есть — проверяем, начинается ли адрес с http://internal.com.

Итого, код может быть таким:

```
var links = document.querySelectorAll('a');

for (var i = 0; i < links.length; i++) {

    var a = links[i];

    var href = a.getAttribute('href');

    if (!href) continue; // нет атрибута

    if (href.indexOf('://') == -1) continue; // без протокола

    if (href.indexOf('http://internal.com') === 0) continue; // внутренняя

    a.classList.add('external');
}
```

...Но, как это часто бывает, знание CSS может упростить задачу. Удобнее и эффективнее здесь — указать проверки для href прямо в CSS-селекторе:

```
// ищем все ссылки, у которых в href есть протокол,
// но адрес начинается не с http://internal.com
var css = 'a[href*="://"]:not([href^="http://internal.com"]);';
var links = document.querySelectorAll(css);

for (var i = 0; i < links.length; i++) {
    links[i].classList.add('external');
}
```

[Открыть решение в песочнице ↗](#)

[К условию](#)

Добавление и удаление узлов

createTextNode vs innerHTML

Результат выполнения может быть разный: innerHTML вставит именно HTML, а

Запустите следующие примеры, чтобы увидеть разницу:

- `createTextNode` создает текст '**текст**':

```
<div id="elem"></div>
<script>
    var text = '<b>текст</b>';

    elem.appendChild(document.createTextNode(text));
</script>
```

- `innerHTML` присваивает HTML **текст**:

```
<div id="elem"></div>
<script>
    var text = '<b>текст</b>';

    elem.innerHTML = text;
</script>
```

[К условию](#)

Удаление элементов

Родителя `parentNode` можно получить из `elem`.

Вот так выглядит решение:

```
<div>Это</div>
<div>Все</div>
<div>Элементы DOM</div>

<script>
    if (!Element.prototype.remove) {
        Element.prototype.remove = function remove() {
            if (this.parentNode) {
                this.parentNode.removeChild(this);
            }
        };
    }

    var elem = document.body.children[0];

    elem.remove();
</script>
```

[К условию](#)

insertAfter

Для того, чтобы добавить элемент *после* `refElem`, мы можем, используя `insertBefore`, вставить его *перед* `refElem.nextSibling`.

Но что если `nextSibling` нет? Это означает, что `refElem` является последним потомком своего родителя и можем использовать `appendChild`.

Код:

```
function insertAfter(elem, refElem) {
  var parent = refElem.parentNode;
  var next = refElem.nextSibling;
  if (next) {
    return parent.insertBefore(elem, next);
  } else {
    return parent.appendChild(elem);
  }
}
```

Но код может быть гораздо короче, если вспомнить, что `insertBefore` со вторым аргументом `null` работает как `appendChild`:

```
function insertAfter(elem, refElem) {
  return refElem.parentNode.insertBefore(elem, refElem.nextSibling);
}
```

Если нет `nextSibling`, то второй аргумент `insertBefore` становится `null` и тогда `insertBefore(elem, null)` осуществит вставку в конец, как и требуется.

В решении нет проверки на существование `refElem.parentNode`, поскольку вставка после элемента без родителя — уже ошибка, пусть она возникнет в функции, это нормально.

[К условию](#)

removeChildren

Неправильное решение

Для начала рассмотрим забавный пример того, как делать *не надо*:

```
function removeChildren(elem) {
  for (var k = 0; k < elem.childNodes.length; k++) {
    elem.removeChild(elem.childNodes[k]);
  }
}
```

Если вы попробуете это на практике, то увидите, то это не сработает.

Не сработает потому, что коллекция `childNodes` всегда начинается с индекса 0 и автоматически обновляется, когда первый потомок удален(т.е. тот, что был вторым, станет первым). А переменная `k` в цикле всё время увеличивается, поэтому такой цикл пропустит половину узлов.

Решение через DOM

Правильное решение:

```
function removeChildren(elem) {
  while (elem.lastChild) {
    elem.removeChild(elem.lastChild);
  }
}
```

Альтернатива через `innerHTML`

Можно и просто обнулить содержимое через `innerHTML`:

```
function removeChildren(elem) {
  elem.innerHTML = '';
}
```

Это не будет работать в IE8- для таблиц, так как на большинстве табличных элементов (кроме ячеек TH/TD) в старых IE запрещено менять `innerHTML`.

Впрочем, можно завернуть `innerHTML` в `try/catch`:

```
function removeChildren(elem) {
  try {
    elem.innerHTML = '';
  } catch (e) {
    while (elem.firstChild) {
      elem.removeChild(elem.firstChild);
    }
  }
}
```

[К условию](#)

Почему остаётся «aaa» ?

HTML в виде HTML некорректен. В этом все дело. У вопроса есть решение, если открыть отладчик.

В нём видно, что браузер поместил текст `aaa` перед таблицей. Поэтому он и остался в документе.

Вообще, в стандарте HTML5 описано, как браузеру обрабатывать некорректный HTML, так что такое действие браузера является правильным.

[К условию](#)

Создать список

Делаем цикл, пока посетитель что-то вводит — добавляет ``.

Содержимое в `` присваиваем через `document.createTextNode`, чтобы правильно работали `<`, `>` и т.д.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Создайте дерево из объекта

Решения через рекурсию.

1. [Через innerHTML ↗](#).
2. [Через DOM ↗](#).

[Открыть решение в песочнице ↗](#)

[К условию](#)

Дерево

Подсказки

1. Получить количество вложенных узлов можно через `elem.getElementsByTagName('*').length`.
2. Текст в начале `` доступен как `li.firstChild`, его содержимое —

Решение

[Открыть решение в песочнице ↗](#)

[К условию](#)

Создать календарь в виде таблицы

Для решения задачи сгенерируем таблицу в виде строки: "<table>...</table>", а затем присвоим в innerHTML.

Алгоритм:

1. Создать объект даты `d = new Date(year, month-1)`. Это первый день месяца `month` (с учетом того, что месяцы в JS начинаются от 0, а не от 1).
2. Ячейки первого ряда пустые от начала и до дня недели `d.getDay()`, с которого начинается месяц. Создадим их.
3. Увеличиваем день в `d` на единицу: `d.setDate(d.getDate()+1)`, и добавляем в календарь очередную ячейку, пока не достигли следующего месяца. При этом последний день недели означает вставку перевода строки "</tr><tr>".
4. При необходимости, если календарь окончился не на воскресенье – добавить пустые TD в таблицу, чтобы было все ровно.

[Открыть полное решение ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Часики с использованием «setInterval»

Для начала, придумаем подходящую HTML/CSS-строктуру.

Здесь каждый компонент времени удобно поместить в соответствующий SPAN:

```
<div id="clock">
  <span class="hour">hh</span>:<span class="min">mm</span>:<span class="sec">ss</span>
</div>
```

Каждый SPAN раскрашивается при помощи CSS.

```
function update() {
    var clock = document.getElementById('clock');

    var date = new Date(); // (*)

    var hours = date.getHours();
    if (hours < 10) hours = '0' + hours;
    clock.children[0].innerHTML = hours;

    var minutes = date.getMinutes();
    if (minutes < 10) minutes = '0' + minutes;
    clock.children[1].innerHTML = minutes;

    var seconds = date.getSeconds();
    if (seconds < 10) seconds = '0' + seconds;
    clock.children[2].innerHTML = seconds;
}
```

В строке (*) каждый раз мы получаем текущую дату. Мы должны это сделать, несмотря на то, что, казалось бы, могли бы просто увеличивать счетчик каждую секунду.

