

Часть 2

Браузер: документ, события, интерфейсы

Js

Илья Кантор

Сборка от 2 апреля 2023 г.

Последняя версия учебника находится на сайте <https://learn.javascript.ru>.

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

- [Документ](#)
 - Браузерное окружение, спецификации
 - DOM-дерево
 - Навигация по DOM-элементам
 - Поиск: getElement*, querySelector*
 - Свойства узлов: тип, тег и содержимое
 - Атрибуты и свойства
 - Изменение документа
 - Стили и классы
 - Размеры и прокрутка элементов
 - Размеры и прокрутка окна
 - Координаты
- [Введение в события](#)
 - Введение в браузерные события
 - Всплытие и погружение
 - Делегирование событий
 - Действия браузера по умолчанию
 - Генерация пользовательских событий
- [Интерфейсные события](#)
 - Основы событий мыши
 - Движение мыши: mouseover/out, mouseenter/leave
 - Drag'n'Drop с событиями мыши
 - Клавиатура: keydown и keyup
 - События указателя
 - Прокрутка
- [Формы, элементы управления](#)
 - Свойства и методы формы
 - Фокусировка: focus/blur
 - События: change, input, cut, copy, paste
 - Отправка формы: событие и метод submit
- [Загрузка документа и ресурсов](#)
 - Страница:DOMContentLoaded, load, beforeunload, unload
 - Скрипты: async, defer

- Загрузка ресурсов: onload и onerror
- Разное
 - MutationObserver: наблюдатель за изменениями
 - Selection и Range
 - Событийный цикл: микрозадачи и макрозадачи

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

Документ

Здесь мы научимся изменять страницу при помощи JavaScript.

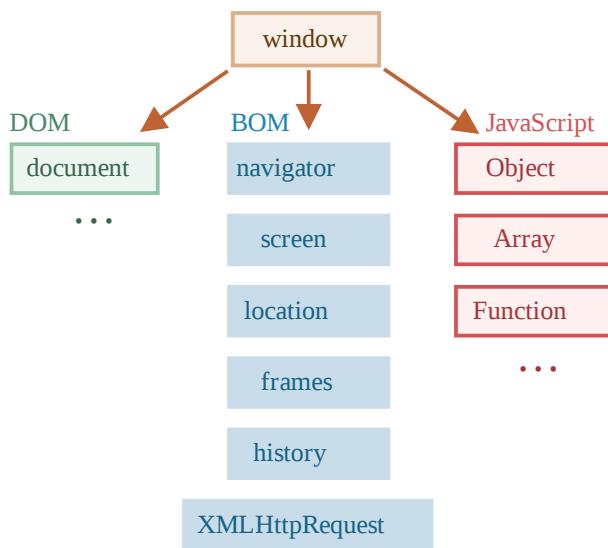
Браузерное окружение, спецификации

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач.

Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свою функциональность, которую спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее.

На картинке ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как мы видим, имеется корневой объект `window`, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода, об этом более подробно говорится в главе [Глобальный объект](#).
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь мы используем `window` как глобальный объект:

```
function sayHi() {
  alert("Hello");
}

// глобальные функции доступны как методы глобального объекта:
window.sayHi();
```

А здесь мы используем `window` как объект окна браузера, чтобы узнать его высоту:

```
alert(window.innerHeight); // внутренняя высота окна браузера
```

Существует гораздо больше свойств и методов для управления окном браузера. Мы рассмотрим их позднее.

DOM (Document Object Model)

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект `document` – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

Например:

```
// заменим цвет фона на красный,
document.body.style.background = "red";

// а через секунду вернём как было
setTimeout(() => document.body.style.background = "", 1000);
```

Мы использовали в примере только `document.body.style`, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации:

- **DOM Living Standard** на <https://dom.spec.whatwg.org> ↗

i DOM – не только для браузеров

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существуют и другие, отличные от браузеров, инструменты, использующие DOM.

Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

i CSSOM для стилей

Правила стилей CSS структурированы иначе чем HTML. Для них есть отдельная спецификация [CSSOM](#), которая объясняет, как стили должны представляться в виде объектов, как их читать и писать.

CSSOM используется вместе с DOM при изменении стилей документа. В реальности CSSOM требуется редко, обычно правила CSS статичны. Мы редко добавляем/удаляем стили из JavaScript, но и это возможно.

BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

- Объект [navigator](#) даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются:
`navigator.userAgent` – информация о текущем браузере, и
`navigator.platform` – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
- Объект [location](#) позволяет получить текущий URL и перенаправить браузер по новому адресу.

Вот как мы можем использовать объект `location`:

```
alert(location.href); // показывает текущий URL
if (confirm("Перейти на Wikipedia?")) {
    location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL
}
```

Функции `alert/confirm/prompt` тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта

окна браузера для коммуникации с пользователем.

1 Спецификации

ВОМ является частью общей [спецификации HTML](#).

Да, вы всё верно услышали. Спецификация HTML по адресу <https://html.spec.whatwg.org> не только про «язык HTML» (теги, атрибуты), она также покрывает целое множество объектов, методов и специфичных для каждого браузера расширений DOM. Это всё «HTML в широком смысле». Для некоторых вещей есть отдельные спецификации, перечисленные на <https://spec.whatwg.org>.

Итого

Говоря о стандартах, у нас есть:

Спецификация DOM

описывает структуру документа, манипуляции с контентом и события, подробнее на <https://dom.spec.whatwg.org>.

Спецификация CSSOM

Описывает файлы стилей, правила написания стилей и манипуляций с ними, а также то, как это всё связано со страницей, подробнее на <https://www.w3.org/TR/cssom-1/>.

Спецификация HTML

Описывает язык HTML (например, теги) и ВОМ (объектную модель браузера) – разные функции браузера: `setTimeout`, `alert`, `location` и так далее, подробнее на <https://html.spec.whatwg.org>. Тут берётся за основу спецификация DOM и расширяется дополнительными свойствами и методами.

Кроме того, некоторые классы описаны отдельно на <https://spec.whatwg.org>.

Пожалуйста, заметьте для себя эти ссылки, так как по ним содержится очень много информации, которую невозможно изучить полностью и держать в уме.

Когда вам нужно будет прочитать о каком-то свойстве или методе, справочник на сайте Mozilla <https://developer.mozilla.org/ru/search> тоже очень хороший ресурс, хотя ничто не сравнится с чтением спецификации: она сложная и объёмная, но сделает ваши знания максимально полными.

Для поиска чего-либо обычно удобно использовать интернет-поиск со словами «WHATWG [термин]» или «MDN [термин]», например <https://google.com?q=whatwg+localStorage>, <https://google.com?q=mdn+localStorage>.

А теперь давайте перейдём к изучению DOM, так как страница – это основа всего.

DOM-дерево

Основой HTML-документа являются теги.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

Например, `document.body` – объект для тега `<body>`.

Если запустить этот код, то `<body>` станет красным на 3 секунды:

```
document.body.style.background = 'red'; // сделать фон красным  
setTimeout(() => document.body.style.background = '', 3000); // вернуть назад
```

Это был лишь небольшой пример того, что может DOM. Скоро мы изучим много способов работать с DOM, но сначала нужно познакомиться с его структурой.

Пример DOM

Начнём с такого, простого, документа:

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

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:



Каждый узел этого дерева – это объект.

Теги являются *узлами-элементами* (или просто элементами). Они образуют структуру дерева: `<html>` – это корневой узел, `<head>` и `<body>` его дочерние узлы и т.д.

Текст внутри элементов образует *текстовые узлы*, обозначенные как `#text`. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Например, в теге `<title>` есть текстовый узел `"О лосях"`.

Обратите внимание на специальные символы в текстовых узлах:

- перевод строки: `\n` (в JavaScript он обозначается как `\n`)
- пробел:

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге `<head>` есть несколько пробелов перед `<title>`, которые образуют текстовый узел `#text` (он содержит в себе только перенос строки и несколько пробелов).

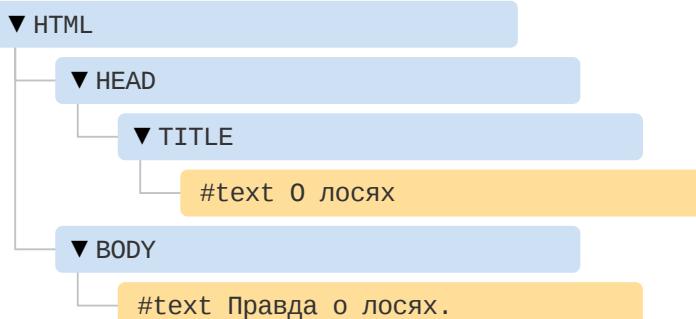
Существует всего два исключения из этого правила:

- По историческим причинам пробелы и перевод строки перед тегом `<head>` игнорируются
- Если мы записываем что-либо после закрывающего тега `</body>`, браузер автоматически перемещает эту запись в конец `body`, поскольку спецификация HTML требует, чтобы всё содержимое было внутри `<body>`. Поэтому после закрывающего тега `</body>` не может быть никаких пробелов.

В остальных случаях всё просто – если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если мы их удалим, то в DOM их тоже не будет.

Здесь пробельных текстовых узлов нет:

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



➊ Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки

Когда мы работаем с деревом DOM, используя инструменты разработчика в браузере (которые мы рассмотрим позже), пробелы в начале/конце текста и пустые текстовые узлы (переносы строк) между тегами обычно не отображаются.

Таким образом инструменты разработки экономят место на экране.

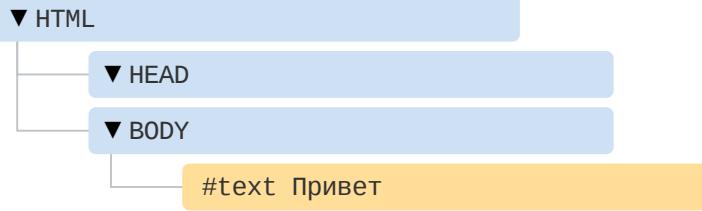
В дальнейших иллюстрациях DOM мы также будем для краткости пропускать пробельные текстовые узлы там, где они не имеют значения. Обычно они не влияют на то, как отображается документ.

Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM.