На самом деле мы не можем опираться на счетчик для вычисления даты, т.к. `setInterval` не гарантирует точную задержку. Если в другом участке кода будет вызван `alert`, то часы остановятся, как и любые счетчики.

Функция `clockStart` для запуска часов:

```
function clockStart() { // запустить часы
    setInterval(update, 1000);
    update(); // (*)
}

function clockStop() {
    clearInterval(timerId);
    timerId = null;
}
```

Обратите внимание, что вызов `update` не только запланирован, но и тут же производится тут же в строке (*). Иначе посетителю пришлось бы ждать до первого выполнения `setInterval`, то есть целую секунду.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Мультивставка: `insertAdjacentHTML` и `DocumentFragment`

Вставьте элементы в конец списка

```
var ul = document.body.children[0];  
ul.insertAdjacentHTML("beforeEnd", "<li>3</li><li>4</li><li>5</li>");
```

[К условию](#)

Отсортировать таблицу

Для сортировки нам поможет функция `sort` массива.

Общая идея лежит на поверхности: сделать массив из строк и отсортировать его. Тонкости кроются в деталях.

В iframe ниже загружен документ, описывающий и реализующий разные алгоритмы. Обратите внимание: разница в производительности может достигать нескольких раз!

Алгоритм 1.

1. Все TR удалить из таблицы, при этом собрав их в JavaScript-массив.
2. Отсортировать этот массив, используя свою функцию в `sort(...)` для сравнения TR
3. Добавить TR из массива в таблицу в нужном порядке

[Померять время](#)

Алгоритм 2.

1. Скопировать TR в JavaScript-массив.
2. Отсортировать этот массив, используя свою функцию в `sort(...)` для сравнения TR
3. Добавить TR из массива в таблицу в нужном порядке. При добавлении каждый TR сам удаляется с предыдущего места.

[Померять время](#)

Алгоритм 3.

1. Создать массив из объектов вида `{elem: ссылка на TR, value: содержимое TR}`.
2. Отсортировать массив по `value`. Функция сравнения во время сортировки теперь будет обращаться не к `innerHTML`, а к свойству объекта, это быстрее. Сортировка может потребовать многократных сравнений одного и того же элемента, отсюда выигрыш.
3. Добавить TR в таблицу в нужном порядке (автоудаляются с предыдущего места).

[Померять время](#)

Алгоритм 4.

1. Выполнить алгоритм 3, но перед этим удалить таблицу из документа, а после - вставить обратно.

[Померять время](#)

Алгоритм 5.

1. Замерить время генерации таблицы (создаётся строка и пишется в `innerHTML`).

[Померять время](#)

P.S. Создавать `DocumentFragment` здесь ни к чему. Можно вытащить из документа `TBODY` и иметь дело с ним в отрыве от DOM (алгоритм 4).

P.P.S. Если нужно сделать много узлов, то обычно `innerHTML` работает быстрее, чем удаление и вставка элементов через DOM-вызовы. То есть, сгенерировать таблицу заново эффективнее.

[Открыть решение в песочнице ↗](#)

Стили, getComputedStyle

Скругленная кнопка со стилями из JavaScript

Есть два варианта.

1. Можно использовать свойство `elem.style.cssText` и присвоить стиль в текстовом виде. При этом все присвоенные ранее свойства `elem.style` будут удалены.
2. Можно назначить подсвойства `elem.style` одно за другим. Этот способ более безопасен, т.к. меняет только явно присваиваемые свойства.

Мы выберем второй путь.

[Открыть решение ↗](#)

Описание CSS-свойств:

```
.button {  
    -moz-border-radius: 8px;  
    -webkit-border-radius: 8px;  
    border-radius: 8px;  
    border: 2px groove green;  
    display: block;  
    height: 30px;  
    line-height: 30px;  
    width: 100px;  
    text-decoration: none;  
    text-align: center;  
    color: red;  
    font-weight: bold;  
}
```

*-border-radius

Добавляет скругленные углы. Свойство присваивается в вариантах для Firefox `-moz-...`, Chrome/Safari `-webkit-...` и стандартное CSS3-свойство для тех, кто его поддерживает (Opera).

display

По умолчанию, у А это свойство имеет значение `display: inline`.

height, line-height

Устанавливает высоту и делает текст вертикально центрированным путем установки `line-height` в значение, равное высоте. Такой способ центрирования текста работает, если он состоит из одной строки.

СОДЕЙСТВИЕ

Центрирует текст горизонтально.

color, font-weight

Делает текст красным и жирным.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Создать уведомление

[Открыть решение в песочнице ↗](#)

[К условию](#)

Размеры и прокрутка элементов

Найти размер прокрутки снизу

Решение: `elem.scrollHeight - elem.scrollTop - elem.clientHeight.`

[К условию](#)

Узнать ширину полосы прокрутки

Создадим элемент с прокруткой, но без padding. Тогда разница между его полной шириной `offsetWidth` и внутренней `clientWidth` будет равна как раз прокрутке:

```
// создадим элемент с прокруткой
var div = document.createElement('div');

div.style.overflowY = 'scroll';
div.style.width = '50px';
div.style.height = '50px';

// при display:none размеры нельзя узнать
// нужно, чтобы элемент был видим,
// visibility:hidden - можно, т.к. сохраняет геометрию
div.style.visibility = 'hidden';

document.body.appendChild(div);
var scrollWidth = div.offsetWidth - div.clientWidth;
document.body.removeChild(div);

alert( scrollWidth );
```

К условию

Подменить div на другой с таким же размером

Нам нужно создать div с такими же размерами и вставить его на место «переезжающего».

Один из вариантов — это просто клонировать элемент.

Если делать это при помощи `div.cloneNode(true)`, то склонируется все содержимое, которого может быть много. Обычно нам это не нужно, поэтому можно использовать `div.cloneNode(false)` для клонирования элемента со стилями, и потом поправить его `width/height`.

Можно и просто создать новый div и поставить ему нужные размеры.

Всё, кроме margin, можно получить из свойств DOM-элемента, а margin — только через `getComputedStyle`.

Причём margin мы обязаны поставить, так как иначе наш элемент при вставке будет вести себя иначе, чем исходный.

Код:

```
var div = document.getElementById('moving-div');

var placeHolder = document.createElement('div');
placeHolder.style.height = div.offsetHeight + 'px';
// можно и width, но в этом примере это не обязательно

// IE || другой браузер
var computedStyle = div.currentStyle || getComputedStyle(div, '');

placeHolder.style.marginTop = computedStyle.marginTop; // (1)
placeHolder.style.marginBottom = computedStyle.marginBottom;
```

В строке (4) используйте полное название свойства — `clientWidth`. Гарантируется, что полученное значение будет корректным.

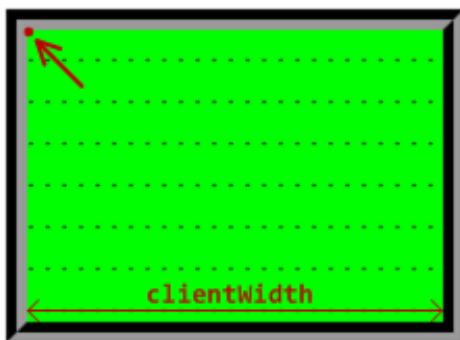
[Открыть решение в песочнице ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Поместите мяч в центр поля

При абсолютном позиционировании мяча внутри поля его координаты `left/top` отсчитываются от **внутреннего** угла поля, например верхнего-левого:



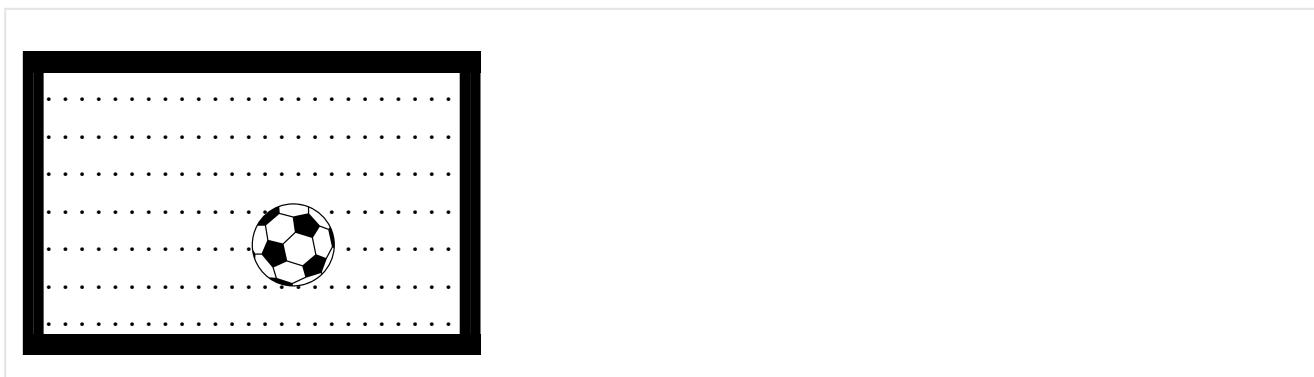
Метрики для внутренней зоны поля — это `clientWidth/Height`.

Центр — это `(clientWidth/2, clientHeight/2)`.

Но если мы установим мячу такие значения `ball.style.left/top`, то в центре будет не сам мяч, а его левый верхний угол:

```
var ball = document.getElementById('ball');
var field = document.getElementById('field');

ball.style.left = Math.round(field.clientWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2) + 'px';
```



Для этого, чтобы центр мяча находился в центре поля, нам нужно сместить мяч на половину его ширины влево и на половину его высоты вверх.

```
var ball = document.getElementById('ball');
var field = document.getElementById('field');

ball.style.left = Math.round(field.clientWidth / 2 - ball.offsetWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight / 2) + 'px';
```

Внимание, подводный камень!

Код выше стабильно работать не будет, потому что IMG идет без ширины/высоты:

```

```

Высота и ширина изображения неизвестны браузеру до тех пор, пока оно не загрузится, если размер не указан явно.

После первой загрузки изображение уже будет в кеше браузера, и его размеры будут известны. Но когда браузер впервые видит документ — он ничего не знает о картинке, поэтому значение ball.offsetWidth равно 0. Вычислить координаты невозможно.

Чтобы это исправить, добавим width/height к картинке:

```

```

Теперь браузер всегда знает ширину и высоту, так что все работает. Тот же эффект дало бы указание размеров в CSS.

[Полный код решения ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Расширить элемент

Решение через width: auto

Вначале рассмотрим решение через «умную» установку CSS-свойства.

Они могут быть разными. Самое простое выглядит так:

```
elem.style.width = 'auto';
```

Такой способ работает, так как <div> по умолчанию распахивается на всю ширину.

Конечно, такое решение не будет работать для элементов, которые сами по себе не растягиваются, например в случае со `` или при наличии `position: absolute`.

Обратим внимание, такой вариант был бы неверен:

```
elem.style.width = '100%';
```

По умолчанию в CSS ширина `width` — это то, что *внутри padding*, а проценты отсчитываются от ширины родителя. То есть, ставя ширину в 100%, мы говорим: «внутренняя область должна занимать 100% ширины родителя». А в элементе есть ещё `padding`, которые в итоге вылезут наружу.

Можно бы поменять блочную модель, указав `box-sizing` через свойство `elem.style.boxSizing`, но такое изменение потенциально может затронуть много других свойств, поэтому нежелательно.

Точное вычисление

Альтернатива — вычислить ширину родителя через `clientWidth`.

Доступную внутреннюю ширину родителя можно получить, вычитая из `clientWidth` размеры `paddingLeft/paddingRight`, и затем присвоить её элементу:

```
var bodyClientWidth = document.body.clientWidth;  
var style = getComputedStyle(elem);  
  
var bodyInnerWidth = bodyClientWidth - parseInt(style.paddingLeft) - parseInt(style.paddingRight);  
elem.style.width = bodyInnerWidth + 'px';
```

Такое решение будет работать всегда, вне зависимости от типа элемента. Конечно, при изменении размеров окна браузера ширина не адаптируется к новому размеру автоматически, как с `width:auto`. Это недостаток. Его, конечно, тоже можно обойти при помощи событий (изучим далее), но как общий рецепт — если CSS может решить задачу — лучше использовать CSS.

[Открыть решение в песочнице ↗](#)

[К условию](#)

В чём отличие «width» и «clientWidth» ?

Отличия:

1. `getComputedStyle` не работает в IE8-.

2. клиентский возвращает число, а браузеристует `(...).width` строку, на конце ря.
3. `getComputedStyle` не всегда даст ширину, он может вернуть, к примеру, "auto" для инлайнового элемента.
 4. `clientWidth` соответствует внутренней видимой области элемента, *включая padding, а CSS-ширина width при стандартном значении box-sizing соответствует зоне *внутри padding.*
 5. Если есть полоса прокрутки, то некоторые браузеры включают её ширину в `width`, а некоторые — нет.

Свойство `clientWidth`, с другой стороны, полностью кросс-браузерно. Оно всегда обозначает размер *за вычетом прокрутки*, т.е. реально доступный для содержимого.

[К условию](#)

Размеры и прокрутка страницы

Полифилл для pageYOffset в IE8

В стандартном режиме IE8 можно получить текущую прокрутку так:

```
alert( document.documentElement.scrollTop );
```

Самым простым, но неверным было бы такое решение:

```
// "полифилл"  
window.pageYOffset = document.documentElement.scrollTop;  
  
// использование "полифилла"  
alert( window.pageYOffset );
```

Код выше не учитывает текущую прокрутку. Он присваивает `window.pageYOffset` один раз и в дальнейшем, чтобы получить текущую прокрутку, нужно снова обратиться к `document.documentElement.scrollTop` не меняет его. А задача как раз — сделать полифилл, то есть дать возможность использовать `window.pageYOffset` для получения текущего состояния прокрутки без «танцев бубном», так же как в современных браузерах.