Например, в начале документа всегда должен быть тег `<html>`. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега `<body>`.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги `<html>` и `<body>`, добавит необходимый тег `<head>`, и DOM будет выглядеть так:

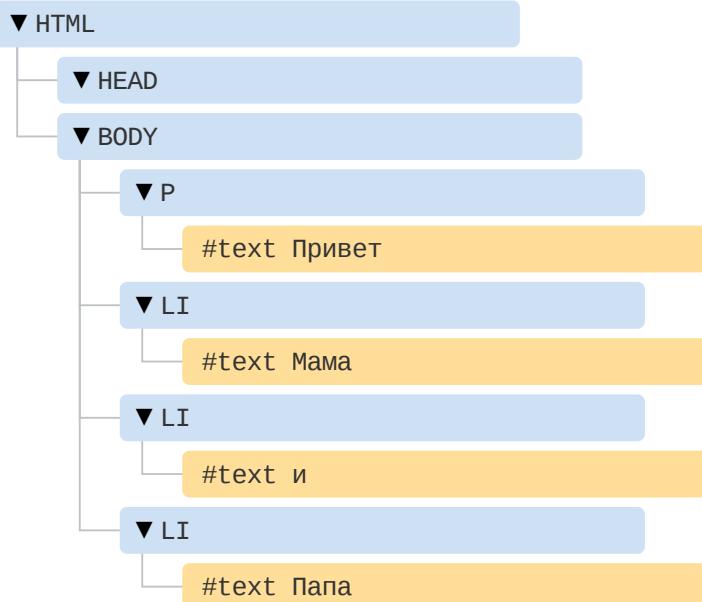


При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

Есть такой документ с незакрытыми тегами:

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

...Но DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали:



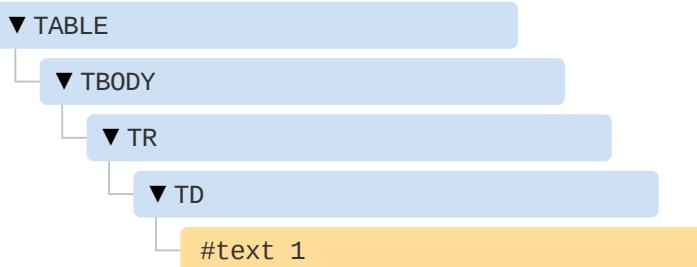
Таблицы всегда содержат `<tbody>`

Важный «косой случай» – работа с таблицами. По стандарту DOM у них должен быть `<tbody>`, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет `<tbody>` в DOM самостоятельно.

Для такого HTML:

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

DOM-структура будет такой:



Видите? Из пустоты появился `<tbody>`, как будто документ и был таким. Важно знать об этом, иначе при работе с таблицами возможны сюрпризы.

Другие типы узлов

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов.

Например, узел-комментарий:

```
<!DOCTYPE HTML>
<html>
<body>
    Правда о лосях.
    <ol>
        <li>Лось -- животное хитрое</li>
        <!-- Kommentarий -->
        <li>...и коварное!</li>
    </ol>
</body>
</html>
```



Здесь мы видим узел нового типа – **комментарий**, обозначенный как `#comment`, между двумя текстовыми узлами.

Казалось бы – зачем комментарий в DOM? Он никак не влияет на визуальное отображение. Но есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве.

Все, что есть в HTML, даже комментарии, является частью DOM.

Даже директива `<!DOCTYPE . . . >`, которую мы ставим в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед `<html>`. Мы не будем рассматривать этот узел, мы даже не рисуем его на наших диаграммах, но он существует.

Даже объект `document`, представляющий весь документ, формально является DOM-узлом.

Существует [12 типов узлов](#). Но на практике мы в основном работаем с 4 из них:

1. `document` – «входная точка» в DOM.
2. узлы-элементы – HTML-теги, основные строительные блоки.
3. текстовые узлы – содержат текст.
4. комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

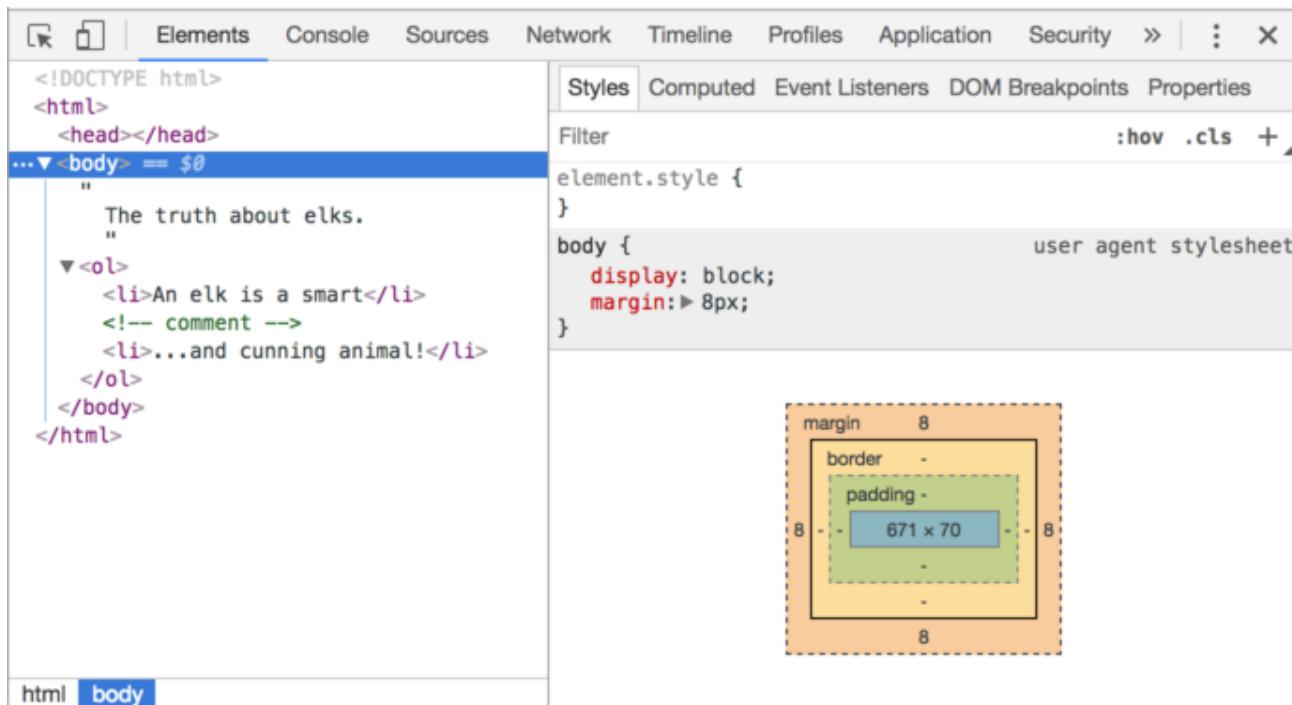
Поэкспериментируйте сами

Чтобы посмотреть структуру DOM в реальном времени, попробуйте [Live DOM Viewer](#). Просто введите что-нибудь в поле, и ниже вы увидите, как меняется DOM.

Другой способ исследовать DOM – это использовать инструменты разработчика браузера. Это то, что мы каждый день делаем при разработке.

Для этого откройте страницу [elks.html](#), включите инструменты разработчика и перейдите на вкладку Elements.

Выглядит примерно так:

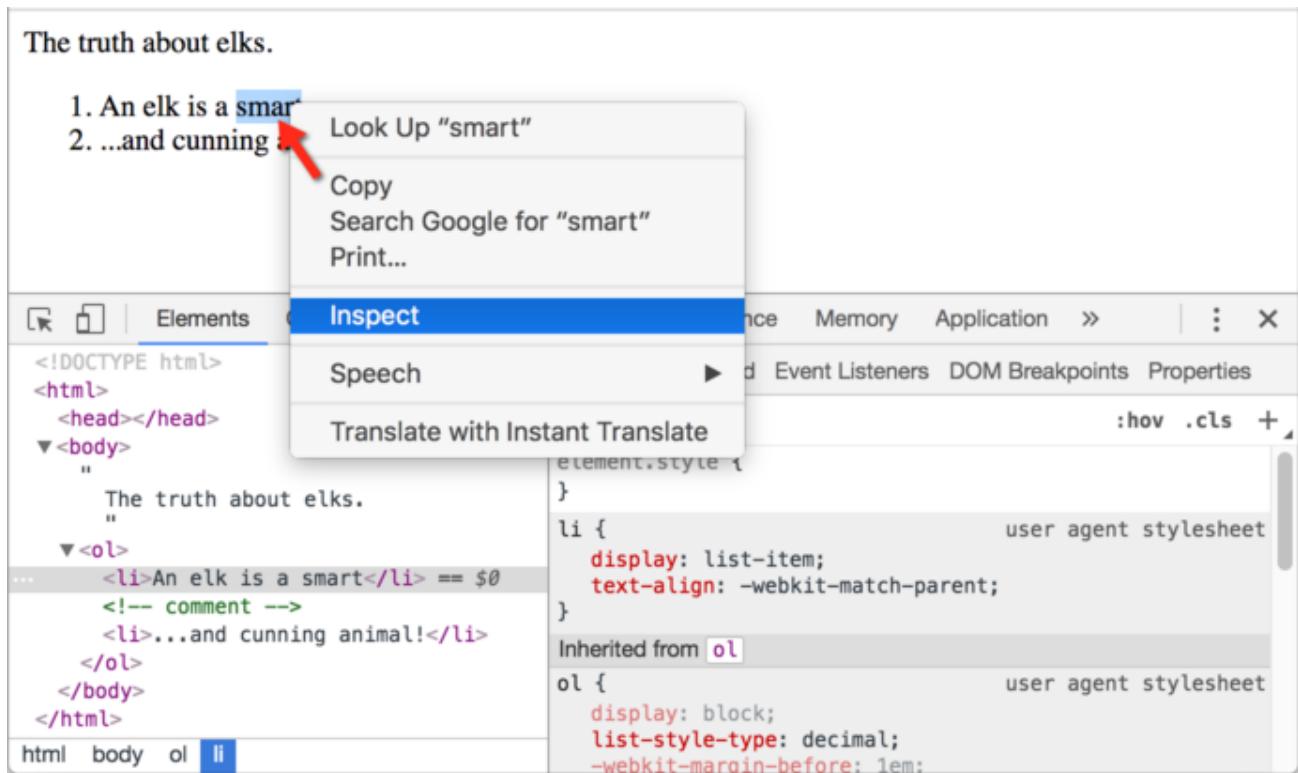


Вы можете увидеть DOM, понажимать на элементы, детально рассмотреть их и так далее.