Для этого создадим свойство через геттер.

В IE8 для DOM-объектов работает `Object.defineProperty`:

```
// полифилл
Object.defineProperty(window, 'pageYOffset', {
  get: function() {
    return document.documentElement.scrollTop;
  }
});

// использование полифилла
alert( window.pageYOffset );
```

[К условию](#)

Координаты в окне

Найдите координаты точки в документе

Координаты внешних углов

Координаты элемента возвращаются функцией `elem.getBoundingClientRect`. Она возвращает все координаты относительно окна в виде объекта со свойствами `left`, `top`, `right`, `bottom`. Некоторые браузеры также добавляют `width`, `height`.

Так что координаты верхнего-левого `coords1` и правого-нижнего `coords4` внешних углов:

```
var coords = elem.getBoundingClientRect();
var coords1 = [coords.left, coords.top];
var coords2 = [coords.right, coords.bottom];
```

Левый-верхний угол внутри

Этот угол отстоит от наружных границ на размер рамки, который доступен через `clientLeft/clientTop`:

```
var coords3 = [coords.left + field.clientLeft, coords.top + field.clientTop];
```

Правый-нижний угол внутри

Этот угол отстоит от правой-нижней наружной границы на размер рамки. Так как нужная рамка находится справа-внизу, то специальных свойств для нее нет, но мы можем получить этот размер из CSS:

```
var coords4 = [
  coords.right - parseInt(getComputedStyle(field).borderRightWidth),
  coords.bottom - parseInt(getComputedStyle(field).borderBottomWidth)
]
```

Можно получить координаты элементом, присовив ему `clientLeft`, `clientWidth` и `clientTop`, `clientHeight` и присоединить к координатам левого-верхнего внутреннего угла. Получится то же самое, пожалуй даже быстрее и изящнее.

```
var coords4 = [  
    coords.left + elem.clientLeft + elem.clientWidth,  
    coords.top + elem.clientTop + elem.clientHeight  
]
```

[Полный код решения ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Разместить заметку рядом с элементом

[Открыть решение в песочнице ↗](#)

[К условию](#)

Координаты в документе

Область видимости для документа

- `top` — это `pageYOffset`.
- `bottom` — это `pageYOffset` плюс высота видимой части `documentElement.clientHeight`.
- `height` — полная высота документа, её вычисление давно в главе [Размеры и прокрутка страницы](#).

Итого:

```
function getDocumentScroll() {
  var scrollHeight = Math.max(
    document.body.scrollHeight, document.documentElement.scrollHeight,
    document.body.offsetHeight, document.documentElement.offsetHeight,
    document.body.clientHeight, document.documentElement.clientHeight
  );

  return {
    top: pageYOffset,
    bottom: pageYOffset + document.documentElement.clientHeight,
    height: scrollHeight
  };
}
```

[К условию](#)

Разместить заметку рядом с элементом (absolute)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Разместить заметку внутри элемента

[Открыть решение в песочнице ↗](#)

[К условию](#)

Введение в браузерные события

Спрятать при клике

[Решение задачи ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Спрятаться

Решение задачи заключается в использовании `this` в обработчике.

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

[К условию](#)

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 неё), что была назначена, а в коде передается такая же с виду функция, но, тем не менее, это другой объект.

Для того, чтобы удалить функцию-обработчик, нужно где-то сохранить ссылку на неё, например так:

```
function handler() {  
    alert( "1" );  
}  
  
button.addEventListener("click", handler);  
button.removeEventListener("click", handler);
```

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

[К условию](#)

Раскрывающееся меню

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

Получаем структуру:

```
<div class="menu">
  <span class="title">Сладости (нажми меня)!</span>
  <ul>
    <li>Пирог</li>
    <li>Пончик</li>
    <li>Мед</li>
  </ul>
</div>
```

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

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



[Сладости (нажми меня)!]

...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>
```



[Сладости (нажми меня)!]

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

Обычно меню будет закрыто:

```
.menu ul {
  margin: 0;
  list-style: none;
  padding-left: 20px;
  display: none;
}

.menu .title::before {
  content: '►';
  font-size: 80%;
  color: green;
}
```

Если же меню раскрыто, то есть имеет класс `.menu-open`, то стрелочка слева заголовка меняется и список детей показывается:

```
.menu.open .title::before {  
    content: '▼';  
}  
  
.menu.open ul {  
    display: block;  
}
```

Для JavaScript остался минимум работы — только добавить/удалить класс при клике.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Спрятать сообщение

- Изменим HTML/CSS, чтобы кнопка была в нужном месте сообщения. Кнопка — это тег `<button>`, поэтому понадобится несколько стилей.

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 частично оказаться «сверху» текста заголовка, перекрыв его конец. Чтобы этого не произошло, можно добавить `padding-right` к заголовку.

Если использовать `float: right`, то кнопка никогда не перекроет текст. Это, пожалуй хорошо.

С другой стороны, 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.

- Для того, чтобы получить кнопку из контейнера, используем `querySelectorAll`. На каждую кнопку повесим обработчик, который будет убирать родителя. Найти родителя можно через `parentNode`.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Карусель

Лента изображений в разметке должна быть представлена как список `` тегов ``.

Нужно расположить его внутри `<div>` фиксированного размера, так чтобы в один момент

`div (контейнер)`

`ul (width: 9999px)`

130x130

...

Чтобы список был длинный и элементы не переходили вниз, ему ставится `width: 9999px`, а элементам ``, соответственно, `float:left`, либо для элементов используется `display: inline-block`, как в этом решении.

Главное — не использовать `display:inline`, так как такие элементы имеют дополнительные отступы снизу для возможных «хвостов букв».

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

Для «прокрутки» будем сдвигать ``. Это можно делать по-разному, например, назначением CSS-свойства `transform: translateX()` или `margin-left:`

`div (контейнер)`

`ul (margin-left: -350px)`

130x130

...

У внешнего `<div>` фиксированная ширина, поэтому «лишние» изображения обрезаются.

Снаружи окошка находятся стрелки и внешний контейнер.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Объект события

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 мяч позиционирован абсолютно внутри поля, его координаты `left/top` нужно ставить относительно левого-верхнего внутреннего (внутри рамки!) угла поля.

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

```
var field = document.getElementById('field');
var ball = document.getElementById('ball');

field.onclick = function(e) {

    var fieldCoords = field.getBoundingClientRect();
    var fieldInnerCoords = {
        top: fieldCoords.top + field.clientTop,
        left: fieldCoords.left + field.clientLeft
    };

    ball.style.left = e.clientX - fieldInnerCoords.left + 'px';
    ball.style.top = e.clientY - fieldInnerCoords.top + 'px';