Обратите внимание, что структура DOM в инструментах разработчика отображается в упрощённом виде. Текстовые узлы показаны как простой текст. И кроме пробелов нет никаких «пустых» текстовых узлов. Ну и отлично, потому что большую часть времени нас будут интересовать узлы-элементы.

Клик по этой кнопке в левом верхнем углу инспектора позволяет при помощи мыши (или другого устройства ввода) выбрать элемент на веб-странице и «проинспектировать» его (браузер сам найдёт и отметит его во вкладке Elements). Этот способ отлично подходит, когда у нас огромная HTML-страница (и соответствующий ей огромный DOM), и мы хотим увидеть, где находится интересующий нас элемент.

Есть и другой способ сделать это: можно кликнуть на странице по элементу правой кнопкой мыши и в контекстном меню выбрать «Inspect».



В правой части инструментов разработчика находятся следующие подразделы:

- **Styles** – здесь мы видим CSS, применённый к текущему элементу: правило за правилом, включая встроенные стили (выделены серым). Почти всё можно отредактировать на месте, включая размеры, внешние и внутренние отступы.
- **Computed** – здесь мы видим итоговые CSS-свойства элемента, которые он приобрёл в результате применения всего каскада стилей (в том числе унаследованные свойства и т.д.).
- **Event Listeners** – в этом разделе мы видим обработчики событий, привязанные к DOM-элементам (мы поговорим о них в следующей части учебника).
- ... И т.д.

Лучший способ изучить инструменты разработчика – это прокликать их. Большинство значений можно менять и тут же смотреть результат.

Взаимодействие с консолью

При работе с DOM нам часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок, как перемещаться между вкладками Elements и Console.

Для начала:

1. На вкладке Elements выберите первый элемент ``.
2. Нажмите `Esc` – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как `$0`; предыдущий, выбранный до него, как `$1` и т.д.

Теперь мы можем запускать на них команды. Например `$0.style.background = 'red'` сделает выбранный элемент красным, как здесь:

The screenshot shows the Chrome DevTools interface with the Elements tab selected. In the main pane, there is a heading "The truth about elks." followed by two list items: "1. An elk is a smart" and "2. ...and cunning animal!". Below this, the DOM tree shows the structure of the page. In the styles panel, a rule for `element.style` is shown with `background: red;`. The console tab at the bottom has a red box around the command `> $0.style.background = 'red'` and its result `< "red"`.

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    "The truth about elks."
    "
  <ol>
    ...
      <li style="background: red;">An elk is a smart</li>
      <!-- comment -->
    <li>
      ...
    </li>
  </ol>
</body>
</html>
```

Styles Computed Event Listeners >
Filter :hov .cls +
element.style {
 background: red;
}
li { user agent stylesheet
 display: list-item;
 text-align: -webkit-match-parent;
}

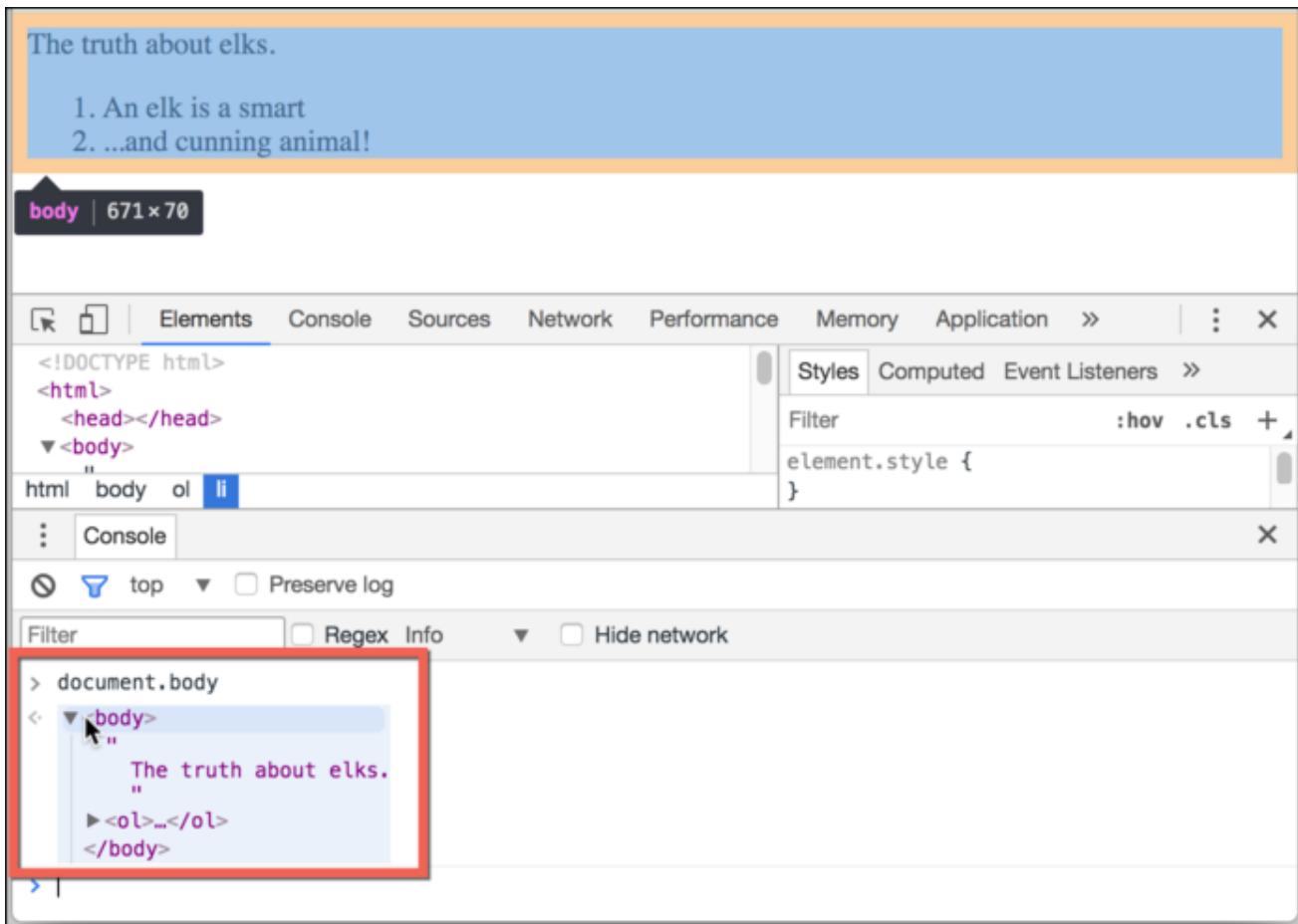
Console
Preserve log
Filter Regex Info Hide network

```
> $0.style.background = 'red'  
< "red"  
> |
```

Это мы посмотрели как получить узел из Elements в Console.

Есть и обратный путь: если есть переменная `node`, ссылающаяся на DOM-узел, можно использовать в консоли команду `inspect(node)`, чтобы увидеть этот элемент во вкладке Elements.

Или мы можем просто вывести DOM-узел в консоль и исследовать «на месте», как `document.body` ниже:



Это может быть полезно для отладки. В следующей главе мы рассмотрим доступ и изменение DOM при помощи JavaScript.

Инструменты разработчика браузера отлично помогают в разработке: мы можем исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

Итого

HTML/XML документы представлены в браузере в виде DOM-дерева.

- Теги становятся узлами-элементами и формируют структуру документа.
- Текст становится текстовыми узлами.
- ... и т.д. Всё, что записано в HTML, есть и в DOM-дереве, даже комментарии.

Для изменения элементов или проверки DOM-дерева мы можем использовать инструменты разработчика в браузере.

Здесь мы рассмотрели основы, наиболее часто используемые и важные действия для начала разработки. Подробную документацию по инструментам разработки Chrome Developer Tools можно найти на странице <https://developers.google.com/web/tools/chrome-devtools>. Лучший способ изучить инструменты – походить по разным вкладкам, почитать меню: большинство действий очевидны для пользователя. Позже, когда вы немного их изучите, прочитайте документацию и узнайте то, что осталось.

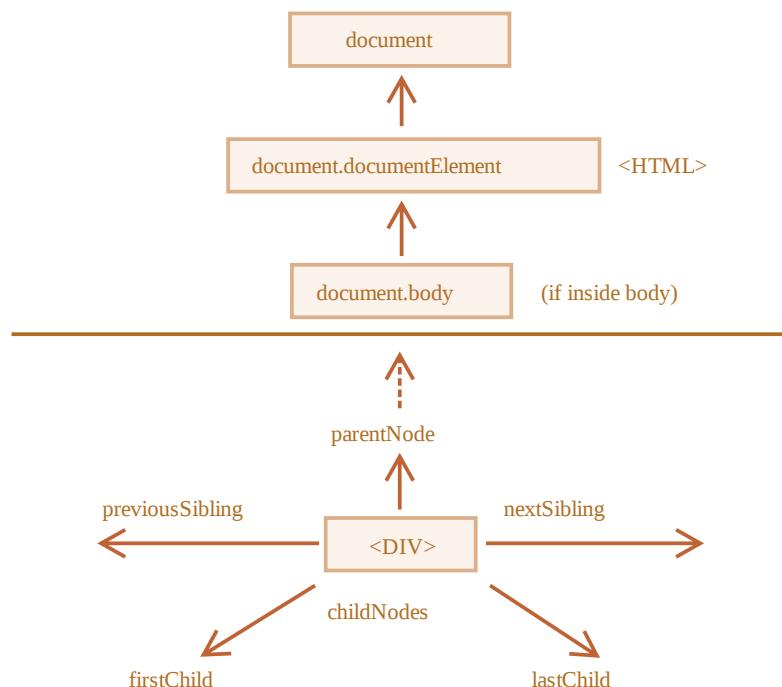
У DOM-узлов есть свойства и методы, которые позволяют выбирать любой из элементов, изменять, перемещать их на странице и многое другое. Мы вернёмся к ним в последующих разделах.

Навигация по DOM-элементам

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект.

Все операции с DOM начинаются с объекта `document`. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу.

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



Поговорим об этом подробнее.

Сверху: `documentElement` и `body`

Самые верхние элементы дерева доступны как свойства объекта `document`:

```
<html> = document.documentElement
```

Самый верхний узел документа: `document.documentElement`. В DOM он соответствует тегу `<html>`.

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

Другой часто используемый DOM-узел – узел тега `<body>`: `document.body`.