};
```

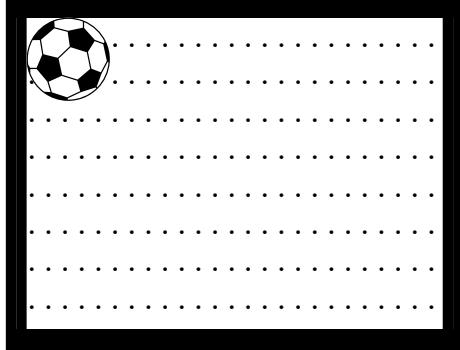
Далее мяч нужно сдвинуть на половину его ширины и высоты `ball.clientWidth/clientHeight`, чтобы он оказался центром под курсором.

Здесь есть важный «подводный камень» — размеры мяча в исходном документе не прописаны. Там просто стоит ``. Но на момент выполнения JavaScript картинка, возможно, ещё не загрузилась, так что высота и ширина мяча будут неизвестны (а они необходимы для центрирования).

Нужно добавить `width/height` в тег `` или задать размеры в CSS, тогда на момент выполнения JavaScript будет знать их и передвинет мяч правильно.

Код, который полностью центрирует мяч, вы найдете в полном решении:

Кликните на любое место поля, чтобы мяч перелетел туда.



[Открыть решение в песочнице ↗](#)

[К условию](#)

Делегирование событий

Скрытие сообщения с помощью делегирования

Поставьте обработчик `click` на контейнере. Он должен проверять, произошел ли клик на кнопке удаления (`target`), и если да, то удалять соответствующий ей DIV.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Раскрывающееся дерево

Схема решения

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

Использовать виджет, который будет показывать только то, что нужно, игнорировать.

```
<style>
  li {
    border: 1px solid green;
  }
</style>

<ul onclick="alert(event.target)">
  <li>Млекопитающие
    <ul>
      <li>Коровы</li>
      <li>Ослы</li>
      <li>Собаки</li>
      <li>Тигры</li>
    </ul>
  </li>
</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, получать по SPAN'у его родителя LI и ставить ему класс открыт/закрыт.

Напишите для этого JavaScript-код.

Оборачиваем заголовки в SPAN

Следующий код ищет все LI и оборачивает текстовые узлы в SPAN.

```
var treeUl = document.getElementsByTagName('ul')[0];

var treeLis = treeUl.getElementsByTagName('li');

for (var i = 0; i < treeLis.length; i++) {
  var li = treeLis[i];

  var span = document.createElement('span');
  li.insertBefore(span, li.firstChild); // добавить пустой SPAN
  span.appendChild(span.nextSibling); // переместить в него заголовок
}
```

Так выглядит дерево с обёрнутыми в SPAN заголовками и делегированием:

```
<style>
  span {
    border: 1px solid red;
  }
</style>

<ul onclick="alert(event.target.tagName)">
  <li><span>Млекопитающие</span>
    <ul>
      <li><span>Коровы</span></li>
      <li><span>Ослы</span></li>
      <li><span>Собаки</span></li>
      <li><span>Тигры</span></li>
    </ul>
  </li>
</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, скорее всего, будет сразу со SPAN: если HTML-код дерева генерируется на сервере, то это несложно, если дерево генерируется в JavaScript — тем более просто.

Итоговое решение

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

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.

```
var tree = document.getElementsByTagName('ul')[0];

tree.onclick = function(event) {
    var target = event.target;

    if (target.tagName != 'SPAN') {
        return; // клик был не на заголовке
    }

    var li = target.parentNode; // получить родительский LI

    // получить UL с потомками -- это первый UL внутри LI
    var childrenContainer = li.getElementsByTagName('ul')[0];

    if (!childrenContainer) return; // потомков нет -- ничего не надо делать

    // спрятать/показать (можно и через CSS-класс)
    childrenContainer.hidden = !childrenContainer.hidden;
}
```

Выделение узлов жирным при наведении делается при помощи CSS-селектора :hover.

[Открыть решение в песочнице](#)

[К условию](#)

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

[Открыть решение в песочнице](#)

[К условию](#)

Поведение «подсказка»

[Открыть решение в песочнице ↗](#)

[К условию](#)

Действия браузера по умолчанию

Почему не работает `return false`?

Дело в том, что обработчик из атрибута `onclick` делается браузером как функция с заданным телом.

То есть, в данном случае он будет таким:

```
function(event) {  
    handler() // тело взято из атрибута onclick  
}
```

При этом возвращаемое `handler` значение никак не используется и не влияет на результат.

Рабочий вариант:

```
<script>  
    function handler() {  
        alert("...");  
        return false;  
    }  
</script>  
w3.org</a>
```

Также можно использовать объект события для вызова `event.preventDefault()`, например:

```
<script>
  function handler(event) {
    alert("...");
    event.preventDefault();
  }
</script>

<a href="http://w3.org" onclick="handler(event)">w3.org</a>
```

[К условию](#)

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 упремся в контейнер.

```
contents.onclick = function(evt) {
  var target = evt.target;

  function handleLink(href) {
    var isLeaving = confirm('Уйти на ' + href + '?');
    if (!isLeaving) return false;
  }

  while (target != this) {
    if (target.nodeName == 'A') {
      return handleLink(target.getAttribute('href')) // (*)
    }
    target = target.parentNode;
  }
};
```

В строке (*) используется атрибут, а не свойство `href`, чтобы показать в `confirm` именно то, что написано в HTML-атрибуте, так как свойство может отличаться, оно обязано содержать полный валидный адрес.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Галерея изображений

Решение состоит в том, чтобы добавить обработчик на контейнер `#thumbs` и отслеживать

Когда происходит событие, обработчик должен изменять `src` #largeImg на `href` ссылки и заменять `alt` на ее `title`.

Код решения:

```
var largeImg = document.getElementById('largeImg');

document.getElementById('thumbs').onclick = function(e) {
    var target = e.target;

    while (target != this) {

        if (target.nodeName == 'A') {
            showThumbnail(target.href, target.title);
            return false;
        }

        target = target.parentNode;
    }
}

function showThumbnail(href, title) {
    largeImg.src = href;
    largeImg.alt = title;
}
```

Предзагрузка картинок

Для того, чтобы картинка загрузилась, достаточно создать новый элемент IMG и указать ему `src`, вот так:

```
var imgs = thumbs.getElementsByTagName('img');
for (var i = 0; i < imgs.length; i++) {
    var url = imgs[i].parentNode.href;

    var img = document.createElement('img');
    img.src = url;
}
```

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 закеширована.

Семантическая верстка

Для списка картинок используется DIV. С точки зрения семантики более верный вариант — список UL/LI.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Список с выделением

[Открыть решение в песочнице ↗](#)

[К условию](#)

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)` ↗.

...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) заворачиваем текстовый узел в SPAN
// 2) проверяем
var elem = document.elementFromPoint(e.clientX, e.clientY);
var isClickOnTitle = (elem == span);

// 3) возвращаем текстовый узел обратно из SPAN
```

На шаге 3 текстовый узел вынимается обратно из SPAN, всё возвращается в исходное состояние.

Решение

[Открыть решение в песочнице ↗](#)

[К условию](#)

Поведение «вложенная подсказка»

[Открыть решение в песочнице ↗](#)

[К условию](#)

Подсказка при замедлении над элементом

Будем замерять скорость движения курсора.

Для этого можно запустить `setInterval`, который каждые 100мс (или другой интервал) будет сравнивать текущие координаты курсора с предыдущими и, если расстояние пройдено маленькое, считаем, что посетитель «навёл указатель на элемент», вызвать `options.over`.