```
<head> = document.head
```

Тег `<head>` доступен как `document.head`.

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

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

В частности, если скрипт находится в `<head>`, `document.body` в нём недоступен, потому что браузер его ещё не прочитал.

Поэтому, в примере ниже первый `alert` выведет `null`:

```
<html>

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

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

ⓘ В мире DOM `null` означает «не существует»

В DOM значение `null` значит «не существует» или «нет такого узла».

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

Здесь и далее мы будем использовать два принципиально разных термина:

- **Дочерние узлы (или дети)** – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, `<head>` и `<body>` являются детьми элемента `<html>`.
- **Потомки** – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

В примере ниже детьми тега `<body>` являются теги `<div>` и `` (и несколько пустых текстовых узлов):

```
<html>
<body>
  <div>Начало</div>

  <ul>
    <li>
      <b>Информация</b>
    </li>
  </ul>
</body>
</html>
```

...А потомки `<body>` – это и прямые дети `<div>`, `` и вложенные в них: `` (ребёнок ``) и `` (ребёнок ``) – в общем, все элементы поддерева.

Коллекция `childNodes` содержит список всех детей, включая текстовые узлы.

Пример ниже последовательно выведет детей `document.body`:

```
<html>
<body>
  <div>Начало</div>

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

  <div>Конец</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT
    }
  </script>
  ...какой-то HTML-код...
</body>
</html>
```

Обратим внимание на маленькую деталь. Если запустить пример выше, то последним будет выведен элемент `<script>`. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его.

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

Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:

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

Для проверки наличия дочерних узлов существует также специальная функция `elem.hasChildNodes()`.

DOM-коллекции

Как мы уже видели, `childNodes` похож на массив. На самом деле это не массив, а коллекция – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

1. Для перебора коллекции мы можем использовать `for...of`:

```
for (let node of document.body.childNodes) {  
  alert(node); // покажет все узлы из коллекции  
}
```

Это работает, потому что коллекция является перебираемым объектом (есть требуемый для этого метод `Symbol.iterator`).

2. Методы массивов не будут работать, потому что коллекция – это не массив:

```
alert(document.body.childNodes.filter); // undefined (у коллекции нет метода filter!)
```

Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить. Если нам хочется использовать именно методы массива, то мы можем создать настоящий массив из коллекции, используя `Array.from`:

```
alert( Array.from(document.body.childNodes).filter ); // сделали массив
```

DOM-коллекции – только для чтения

DOM-коллекции, и даже более – все навигационные свойства, перечисленные в этой главе, доступны только для чтения.

Мы не можем заменить один дочерний узел на другой, просто написав `childNodes[i] = ...`.

Для изменения DOM требуются другие методы. Мы увидим их в следующей главе.

DOM-коллекции живые

Почти все DOM-коллекции, за небольшим исключением, *живые*. Другими словами, они отражают текущее состояние DOM.

Если мы сохраним ссылку на `elem.childNodes` и добавим/удадим узлы в DOM, то они появятся в сохранённой коллекции автоматически.

Не используйте цикл `for .. in` для перебора коллекций

Коллекции перебираются циклом `for .. of`. Некоторые начинающие разработчики пытаются использовать для этого цикл `for .. in`.

Не делайте так. Цикл `for .. in` перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно нам не нужны:

```
<body>
<script>
  // выводит 0, 1, length, item, values и другие свойства.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

Соседи и родитель

Соседи – это узлы, у которых один и тот же родитель.

Например, здесь `<head>` и `<body>` соседи:

```
<html>
  <head>...</head><body>...</body>
</html>
```

- говорят, что `<body>` – «следующий» или «правый» сосед `<head>`
- также можно сказать, что `<head>` «предыдущий» или «левый» сосед `<body>`.

Следующий узел того же родителя (следующий сосед) – в свойстве `nextSibling`, а предыдущий – в `previousSibling`.

Родитель доступен через `parentNode`.

Например:

```
// родителем <body> является <html>
alert( document.body.parentNode === document.documentElement ); // выведет true

// после <head> идёт <body>
alert( document.head.nextSibling ); // HTMLBodyElement

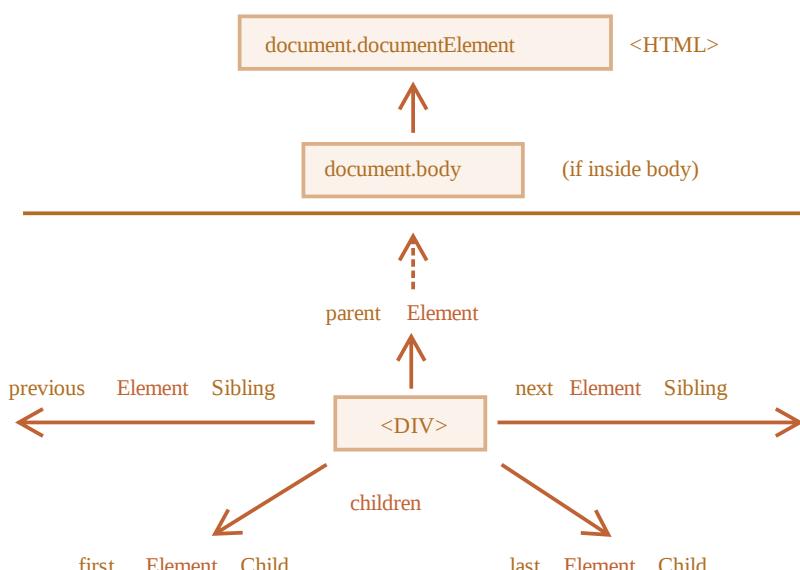
// перед <body> находится <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Навигация только по элементам

Навигационные свойства, описанные выше, относятся ко *всем* узлам в документе. В частности, в `childNodes` находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть.

Но для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы.

Поэтому давайте рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



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

- `children` – коллекция детей, которые являются элементами.
- `firstElementChild`, `lastElementChild` – первый и последний дочерний элемент.
- `previousElementSibling`, `nextElementSibling` – соседи-элементы.
- `parentElement` – родитель-элемент.

i Зачем нужен `parentElement`? Разве может родитель быть не элементом?

Свойство `parentElement` возвращает родитель-элемент, а `parentNode` возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя.

За исключением `document.documentElement`:

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

Причина в том, что родителем корневого узла

`document.documentElement (<html>)` является `document`. Но `document` – это не узел-элемент, так что `parentNode` вернёт его, а `parentElement` нет.

Эта деталь может быть полезна, если мы хотим пройти вверх по цепочке родителей от произвольного элемента `elem` к `<html>`, но не до `document`:

```
while(elem = elem.parentElement) { // идти наверх до <html>
  alert( elem );
}
```

Изменим один из примеров выше: заменим `childNodes` на `children`. Теперь цикл выводит только элементы:

```
<html>
<body>
  <div>Начало</div>

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

  <div>Конец</div>
```

```
<script>
  for (let elem of document.body.children) {
    alert(elem); // DIV, UL, DIV, SCRIPT
  }
</script>
...
</body>
</html>
```

Ещё немного ссылок: таблицы

До сих пор мы описывали основные навигационные ссылки.

Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа.

Таблицы – отличный пример таких элементов.

Элемент `<table>`, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

- `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 больше одного).

`<thead>`, `<tfoot>`, `<tbody>` предоставляют свойство `rows`:

- `tbody.rows` – коллекция строк `<tr>` секции.

`<tr>`:

- `tr.cells` – коллекция `<td>` и `<th>` ячеек, находящихся внутри строки `<tr>`.
- `tr.sectionRowIndex` – номер строки `<tr>` в текущей секции `<thead>/<tbody>/<tfoot>`.
- `tr.rowIndex` – номер строки `<tr>` в таблице (включая все строки таблицы).

`<td>` and `<th>`:

- `td.cellIndex` – номер ячейки в строке `<tr>`.

Пример использования:

```
<table id="table">
```

```

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

<script>
  // выводит содержимое первой строки, второй ячейки
  alert( table.rows[0].cells[1].innerHTML ) // "два"
</script>

```

Спецификация: [tabular data ↗](#).

Существуют также дополнительные навигационные ссылки для HTML-форм. Мы рассмотрим их позже, когда начнём работать с формами.

Итого

Получив DOM-узел, мы можем перейти к его ближайшим соседям используя навигационные ссылки.

Есть два основных набора ссылок:

- Для всех узлов: `parentNode`, `childNodes`, `firstChild`, `lastChild`, `previousSibling`, `nextSibling`.
- Только для узлов-элементов: `parentElement`, `children`, `firstElementChild`, `lastElementChild`, `previousElementSibling`, `nextElementSibling`.

Некоторые виды DOM-элементов, например таблицы, предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому.

✓ Задачи

Дочерние элементы в DOM

важность: 5

Для страницы:

```

<html>
<body>
  <div>Пользователи:</div>
  <ul>
    <li>Джон</li>
    <li>Пит</li>

```

```
</ul>
</body>
</html>
```

Напишите код, как получить...

- элемент `<div>`?
- ``?
- второй `` (с именем Пит)?

[К решению](#)

Вопрос о соседях

важность: 5

Если `elem` – произвольный узел DOM-элемента...

- Правда, что `elem.lastChild.nextSibling` всегда равен `null`?
- Правда, что `elem.children[0].previousSibling` всегда равен `null`?

[К решению](#)

Выделите ячейки по диагонали

важность: 5

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

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

```
// в переменной td находится 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*`, `querySelector*`

Свойства навигации по DOM хороши, когда элементы расположены рядом. А что, если нет? Как получить произвольный элемент страницы?

Для этого в DOM есть дополнительные методы поиска.

`document.getElementById` или просто `id`

Если у элемента есть атрибут `id`, то мы можем получить его вызовом `document.getElementById(id)`, где бы он ни находился.

Например:

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // получить элемент
  let elem = document.getElementById('elem');

  // сделать его фон красным
  elem.style.background = 'red';
</script>
```

Также есть глобальная переменная с именем, указанным в `id`:

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

<script>
  // elem - ссылка на элемент с id="elem"
  elem.style.background = 'red';

  // внутри id="elem-content" есть дефис, так что такой id не может служить именем переменной
  // ...но мы можем обратиться к нему через квадратные скобки: window['elem-content']
</script>
```

...Но это только если мы не объявили в JavaScript переменную с таким же именем, иначе она будет иметь приоритет:

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

<script>
  let elem = 5; // теперь elem равен 5, а не <div id="elem">

  alert(elem); // 5
</script>
```

⚠ Пожалуйста, не используйте такие глобальные переменные для доступа к элементам

Это поведение соответствует [стандарту ↗](#), но поддерживается в основном для совместимости, как осколок далёкого прошлого.

Браузер пытается помочь нам, смешивая пространства имён JS и DOM. Это удобно для простых скриптов, которые находятся прямо в HTML, но, вообще говоря, не очень хорошо. Возможны конфликты имён. Кроме того, при чтении JS-кода, не видя HTML, непонятно, откуда берётся переменная.

В этом учебнике мы будем обращаться к элементам по `id` в примерах для краткости, когда очевидно, откуда берётся элемент.

В реальной жизни лучше использовать `document.getElementById`.

i Значение `id` должно быть уникальным

Значение `id` должно быть уникальным. В документе может быть только один элемент с данным `id`.

Если в документе есть несколько элементов с одинаковым значением `id`, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому, пожалуйста, придерживайтесь правила сохранения уникальности `id`.

⚠ Только `document.getElementById`, а не `anyElem.getElementById`

Метод `getElementById` можно вызвать только для объекта `document`. Он осуществляет поиск по `id` по всему документу.

querySelectorAll

Самый универсальный метод поиска – это `elem.querySelectorAll(css)`, он возвращает все элементы внутри `elem`, удовлетворяющие данному CSS-селектору.

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

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

  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

Этот метод действительно мощный, потому что можно использовать любой CSS-селектор.

i Псевдоклассы тоже работают

Псевдоклассы в CSS-селекторе, в частности `:hover` и `:active`, также поддерживаются. Например, `document.querySelectorAll(':hover')` вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

querySelector

Метод `elem.querySelector(css)` возвращает первый элемент, соответствующий данному CSS-селектору.

Иначе говоря, результат такой же, как при вызове `elem.querySelectorAll(css)[0]`, но он сначала найдёт все элементы, а потом возьмёт первый, в то время как `elem.querySelector` найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

matches

Предыдущие методы искали по DOM.

Метод `elem.matches(css)` ничего не ищет, а проверяет, удовлетворяет ли `elem` CSS-селектору, и возвращает `true` или `false`.

Этот метод удобен, когда мы перебираем элементы (например, в массиве или в чём-то подобном) и пытаемся выбрать те из них, которые нас интересуют.

Например:

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

<script>
    // может быть любая коллекция вместо document.body.children
    for (let elem of document.body.children) {
        if (elem.matches('a[href$="zip"]')) {
            alert("Ссылка на архив: " + elem.href );
        }
    }
</script>
```

closest

Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины.

Метод `elem.closest(css)` ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

Другими словами, метод `closest` поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо `null`, если такой элемент не найден.

Например:

```
<h1>Содержание</h1>

<div class="contents">
    <ul class="book">
        <li class="chapter">Глава 1</li>
        <li class="chapter">Глава 2</li>
    </ul>
</div>

<script>
    let chapter = document.querySelector('.chapter'); // LI

    alert(chapter.closest('.book')); // UL
    alert(chapter.closest('.contents')); // DIV

    alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</script>
```

getElementsBy*

Существуют также другие методы поиска элементов по тегу, классу и так далее.

На данный момент, они скорее исторические, так как `querySelector` более эффективен.

Здесь мы рассмотрим их для полноты картины, также вы можете встретить их в старом коде.

- `elem.getElementsByTagName(tag)` ищет элементы с данным тегом и возвращает их коллекцию. Передав `"*"` вместо тега, можно получить всех ПОТОМКОВ.
- `elem.getElementsByClassName(className)` возвращает элементы, которые имеют данный CSS-класс.
- `document.getElementsByName(name)` возвращает элементы с заданным атрибутом `name`. Очень редко используется.

Например:

```
// получить все элементы div в документе
let divs = document.getElementsByTagName('div');
```

Давайте найдём все `input` в таблице:

```
<table id="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>
  let inputs = table.getElementsByTagName('input');

  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

```
}
```

```
</script>
```

⚠ Не забываем про букву "s" !

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

Буква "s" отсутствует в названии метода `getElementById`, так как в данном случае возвращает один элемент. Но `getElementsByTagName` вернёт список элементов, поэтому "s" обязательна.

⚠ Возвращает коллекцию, а не элемент!

Другая распространённая ошибка – написать:

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

Попытка присвоить значение *коллекции*, а не элементам внутри неё, не сработает.

Нужно перебрать коллекцию в цикле или получить элемент по номеру и уже ему присваивать значение, например, так:

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

Ищем элементы с классом `.article`:

```
<form name="my-form">
  <div class="article">Article</div>
  <div class="long article">Long article</div>
</form>

<script>
  // ищем по имени атрибута
  let form = document.getElementsByName('my-form')[0];

  // ищем по классу внутри form
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, находим два элемента с классом article
</script>
```

Живые коллекции

Все методы "getElementsBy*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении.

В приведённом ниже примере есть два скрипта.

1. Первый создаёт ссылку на коллекцию `<div>`. На этот момент её длина равна `1`.
2. Второй скрипт запускается после того, как браузер встречает ещё один `<div>`, теперь её длина – `2`.

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

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 2
</script>
```

Напротив, `querySelectorAll` возвращает *статическую* коллекцию. Это похоже на фиксированный массив элементов.

Если мы будем использовать его в примере выше, то оба скрипта вернут длину коллекции, равную `1`:

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

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 1
</script>
```

Теперь мы легко видим разницу. Длина статической коллекции не изменилась после появления нового `div` в документе.

Итого

Есть 6 основных методов поиска элементов в DOM:

Метод	Ищет по...	Ищет внутри элемента?	Возвращает живую коллекцию?
querySelector	CSS-selector	✓	-
querySelectorAll	CSS-selector	✓	-
getElementById	id	-	-
getElementsByName	name	-	✓
getElementsByTagName	tag or '*'	✓	✓
getElementsByClassName	class	✓	✓

Безусловно, наиболее часто используемыми в настоящее время являются методы `querySelector` и `querySelectorAll`, но и методы `getElement(s)By*` могут быть полезны в отдельных случаях, а также встречаются в старом коде.

Кроме того:

- Есть метод `elem.matches(css)`, который проверяет, удовлетворяет ли элемент CSS-селектору.
- Метод `elem.closest(css)` ищет ближайшего по иерархии предка, соответствующему данному CSS-селектору. Сам элемент также включён в поиск.

И, напоследок, давайте упомянем ещё один метод, который проверяет наличие отношений между предком и потомком:

- `elemA.contains(elemB)` вернёт `true`, если `elemB` находится внутри `elemA` (`elemB` потомок `elemA`) или когда `elemA==elemB`.

✓ Задачи

Поиск элементов

важность: 4

Вот документ с таблицей и формой.

Как найти?...

1. Таблицу с `id="age-table"`.
2. Все элементы `label` внутри этой таблицы (их три).
3. Первый `td` в этой таблице (со словом «Age»).

4. Форму `form` с именем `name="search"`.

5. Первый `input` в этой форме.

6. Последний `input` в этой форме.

Откройте страницу [table.html](#) в отдельном окне и используйте для этого браузерные инструменты разработчика.

[К решению](#)

Свойства узлов: тип, тег и содержимое

Теперь давайте более внимательно взглянем на DOM-узлы.

В этой главе мы подробнее разберём, что они собой представляют и изучим их основные свойства.

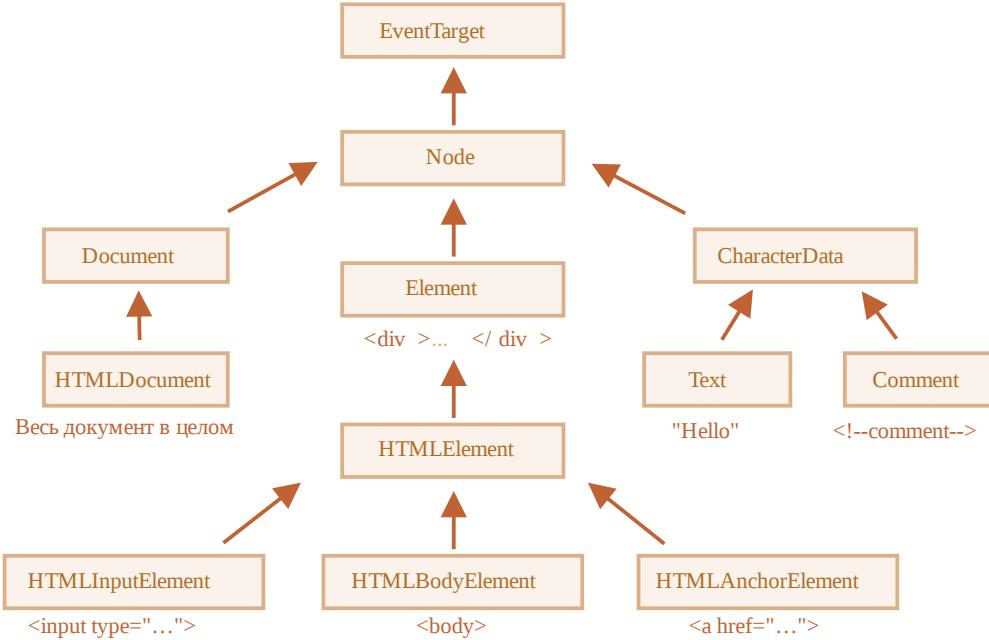
Классы DOM-узлов

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу `<a>`, есть свойства, связанные со ссылками, а у соответствующего тегу `<input>` – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию.

Каждый DOM-узел принадлежит соответствующему встроенному классу.

Корнем иерархии является [EventTarget](#), от него наследует [Node](#) и остальные DOM-узлы.

На рисунке ниже изображены основные классы:



Существуют следующие классы:

- `EventTarget` – это корневой «абстрактный» класс для всего.

Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых мы поговорим позже.

- `Node` – также является «абстрактным» классом, и служит основой для DOM-узлов.

Он обеспечивает базовую функциональность: `parentNode`, `nextSibling`, `childNodes` и т.д. (это геттеры). Объекты класса `Node` никогда не создаются. Но есть определённые классы узлов, которые наследуются от него (и следовательно наследуют функционал `Node`).

- `Document`, по историческим причинам часто наследуется `HTMLDocument` (хотя последняя спецификация этого не наязывает) – это документ в целом.