В браузере нет способа «просто получить» текущие координаты. Это может сделать обработчик события, в данном случае `mousemove`. Поэтому нужно будет поставить обработчик на `mousemove` и при каждом движении запоминать текущие координаты, чтобы `setInterval` мог раз в 100мс сравнивать их.

Можно обойтись и без `setInterval` — сравнивать координаты при каждом срабатывании `mousemove`. Если передвинулись на маленькое расстояние с последнего `mousemove` — это «наведение на элемент», а на большое — игнорируем. Вариант с `setInterval` лучше с точки зрения производительности — `mousemove` происходит уж очень часто, но если проверка несложная, то и `mousemove` подойдёт.

Имеет смысл начинать анализ координат и отслеживание `mousemove` при заходе на элемент, а заканчивать — при выходе с него.

Чтобы точно отловить момент входа и выхода, без учёта подэлементов (во избежание мигания), можно использовать `mouseenter/mouseleave`.

В решении, предложенном ниже, однако, используется `mouseover/mouseout`, так как это позволит легко «прикрутить» к такому объекту делегирование, если потребуется. А, чтобы не было лишних срабатываний, лишние переходы отфильтровываются.

При этом при обнаружении «наведения на элемент» это запоминается в переменной `isHover` и вызывается `options.over`, а затем, при выходе с элемента, если было «наведение», вызывается `options.out`.

[Открыть решение в песочнице ↗](#)

Мышь: Drag'n'Drop

Слайдер

Как можно видеть из HTML/CSS, слайдер — это DIV, подкрашенный фоном/градиентом, внутри которого находится другой DIV, оформленный как бегунок, с `position: relative`.

Бегунок немного поднят, и вылезает по высоте из родителя.

На этой основе мы реализуем горизонтальный Drag'n'Drop, ограниченный по ширине. Его особенность — в `position: relative` у переносимого элемента, т.е. координата ставится не абсолютная, а относительно родителя.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Расставить супергероев по полю

В решении этой задачи для переноса мы используем координаты относительно окна и `position: fixed`. Так проще.

А по окончании — прибавляем прокрутку и делаем `position: absolute`, чтобы элемент был привязан к определённому месту в документе, а не в окне. Можно было и сразу `position: absolute` и оперировать в абсолютных координатах, но код был бы немного длиннее.

Детали решения расписаны в комментариях в исходном коде.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Мышь: колёсико, событие wheel

Масштабирование колёсиком мыши

Решение использует прокрутку с помощью свойства transform, а не style.transform.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Прокрутка без влияния на страницу

[Открыть решение в песочнице ↗](#)

[К условию](#)

Прокрутка: событие scroll

Аватар наверху при прокрутке

[Открыть решение в песочнице ↗](#)

[К условию](#)

Кнопка вверх-вниз

Добавим в документ DIV с кнопкой:

```
<div id="updown"></div>
```

Сама кнопка должна иметь position:fixed.

```
#updown {  
    position: fixed;  
    top: 30px;  
    left: 10px;  
    cursor: pointer;  
}
```

Кнопка является CSS-спрайтом, поэтому мы дополнительно добавляем ей размер и два состояния:

```

#updown {
  height: 9px;
  width: 14px;
  position: fixed;
  top: 30px;
  left: 10px;
  cursor: pointer;
}

#updown.up {
  background: url(...updown.gif) left top;
}

#updown.down {
  background: url(...updown.gif) left -9px;
}

```

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:

```

window.onscroll = function() {
  var pageY = window.pageYOffset || document.documentElement.scrollTop;
  var innerHeight = document.documentElement.clientHeight;

  switch (updownElem.className) {
    case '':
      if (pageY > innerHeight) {
        updownElem.className = 'up';
      }
      break;

    case 'up':
      if (pageY < innerHeight) {
        updownElem.className = '';
      }
      break;

    case 'down':
      if (pageY > innerHeight) {
        updownElem.className = 'up';
      }
      break;
  }
}

```

При клике:

```
var pageYLabel = 0;

updownElem.onclick = function() {
    var pageY = window.pageYOffset || document.documentElement.scrollTop;

    switch (this.className) {
        case 'up':
            pageYLabel = pageY;
            window.scrollTo(0, 0);
            this.className = 'down';
            break;

        case 'down':
            window.scrollTo(0, pageYLabel);
            this.className = 'up';
    }
}
```

[Открыть решение в песочнице](#)

[К условию](#)

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:

```
...страница...

function isVisible(elem) {
    var coords = elem.getBoundingClientRect();
    var windowHeight = document.documentElement.clientHeight;

    // верхняя граница elem в пределах видимости ИЛИ нижняя граница видима
    var topVisible = coords.top > 0 && coords.top < windowHeight;
    var bottomVisible = coords.bottom < windowHeight && coords.bottom > 0;

    return topVisible || bottomVisible;
}

showVisible();
window.onscroll = 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.

видимой области и сравнивает их с элементом.

Для видимости достаточно, чтобы координаты верхней(или нижней) границы элемента находились между границами видимой области.

В решении также указан вариант с `isVisible`, который расширяет область видимости на ± 1 страницу (высота страницы — `document.documentElement.clientHeight`).

[Открыть полное решение в песочнице ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Клавиатура: keyup, keydown, keypress

Поле только для цифр

Подсказка: выбор события

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` / `Cmd`.

Итак, вот решение:

```
input.onkeypress = function(e) {
  e = e || event;

  if (e.ctrlKey || e.altKey || e.metaKey) return;

  var chr = getChar(e);

  // с null надо осторожно в неравенствах,
  // т.к. например null >= '0' => true
  // на всякий случай лучше вынести проверку chr == null отдельно
  if (chr == null) return;

  if (chr < '0' || chr > '9') {
    return false;
  }
}
```

[Открыть полное решение в песочнице ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

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

[Открыть решение в песочнице ↗](#)

[К условию](#)

Загрузка скриптов, картинок, фреймов: `.onload` и `onerror`

Красивый «ALT»

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

[Открыть решение в песочнице ↗](#)

[К условию](#)

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

[Открыть решение в песочнице ↗](#)

[К условию](#)

Скрипт с коллбэком

Подсказка

Добавляйте SCRIPT при помощи методов DOM:

```
var script = document.createElement('script');
script.src = src;

// в документе может не быть HEAD или BODY,
// но хотя бы один (текущий) SCRIPT в документе есть
var s = document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(script, s); // перед ним и вставим
```

На скрипт повесьте обработчики `onload/onreadystatechange`.

[Код](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Скрипты с коллбэком

Подсказки

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

[Открыть решение в песочнице ↗](#)

[К условию](#)

Навигация и свойства элементов формы

Добавьте опцию к селекту

Решение:

```
<select>
  <option value="Rock">Рок</option>
  <option value="Blues" selected>Блюз</option>
</select>

<script>
  var select = document.body.children[0];

  // 1)
  var selectedOption = select.options[select.selectedIndex];
  alert( selectedOption.value );

  // 2)
  var newOption = new Option("Classic", "Классика");
  select.appendChild(newOption);