Глобальный объект `document` принадлежит именно к этому классу. Он служит точкой входа в DOM.

- `CharacterData` – «абстрактный» класс. Вот, кем он наследуется:

- `Text` – класс, соответствующий тексту внутри элементов. Например, `Hello` в `<p>Hello</p>`.
- `Comment` – класс для комментариев. Они не отображаются, но каждый комментарий становится членом DOM.
- `Element` – это базовый класс для DOM-элементов.

Он обеспечивает навигацию на уровне элементов: `nextElementSibling`, `children`. А также и методы поиска элементов: `getElementsByName`, `querySelector`.

Браузер поддерживает не только HTML, но также XML и SVG. Таким образом, класс `Element` служит основой для более специфичных классов: `SVGElement`, `XMLElement` (они нам здесь не нужны) и `HTMLElement`.

- И наконец, `HTMLElement` ↗ является базовым классом для всех остальных HTML-элементов. Мы будем работать с ним большую часть времени.

От него наследуются конкретные элементы:

- `HTMLInputElement` ↗ – класс для тега `<input>`,
- `HTMLBodyElement` ↗ – класс для тега `<body>`,
- `HTMLAnchorElement` ↗ – класс для тега `<a>`,
- ...и т.д.

Также существует множество других тегов со своими собственными классами, которые могут иметь определенные свойства и методы, в то время как некоторые элементы, такие как ``, `<section>` и `<article>`, не имеют каких-либо определенных свойств, поэтому они являются экземплярами класса `HTMLElement`.

Таким образом, полный набор свойств и методов данного узла является результатом цепочки наследования.

Рассмотрим DOM-объект для тега `<input>`. Он принадлежит классу `HTMLInputElement` ↗ .

Он получает свойства и методы из (в порядке наследования):

- `HTMLInputElement` – этот класс предоставляет специфичные для элементов формы свойства,
- `HTMLElement` – предоставляет общие для HTML-элементов методы (и геттеры/сеттеры),
- `Element` – предоставляет типовые методы элемента,
- `Node` – предоставляет общие свойства DOM-узлов,
- `EventTarget` – обеспечивает поддержку событий (поговорим о них дальше),
- ...и, наконец, он наследует от `Object`, поэтому доступны также методы «обычного объекта», такие как `hasOwnProperty`.

Для того, чтобы узнать имя класса DOM-узла, вспомним, что обычно у объекта есть свойство `constructor`. Оно ссылается на конструктор класса, и в свойстве `constructor.name` содержится его имя:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

...Или мы можем просто привести его к строке :

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

Проверить наследование можно также 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
alert( document.body instanceof EventTarget ); // true
```

Как видно, DOM-узлы – это обычные JavaScript объекты. Для наследования они используют классы, основанные на прототипах.

В этом легко убедиться, если вывести в консоли браузера любой элемент через `console.dir(elem)`. Или даже напрямую обратиться к методам, которые хранятся в `HTMLElement.prototype`, `Element.prototype` и т.д.

i `console.dir(elem)` и `console.log(elem)`

Большинство браузеров поддерживают в инструментах разработчика две команды: `console.log` и `console.dir`. Они выводят свои аргументы в консоль. Для JavaScript-объектов эти команды обычно выводят одно и то же.

Но для DOM-элементов они работают по-разному:

- `console.log(elem)` выводит элемент в виде DOM-дерева.
- `console.dir(elem)` выводит элемент в виде DOM-объекта, что удобно для анализа его свойств.

Попробуйте сами на `document.body`.

Спецификация IDL

В спецификации для описания классов DOM используется не JavaScript, а специальный язык [Interface description language ↗](#) (IDL), с которым достаточно легко разобраться.

В IDL все свойства представлены с указанием их типов. Например, `DOMString`, `boolean` и т.д.

Небольшой отрывок IDL с комментариями:

```
// Объявление HTMLInputElement
// Двоеточие ":" после HTMLInputElement означает, что он наследует от HTMLElement
interface HTMLInputElement: HTMLElement {
    // далее идут свойства и методы элемента <input>

    // "DOMString" означает, что значение свойства - строка
    attribute DOMString accept;
    attribute DOMString alt;
    attribute DOMString autocomplete;
    attribute DOMString value;

    // boolean - значит, что autofocus хранит логический тип данных (true/false)
    attribute boolean autofocus;
    ...

    // "void" перед методом означает, что данный метод не возвращает значение
    void select();
    ...

}
```

Свойство «nodeType»

Свойство `nodeType` предоставляет ещё один, «старомодный» способ узнать «типа» DOM-узла.

Его значением является цифра:

- `elem.nodeType == 1` для узлов-элементов,
- `elem.nodeType == 3` для текстовых узлов,
- `elem.nodeType == 9` для объектов документа,
- В спецификации ↗ можно посмотреть остальные значения.

Например:

```
<body>
  <script>
    let elem = document.body;
```

```
// давайте разберёмся: какой тип узла находится в elem?  
alert(elem.nodeType); // 1 => элемент  
  
// и его первый потомок...  
alert(elem.firstChild.nodeType); // 3 => текст  
  
// для объекта document значение типа -- 9  
alert( document.nodeType ); // 9  
</script>  
</body>
```

В современных скриптах, чтобы узнать тип узла, мы можем использовать метод `instanceof` и другие способы проверить класс, но иногда `nodeType` проще использовать. Мы не можем изменить значение `nodeType`, только прочитать его.

Тег: `nodeName` и `tagName`

Получив DOM-узел, мы можем узнать имя его тега из свойств `nodeName` и `tagName`:

Например:

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

Есть ли какая-то разница между `tagName` и `nodeName`?

Да, она отражена в названиях свойств, но не очевидна.

- Свойство `tagName` есть только у элементов `Element`.
- Свойство `nodeName` определено для любых узлов `Node`:
 - для элементов оно равно `tagName`.
 - для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

Другими словами, свойство `tagName` есть только у узлов-элементов (поскольку они происходят от класса `Element`), а `nodeName` может что-то сказать о других типах узлов.

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

```
<body><! --- комментарий --->
```

```
<script>
    // для комментария
    alert( document.body.firstChild.tagName ); // undefined (не элемент)
    alert( document.body.firstChild.nodeName ); // #comment

    // for document
    alert( document.tagName ); // undefined (не элемент)
    alert( document.nodeName ); // #document
</script>
</body>
```

Если мы имеем дело только с элементами, то можно использовать `tagName` или `nodeName`, нет разницы.

i Имена тегов (кроме XHTML) всегда пишутся в верхнем регистре

В браузере существуют два режима обработки документа: HTML и XML. HTML-режим обычно используется для веб-страниц. XML-режим включается, если браузер получает XML-документ с заголовком: `Content-Type: application/xml+xhtml`.

В HTML-режиме значения `tagName/nodeName` всегда записаны в верхнем регистре. Будет выведено `BODY` вне зависимости от того, как записан тег в HTML `<body>` или `<BoDy>`.

В XML-режиме регистр сохраняется «как есть». В настоящее время XML-режим применяется редко.

innerHTML: содержимое элемента

Свойство [innerHTML](#) позволяет получить HTML-содержимое элемента в виде строки.

Мы также можем изменять его. Это один из самых мощных способов менять содержимое на странице.

Пример ниже показывает содержимое `document.body`, а затем полностью заменяет его:

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

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

</body>
```

Мы можем попробовать вставить некорректный HTML, браузер исправит наши ошибки:

```
<body>

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

</body>
```

Скрипты не выполняются

Если `innerHTML` вставляет в документ тег `<script>` – он становится частью HTML, но не запускается.

Будьте внимательны: «`innerHTML+=`» осуществляет перезапись
Мы можем добавить HTML к элементу, используя `elem.innerHTML+="ещё html"`.

Вот так:

```
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.

Технически эти две строки делают одно и то же:

```
elem.innerHTML += "...";
// это более короткая запись для:
elem.innerHTML = elem.innerHTML + "..."
```

Другими словами, `innerHTML+=` делает следующее:

1. Старое содержимое удаляется.
2. На его место становится новое значение `innerHTML` (с добавленной строкой).

Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены.

В примере `chatDiv` выше строка `chatDiv.innerHTML+="Как дела?"` заново создаёт содержимое HTML и перезагружает `smile.gif` (надеемся, картинка закеширована). Если в `chatDiv` много текста и изображений, то эта перезагрузка будет очень заметна.

Есть и другие побочные эффекты. Например, если существующий текст выделен мышкой, то при переписывании `innerHTML` большинство браузеров снимут выделение. А если это поле ввода `<input>` с текстом, введённым пользователем, то текст будет удалён. И т.д.

К счастью, есть и другие способы добавить содержимое, не использующие `innerHTML`, которые мы изучим позже.

outerHTML: HTML элемента целиком

Свойство `outerHTML` содержит HTML элемента целиком. Это как `innerHTML` плюс сам элемент.

Посмотрим на пример:

```
<div id="elem">Привет <b>Мир</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>
</script>
```

Будьте осторожны: в отличие от `innerHTML`, запись в `outerHTML` не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте.

Да, звучит странно, и это действительно необычно, поэтому здесь мы и отмечаем это особо.

Рассмотрим пример:

```
<div>Привет, мир!</div>

<script>
  let div = document.querySelector('div');

  // заменяем div.outerHTML на <p>...</p>
  div.outerHTML = '<p>Новый элемент</p>'; // (*)

  // Содержимое div осталось тем же!
  alert(div.outerHTML); // <div>Привет, мир!</div> (**)
</script>
```

Какая-то магия, да?

В строке `(*)` мы заменили `div` на `<p>Новый элемент</p>`. Во внешнем документе мы видим новое содержимое вместо `<div>`. Но, как видно в строке `(**)`, старая переменная `div` осталась прежней!

Это потому, что использование `outerHTML` не изменяет DOM-элемент, а удаляет его из внешнего контекста и вставляет вместо него новый HTML-код.

То есть, при `div.outerHTML=...` произошло следующее:

- `div` был удалён из документа.
- Вместо него был вставлен другой HTML `<p>Новый элемент</p>`.
- В `div` осталось старое значение. Новый HTML не сохранён ни в какой переменной.

Здесь легко сделать ошибку: заменить `div.outerHTML`, а потом продолжить работать с `div`, как будто там новое содержимое. Но это не так. Подобное верно для `innerHTML`, но не для `outerHTML`.

Мы можем писать в `elem.outerHTML`, но надо иметь в виду, что это не меняет элемент, в который мы пишем. Вместо этого создаётся новый HTML на его месте. Мы можем получить ссылки на новые элементы, обратившись к DOM.

nodeValue/data: содержимое текстового узла

Свойство `innerHTML` есть только у узлов-элементов.

У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства `nodeValue` и `data`. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Мы будем использовать `data`, потому что оно короче.

Прочитаем содержимое текстового узла и комментария:

```
<body>
    Привет
    <!-- Комментарий -->
    <script>
        let text = document.body.firstChild;
        alert(text.data); // Привет

        let comment = text.nextSibling;
        alert(comment.data); // Комментарий
    </script>
</body>
```

Мы можем представить, для чего нам может понадобиться читать или изменять текстовый узел, но комментарии?

Иногда их используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:

```
<!-- if isAdmin -->
<div>Добро пожаловать, Admin!</div>
<!-- /if -->
```

...Затем JavaScript может прочитать это из свойства `data` и обработать инструкции.

textContent: просто текст

Свойство `textContent` предоставляет доступ к *тексту* внутри элемента за вычетом всех `<тегов>`.

Например:

```
<div id="news">
  <h1>Срочно в номер!</h1>
  <p>Марсиане атаковали человечество!</p>
</div>

<script>
  // Срочно в номер! Марсиане атаковали человечество!
  alert(news.textContent);
</script>
```

Как мы видим, возвращается только текст, как если бы все `<теги>` были вырезаны, но текст в них остался.

На практике редко появляется необходимость читать текст таким образом.

Намного полезнее возможность записывать текст в `textContent`, т.к. позволяет писать текст «безопасным способом».

Представим, что у нас есть произвольная строка, введённая пользователем, и мы хотим показать её.

- С `innerHTML` вставка происходит «как HTML», со всеми HTML-тегами.
- С `textContent` вставка получается «как текст», все символы трактуются буквально.

Сравним два тега `div`:

```
<div id="elem1"></div>
<div id="elem2"></div>

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

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

1. В первый `<div>` имя приходит «как HTML»: все теги стали именно тегами, поэтому мы видим имя, выделенное жирным шрифтом.
2. Во второй `<div>` имя приходит «как текст», поэтому мы видим `Винни-пух!`.

В большинстве случаев мы рассчитываем получить от пользователя текст и хотим, чтобы он интерпретировался как текст. Мы не хотим, чтобы на сайте появлялся произвольный HTML-код. Присваивание через `textContent` – один из способов от этого защититься.

Свойство «hidden»

Атрибут и DOM-свойство «`hidden`» указывает на то, видим ли мы элемент или нет.

Мы можем использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

```
<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

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

Технически, `hidden` работает так же, как `style="display:none"`. Но его применение проще.

Мигающий элемент:

```
<div id="elem">Мигающий элемент</div>

<script>
```

```
setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

Другие свойства

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

- `value` – значение для `<input>`, `<select>` и `<textarea>` (`HTMLInputElement`, `HTMLSelectElement` ...).
- `href` – адрес ссылки «`href`» для `` (`HTMLAnchorElement`).
- `id` – значение атрибута «`id`» для всех элементов (`HTMLElement`).
- ...и многие другие...

Например:

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

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

Большинство стандартных HTML-атрибутов имеют соответствующее DOM-свойство, и мы можем получить к нему доступ.

Если мы хотим узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс `HTMLInputElement` описывается здесь: [https://html.spec.whatwg.org/#htmlinputelement ↗](https://html.spec.whatwg.org/#htmlinputelement).

Если же нам нужно быстро что-либо узнать или нас интересует специфика определённого браузера – мы всегда можем вывести элемент в консоль, используя `console.dir(elem)`, и прочитать все свойства. Или исследовать «свойства DOM» во вкладке `Elements` браузерных инструментов разработчика.

Итого

Каждый DOM-узел принадлежит определённому классу. Классы формируют иерархию. Весь набор свойств и методов является результатом наследования.

Главные свойства DOM-узла:

`nodeType`

Свойство `nodeType` позволяет узнать тип DOM-узла. Его значение – числовое: 1 для элементов, 3 для текстовых узлов, и т.д. Только для чтения.

`nodeName/tagName`

Для элементов это свойство возвращает название тега (записывается в верхнем регистре, за исключением XML-режима). Для узлов-неэлементов `nodeName` описывает, что это за узел. Только для чтения.

`innerHTML`

Внутреннее HTML-содержимое узла-элемента. Можно изменять.

`outerHTML`

Полный HTML узла-элемента. Запись в `elem.outerHTML` не меняет `elem`. Вместо этого она заменяет его во внешнем контексте.

`nodeValue/data`

Содержимое узла-неэлемента (текст, комментарий). Эти свойства практически одинаковые, обычно мы используем `data`. Можно изменять.

`textContent`

Текст внутри элемента: HTML за вычетом всех `<тегов>`. Запись в него помещает текст в элемент, при этом все специальные символы и теги интерпретируются как текст. Можно использовать для защиты от вставки произвольного HTML кода.

`hidden`

Когда значение установлено в `true`, делает то же самое, что и CSS `display:none`.

В зависимости от своего класса DOM-узлы имеют и другие свойства. Например у элементов `<input>` (`HTMLInputElement`) есть свойства `value`, `type`, у элементов `<a>` (`HTMLAnchorElement`) есть `href` и т.д. Большинство стандартных HTML-атрибутов имеют соответствующие свойства DOM.

Впрочем, HTML-атрибуты и свойства DOM не всегда одинаковы, мы увидим это в следующей главе.

✓ Задачи

Считаем потомков

важность: 5

У нас есть дерево, структурированное как вложенные списки `ul/li`.

Напишите код, который выведет каждый элемент списка ``:

1. Какой в нём текст (без поддерева) ?
2. Какое число потомков – всех вложенных `` (включая глубоко вложенные) ?

[Демо в новом окне ↗](#)

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Что содержит свойство `nodeType`?

важность: 5

Что выведет этот код?

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

[К решению](#)

Тег в комментарии

важность: 3

Что выведет этот код?

```
<script>  
  let 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 от `Node` или от `Element`, или может от `HTMLElement`?

[К решению](#)

Атрибуты и свойства

Когда браузер загружает страницу, он «читает» (также говорят: «парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов.

Например, для такого тега `<body id="page">` у DOM-объекта будет такое свойство `body.id="page"`.

Но преобразование атрибута в свойство происходит не один-в-один! В этой главе мы уделим внимание различию этих двух понятий, чтобы посмотреть, как работать с ними, когда они одинаковые и когда разные.

DOM-свойства

Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

DOM-узлы – это обычные объекты JavaScript. Мы можем их изменять.

Например, создадим новое свойство для `document.body`:

```
document.body.myData = {  
    name: 'Caesar',  
    title: 'Imperator'  
};  
  
alert(document.body.myData.title); // Imperator
```

Мы можем добавить и метод:

```
document.body.sayTagName = function() {  
    alert(this.tagName);  
};
```

```
document.body.sayTagName(); // BODY (значением "this" в этом методе будет document.body)
```

Также можно изменять встроенные прототипы, такие как `Element.prototype` и добавлять новые методы ко всем элементам:

```
Element.prototype.sayHi = function() {
  alert(`Hello, I'm ${this.tagName}`);
};

document.documentElement.sayHi(); // Hello, I'm HTML
document.body.sayHi(); // Hello, I'm BODY
```

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript:

- Им можно присвоить любое значение.
- Они регистрозависимы (нужно писать `elem.nodeType`, не `elem.NoDeTyPe`).

HTML-атрибуты

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт *стандартные* атрибуты и создаёт DOM-свойства для них.

Таким образом, когда у элемента есть `id` или другой *стандартный* атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут *нестандартный*.

Например:

```
<body id="test" something="non-standard">
  <script>
    alert(document.body.id); // test
    // нестандартный атрибут не преобразуется в свойство
    alert(document.body.something); // undefined
  </script>
</body>
```

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть *нестандартным* для другого. Например, атрибут `"type"` является стандартным для элемента `<input>` ([HTMLInputElement](#) ↗), но не является стандартным для `<body>` ([HTMLBodyElement](#) ↗). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента.

Мы можем увидеть это на примере ниже:

```
<body id="body" type="...>
  <input id="input" type="text">
  <script>
    alert(input.type); // text
    alert(body.type); // undefined: DOM-свойство не создалось, потому что оно нестандартное
  </script>
</body>
```

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Есть ли способ получить такие атрибуты?

Конечно. Все атрибуты доступны с помощью следующих методов:

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

Эти методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства `elem.attributes`: коллекция объектов, которая принадлежит ко встроенному классу [Attr](#) со свойствами `name` и `value`.

Вот демонстрация чтения нестандартного свойства:

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // non-standard
  </script>
</body>
```

У HTML-атрибутов есть следующие особенности:

- Их имена регистронезависимы (`id` то же самое, что и `ID`).
- Их значения всегда являются строками.