  // 3)
  newOption.selected = true;
</script>
```

Фокусировка: focus/blur

Улучшенный плейсхолдер

В данном случае достаточно событий `input.focus`/`input.blur`.

Если бы мы хотели реализовать это на уровне документа, то применили бы делегирование и события `focusin`/`focusout` (эмуляцию для firefox), так как обычные `focus`/`blur` не всплывают.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Мышонок на «клавиатурном» приводе

Нам нужно ловить `onclick` на мышонке и в `onkeydown` на нём смотреть коды символов. При скан-кодах стрелок двигать мышонка через `position:absolute` или `position:fixed`.

Скан-коды для клавиш стрелок можно узнать, нажимая на них на [тестовом стенде](#). Вот они: 37-38-39-40 (влево-вверх-вправо-вниз).

Проблема может возникнуть одна — `keydown` не возникает на элементе, если на нём нет фокуса.

Чтобы фокус был — нужно добавить мышонку атрибут `tabindex` через JS или в HTML.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Горячие клавиши

[CSS для решения](#)

Как видно из исходного кода, `#view` — это `<div>`, который будет содержать результат, а `#area` — это редактируемое текстовое поле.

Так как мы преобразуем `<div>` в `<textarea>` и обратно, нам нужно сделать их практически одинаковыми с виду:

```
#view,  
#area {  
    height: 150px;  
    width: 400px;  
    font-family: arial;  
    font-size: 14px;  
}
```

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):

```
#view {  
    /* padding + border = 3px */  
  
    padding: 2px;  
    border: 1px solid black;  
}
```

CSS для `#area` заменяет поля границами:

```
#area {  
    border: 3px groove blue;  
    padding: 0px;  
    display: none;  
}
```

По умолчанию, текстовое поле скрыто. Кстати, этот код убирает дополнительную рамку в ряде браузеров, которая появляется вокруг поля, когда на него попадает фокус:

```
#area:focus {  
    outline: none; /* убирает рамку при фокусе */  
}
```

Горячие клавиши

Чтобы отследить горячие 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`:

```
document.onkeydown = function(e) {
  if (e.keyCode == 27) { // escape
    cancel();
    return false;
  }

  if ((e.ctrlKey && e.keyCode == 'E'.charCodeAt(0)) && !area.offsetHeight) {
    edit();
    return false;
  }

  if ((e.ctrlKey && e.keyCode == 'S'.charCodeAt(0)) && area.offsetHeight) {
    save();
    return false;
  }
};
```

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, спрятанным с помощью стилей, а так же для элементов, у которых скрыты родители.

Редактирование

Следующие функции переключают режимы. HTML-код разрешен, поэтому возможна прямая трансформация в `<textarea>` и обратно.

```
function edit() {
  view.style.display = 'none';
  area.value = view.innerHTML;
  area.style.display = 'block';
  area.focus();
}

function save() {
  area.style.display = 'none';
  view.innerHTML = area.value;
  view.style.display = 'block';
}

function cancel() {
  area.style.display = 'none';
  view.style.display = 'block';
}
```

[Открыть решение в песочнице ↗](#)

[К условию](#)

Редактирование TD по клику

1. При клике — заменяем `innerHTML` ячейки на `<textarea>` с размерами «под ячейку», без

- 2. В `textarea.value` присваиваем содержимое ячейки.
- 3. Фокусируем посетителя на ячейке вызовом `focus()`.
- 4. Показываем кнопки OK/CANCEL под ячейкой.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Красивый плейсхолдер для INPUT

Вёрстка

Для вёрстки можно использовать отрицательный `margin` у текста с подсказкой.

Решение в плане вёрстки есть в решении задачи [Расположить текст внутри INPUT](#).

Решение

```
placeholder.onclick = function() {
  input.focus();
}

// onfocus сработает и вызове input.focus() и при клике на input
input.onfocus = function() {
  if (placeholder.parentNode) {
    placeholder.parentNode.removeChild(placeholder);
  }
}
```

[Открыть полный код решения ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Поле, предупреждающее о включенном CapsLock

Алгоритм

JavaScript не имеет доступа к текущему состоянию `CapsLock`. При загрузке страницы не известно, включён он или нет.

Но мы можем догадаться о его состоянии из событий:

`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 `CapsLock`.

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.

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 доплынет 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 состояние `Shift`:

```
document.onkeypress = function(e) {  
    var chr = getChar(e);  
    if (!chr) return; // специальная клавиша  
  
    if (chr.toLowerCase() == chr.toUpperCase()) {  
        // символ, который не имеет регистра, такой как пробел,  
        // мы не можем использовать для определения состояния CapsLock  
        return;  
    }  
  
    capsLockEnabled = (chr.toLowerCase() == chr && e.shiftKey) || (chr.toUpperCase() == chr && !e.shiftKey);  
}
```

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.

```
if (navigator.platform.substr(0, 3) != 'Mac') { // событие для CapsLock глючит под Mac
  document.onkeydown = function(e) {
    if (e.keyCode == 20 && capsLockEnabled !== null) {
      capsLockEnabled = !capsLockEnabled;
    }
  };
}
```

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:

```
<input type="text" onkeyup="checkCapsWarning(event)" onfocus="checkCapsWarning(event)" onblur="removeCapsWarning()>

<div style="display:none;color:red" id="caps">Внимание: нажат CapsLock!</div>

<script>
  function checkCapsWarning() {
    document.getElementById('caps').style.display = capsLockEnabled ? 'block' : 'none';
  }

  function removeCapsWarning() {
    document.getElementById('caps').style.display = 'none';
  }
</script>
```

[Полный код решения ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Автовычисление процентов по вкладу

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 IE8-.
- На `select`: событие `change`.

[Открыть решение в песочнице ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Формы: отправка, событие и метод submit

Модальное диалоговое окно

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:

```
#cover-div {  
    position: fixed;  
    top: 0;  
    left: 0;  
    z-index: 9000;  
    width: 100%;  
    height: 100%;  
    background-color: gray;  
    opacity: 0.3;  
}
```

Самой форме можно дать еще больший z-index, чтобы она была над DIV'ом. Мы не

[Открыть решение в песочнице ↗](#)

[К условию](#)

Валидация формы

[Открыть решение в песочнице ↗](#)

[Открыть решение в песочнице ↗](#)

[К условию](#)

Графические компоненты

Часики

[Открыть решение в песочнице ↗](#)

[К условию](#)

Слайдер-компонент

[Открыть решение в песочнице ↗](#)

[К условию](#)

Компонент: список с выделением

[Открыть решение в песочнице ↗](#)

[К условию](#)

Голосовалка

[Открыть решение в песочнице ↗](#)

[К условию](#)

Голосовалка в прототипном стиле ООП

[Открыть решение в песочнице ↗](#)

[К условию](#)

Добавить двойной голос в голосовалку

[Открыть решение в песочнице ↗](#)

[К условию](#)

Вёрстка графических компонентов

Семантическое меню

Несмотря на то, что меню более-менее прилично отображается, эта вёрстка совершенно не семантична.

Ошибки:

1. Во-первых, меню представляет собой *список элементов*, а для списка существует тег `LI`.

Семантический подход — это когда теги используются по назначению. Для элементов списка ``, для адреса `<address>`, для заголовка таблицы `<th>` и т.п.

2. Во-вторых, класс `rounded-horizontal-blocks` показывает, что содержимое должно быть оформлено как скругленные горизонтальные блоки. Любой класс, отражающий оформление, несемантичен.

говорить о том, что смысл элемента — «меню».

3. В-третьих, элемент `.vertical-splitter`. Здесь класс вполне семантичен, этот элемент списка является вертикальным разделителем, так что здесь всё в порядке. Но на этот раз несемантичность — в содержимом.

Мы, по возможности, стараемся, чтобы HTML содержал именно информацию, а символ вертикальной черты | выполняет чисто оформительскую функцию.

Поэтому от него следует либо вообще избавиться, либо переместить в CSS при помощи `::before`.

И, наконец, это не обязательно и не ошибка, но обычно элементы, которые являются ссылками или кнопками, оформляют в `<a>` или `<button>`.

Вариант ниже — семантичен:

```
<ul class="menu">
  <li class="menu__item"><a href="#">Главная</a></li>
  <li class="menu__vertical-splitter"></li>
  <li class="menu__item"><a href="#">Товары</a></li>
  <li class="menu__item"><a href="#">Фотографии</a></li>
  <li class="menu__item"><a href="#">Контакты</a></li>
</ul>
```

Дополнительно, классы помечены префиксом компонента, на тот случай, если в заголовках появится произвольный HTML.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Шаблонизатор LoDash

Шаблон для таблицы с пользователями

[Открыть решение в песочнице ↗](#)

[К условию](#)

Шаблон в div с display:none?

доставить незакрытым тег — и могут быть проблемы. А в скрипте может быть почти что угодно, его содержимое полностью игнорируется.

Кроме того, содержимое DIV браузер обрабатывает, создаёт DOM-узлы, добавляет их в документ, но там они совсем не нужны, это же шаблон.

Альтернативный вариант — поместить шаблон в комментарий `<!-- ... -->`, его содержимое можно получить при помощи `node.data`. Но у узла-комментария не может быть `id`, так что это менее удобно.

[К условию](#)

Сделайте меню ссылками

В решении обратим внимание:

- Чтобы ссылка `href` была корректной, даже если в ключах `items` попались русские символы и пробелы — используется функция [encodeURIComponent ↗](#).
- Для вывода `href` при клике на ссылку используется делегирование. Причём обработчик не сам выводит `href`, а лишь разбирается в произошедшем и вызывает функцию `select`, которая представляет действие «выбора» элемента меню. В последующих примерах эта функция станет сложнее.

[Открыть решение в песочнице ↗](#)

[К условию](#)

Коллбэки и события на компонентах

Голосовалка «на событиях»

[Открыть решение в песочнице ↗](#)

[К условию](#)

Список с выделением и событием

[К условию](#)

Свой селект

В этом решении для закрытия селекта по клику вне него используется отслеживание произвольных кликов вне документа.

Альтернатива — события `focusin/focusout`, т.е. считаем, что пока фокус в селекте — он открыт. С одной стороны, это более мощный способ, он позволяет перемещаться по элементам управления при помощи `Tab` и корректно обрабатывать уход при помощи клавиатуры.

С другой стороны, это решение не универсально: если выводится `alert`, то фокус «прыгает» в него, уходя с элемента, а потом возвращается обратно. При этом JavaScript зарегистрирует уход фокуса `focusout` и возвращение `focusin`, хотя по смыслу фокус с элемента никуда не уходил, просто был `alert`.

Побочный эффект — к закрытию и раскрытию (лишнему) элемента управления при таких «ненамеренных» потерях фокуса. Поэтому был выбран `onclick`.

Решение:

[Открыть решение в песочнице ↗](#)

[К условию](#)

Слайдер с событиями

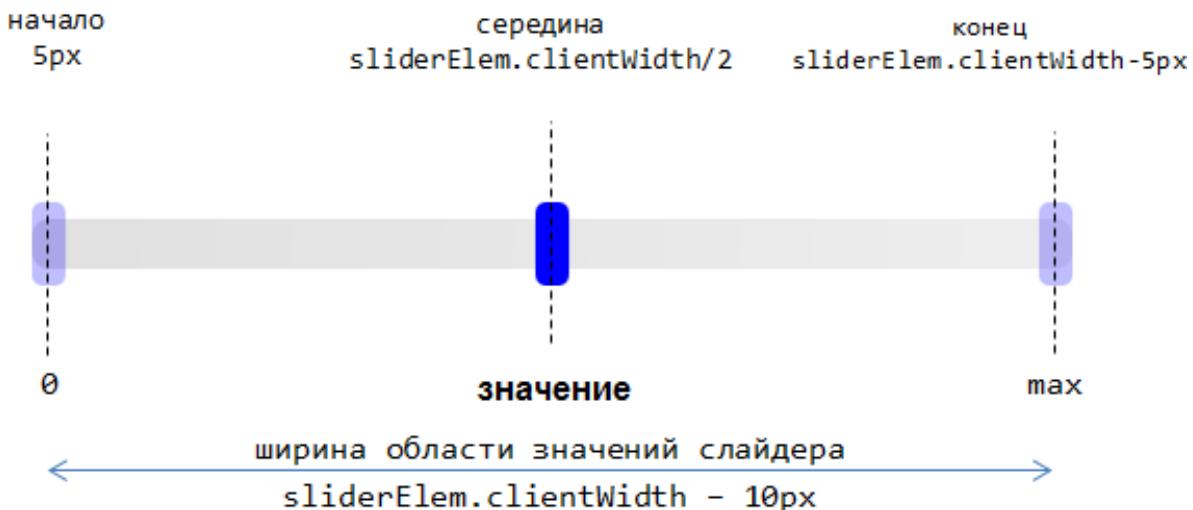
Для решения этой задачи достаточно создать две функции: `valueToPosition` будет получать по значению положение бегунка, а `positionToValue` — наоборот, транслировать текущую координату бегунка в значение.

Как сопоставить позицию слайдера и значение?

Для этого посмотрим крайние значения слайдера. Допустим, размер бегунка 10px.

Раз центр соответствует значению, то крайнее левое значение будет соответствовать центру на 5px, а крайнее правое — центру на 5px от правой границы:

координаты



Соответственно, ширина области изменения будет `sliderElem.clientWidth - thumbElem.clientWidth`. Далее её можно уже поделить на части, количество пикселей на значение будет:

```
pixelsPerValue = (sliderElem.clientWidth - thumbElem.clientWidth) / max;
```

Может получиться так, что это значение будет дробным, меньше единицы. Например, если `max = 1000`, а ширина слайдера 110 (пробег 100), то будет 0.1 пикселя на значение.

Используя `pixelsPerValue` мы сможем переводить позицию бегунка в значение и обратно.

Крайнее левое значение `thumbElem.style.left` равно нулю, крайнее правое — как раз ширине доступной области `sliderElem.clientWidth - thumbElem.clientWidth`. Поэтому можно получить значение слайдера, поделив его на `pixelsPerValue`:

```
function positionToValue(left) {
  return Math.round(left / pixelsPerValue);
}

function valueToPosition(value) {
  return pixelsPerValue * value;
}
```

[Открыть решение в песочнице ↗](#)

[К условию](#)