Расширенная демонстрация работы с атрибутами:

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

  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant', чтение
```

```

elem.setAttribute('Test', 123); // (2), запись

alert( elem.outerHTML ); // (3), посмотрим, есть ли атрибут в HTML (да)

for (let attr of elem.attributes) { // (4) весь список
    alert(` ${attr.name} = ${attr.value}`);
}
</script>
</body>

```

Пожалуйста, обратите внимание:

1. `getAttribute('About')` – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Мы можем присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке мы получаем значение `"123"`.
3. Все атрибуты, в том числе те, которые мы установили, видны в `outerHTML`.
4. Коллекция `attributes` является перебираемой. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами `name` и `value`.

Синхронизация между атрибутами и свойствами

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями).

В примере ниже `id` модифицируется как атрибут, и можно увидеть, что свойство также изменено. То же самое работает и в обратную сторону:

```

<input>

<script>
let input = document.querySelector('input');

// атрибут => свойство
input.setAttribute('id', 'id');
alert(input.id); // id (обновлено)

// свойство => атрибут
input.id = 'newId';
alert(input.getAttribute('id')); // newId (обновлено)
</script>

```

Но есть и исключения, например, `input.value` синхронизируется только в одну сторону – атрибут → значение, но не в обратную:

```

<input>

<script>
  let input = document.querySelector('input');

  // атрибут => значение
  input.setAttribute('value', 'text');
  alert(input.value); // text

  // свойство => атрибут
  input.value = 'newValue';
  alert(input.getAttribute('value')); // text (не обновилось!)
</script>

```

В примере выше:

- Изменение атрибута `value` обновило свойство.
- Но изменение свойства не повлияло на атрибут.

Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям `value`, и если после этого мы захотим восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

DOM-свойства типизированы

DOM-свойства не всегда являются строками. Например, свойство `input.checked` (для чекбоксов) имеет логический тип:

```

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

<script>
  alert(input.getAttribute('checked')); // значение атрибута: пустая строка
  alert(input.checked); // значение свойства: true
</script>

```

Есть и другие примеры. Атрибут `style` – строка, но свойство `style` является объектом:

```

<div id="div" style="color:red;font-size:120%">Hello</div>

<script>
  // строка
  alert(div.getAttribute('style')); // color:red;font-size:120%

  // объект
  alert(div.style); // [object CSSStyleDeclaration]

```

```
    alert(div.style.color); // red
</script>
```

Хотя большинство свойств, всё же, строки.

При этом некоторые из них, хоть и строки, могут отличаться от атрибутов. Например, DOM-свойство `href` всегда содержит *полный* URL, даже если атрибут содержит относительный URL или просто `#hash`.

Ниже пример:

```
<a id="a" href="#hello">link</a>
<script>
  // атрибут
  alert(a.getAttribute('href')); // #hello

  // свойство
  alert(a.href); // полный URL в виде http://site.com/page#hello
</script>
```

Если же нужно значение `href` или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться `getAttribute`.

Нестандартные атрибуты, dataset

При написании HTML мы используем много стандартных атрибутов. Но что насчёт нестандартных, пользовательских? Давайте посмотрим, полезны они или нет, и для чего они нужны.

Иногда нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript.

Как тут:

```
<!-- пометить div, чтобы показать здесь поле "name" -->
<div show-info="name"></div>
<!-- а здесь возраст "age" -->
<div show-info="age"></div>

<script>
  // код находит элемент с пометкой и показывает запрошенную информацию
  let user = {
    name: "Pete",
    age: 25
  };

  for(let div of document.querySelectorAll('[show-info]')) {
```

```
// вставить соответствующую информацию в поле
let field = div.getAttribute('show-info');
div.innerHTML = user[field]; // сначала Pete в name, потом 25 в age
}
</script>
```

Также они могут быть использованы, чтобы стилизовать элементы.

Например, здесь для состояния заказа используется атрибут `order-state`:

```
<style>
/* стили зависят от пользовательского атрибута "order-state" */
.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">
  A new order.
</div>

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

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

Почему атрибут может быть предпочтительнее таких классов, как `.order-state-new`, `.order-state-pending`, `order-state-canceled`?

Это потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто:

```
// немного проще, чем удаление старого/добавление нового класса
div.setAttribute('order-state', 'canceled');
```

Но с пользовательскими атрибутами могут возникнуть проблемы. Что если мы используем нестандартный атрибут для наших целей, а позже он появится в стандарте и будет выполнять какую-то функцию? Язык HTML живой, он растёт,

появляется больше атрибутов, чтобы удовлетворить потребности разработчиков. В этом случае могут возникнуть неожиданные эффекты.

Чтобы избежать конфликтов, существуют атрибуты вида `data-* ↗`.

Все атрибуты, начинающиеся с префикса «`data-`», зарезервированы для использования программистами. Они доступны в свойстве `dataset`.

Например, если у `elem` есть атрибут `"data-about"`, то обратиться к нему можно как `elem.dataset.about`.

Как тут:

```
<body data-about="Elephants">
<script>
  alert(document.body.dataset.about); // Elephants
</script>
```

Атрибуты, состоящие из нескольких слов, к примеру `data-order-state`, становятся свойствами, записанными с помощью верблюжьей нотации: `dataset.orderState`.

Вот переписанный пример «состояния заказа»:

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

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

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

<div id="order" class="order" data-order-state="new">
  A new order.
</div>

<script>
  // чтение
  alert(order.dataset.orderState); // new

  // изменение
  order.dataset.orderState = "pending"; // (*)
</script>
```

Использование `data-*` атрибутов – валидный, безопасный способ передачи пользовательских данных.

Пожалуйста, примите во внимание, что мы можем не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка `(*)` меняет цвет на синий.

Итого

- Атрибуты – это то, что написано в HTML.
- Свойства – это то, что находится в DOM-объектах.

Небольшое сравнение:

	Свойства	Атрибуты
Тип	Любое значение, стандартные свойства имеют типы, описанные в спецификации	Строка
Имя	Имя регистрозависимо	Имя регистронезависимо

Методы для работы с атрибутами:

- `elem.hasAttribute(name)` – проверить на наличие.
- `elem.getAttribute(name)` – получить значение.
- `elem.setAttribute(name, value)` – установить значение.
- `elem.removeAttribute(name)` – удалить атрибут.
- `elem.attributes` – это коллекция всех атрибутов.

В большинстве ситуаций предпочтительнее использовать DOM-свойства. Нужно использовать атрибуты только тогда, когда DOM-свойства не подходят, когда нужны именно атрибуты, например:

- Нужен нестандартный атрибут. Но если он начинается с `data-`, тогда нужно использовать `dataset`.
- Мы хотим получить именно то значение, которое написано в HTML. Значение DOM-свойства может быть другим, например, свойство `href` – всегда полный URL, а нам может понадобиться получить «оригинальное» значение.

✓ Задачи

Получите атрибут

важность: 5

Напишите код для выбора элемента с атрибутом `data-widget-name` из документа и прочтайте его значение.

```
<!DOCTYPE html>
<html>
<body>

<div data-widget-name="menu">Choose the genre</div>

<script>
/* your code */
</script>
</body>
</html>
```

К решению

Сделайте внешние ссылки оранжевыми

важность: 3

Сделайте все внешние ссылки оранжевыми, изменяя их свойство `style`.

Ссылка является внешней, если:

- Её `href` содержит `://`
- Но не начинается с `http://internal.com`.

Пример:

```
<a name="list">the list</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>

<script>
  // добавление стиля для одной ссылки
  let link = document.querySelector('a');
  link.style.color = 'orange';
</script>
```

Результат должен быть таким:

The list:

- <https://google.com>
- </tutorial.html>
- [local/path](#)
- <ftp://ftp.com/my.zip>
- <https://nodejs.org>
- <http://internal.com/test>

Открыть песочницу для задачи. ↗

К решению

Изменение документа

Модификации DOM – это ключ к созданию «живых» страниц.

Здесь мы увидим, как создавать новые элементы «на лету» и изменять уже существующие.

Пример: показать сообщение

Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем `alert`.

Вот такое:

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

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

Всем привет! Вы прочитали важное сообщение.

Это был пример HTML. Теперь давайте создадим такой же `div`, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

Создание элемента

DOM-узел можно создать двумя методами:

`document.createElement(tag)`

Создаёт новый элемент с заданным тегом:

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

`document.createTextNode(text)`

Создаёт новый текстовый узел с заданным текстом:

```
let textNode = document.createTextNode('А вот и я');
```

Большую часть времени нам нужно создавать узлы элементов, такие как `div` для сообщения.

Создание сообщения

В нашем случае сообщение – это `div` с классом `alert` и HTML в нём:

```
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение。";
```

Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа.

Методы вставки

Чтобы наш `div` появился, нам нужно вставить его где-нибудь в `document`. Например, в `document.body`.

Для этого есть метод `append`, в нашем случае:

```
document.body.append(div).
```

Вот полный пример:

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

```

background-color: #dff0d8;
}
</style>

<script>
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);
</script>

```

Вот методы для различных вариантов вставки:

- `node.append(...nodes or strings)` – добавляет узлы или строки в конец `node`,
- `node.prepend(...nodes or strings)` – вставляет узлы или строки в начало `node`,
- `node.before(...nodes or strings)` – вставляет узлы или строки до `node`,
- `node.after(...nodes or strings)` – вставляет узлы или строки после `node`,
- `node.replaceWith(...nodes or strings)` – заменяет `node` заданными узлами или строками.

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

```

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

<script>
ol.before('before'); // вставить строку "before" перед <ol>
ol.after('after'); // вставить строку "after" после <ol>

let liFirst = document.createElement('li');
liFirst.innerHTML = 'prepend';
ol.prepend(liFirst); // вставить liFirst в начало <ol>

let liLast = document.createElement('li');
liLast.innerHTML = 'append';
ol.append(liLast); // вставить liLast в конец <ol>
</script>

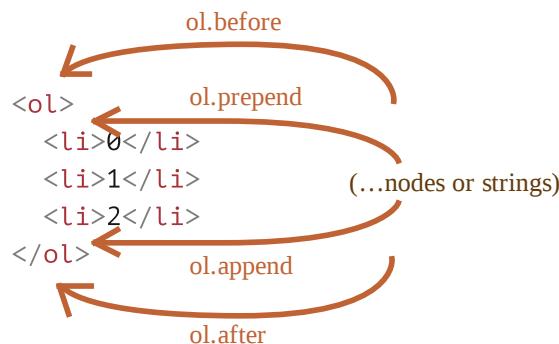
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Наглядная иллюстрация того, куда эти методы вставляют:



Итоговый список будет таким:

```
before
<ol id="ol">
  <li>prepend</li>
  <li>0</li>
  <li>1</li>
  <li>2</li>
  <li>append</li>
</ol>
after
```

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов.

Например, здесь вставляется строка и элемент:

```
<div id="div"></div>
<script>
  div.before( '<p>Привет</p>' , document.createElement( 'hr' ) );
</script>
```

Весь текст вставляется как текст.

Поэтому финальный HTML будет:

```
&lt;p&gt;Привет&lt;/p&gt;  
<hr>  
<div id="div"></div>
```

Другими словами, строки вставляются безопасным способом, как делает это `elem.textContent`.

Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов.

А что, если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это `elem.innerHTML`?

insertAdjacentHTML/Text/Element

С этим может помочь другой, довольно универсальный метод:

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

Первый параметр – это специальное слово, указывающее, куда по отношению к `elem` производить вставку. Значение должно быть одним из следующих:

- `"beforebegin"` – вставить `html` непосредственно перед `elem`,
- `"afterbegin"` – вставить `html` в начало `elem`,
- `"beforeend"` – вставить `html` в конец `elem`,
- `"afterend"` – вставить `html` непосредственно после `elem`.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML».

Например:

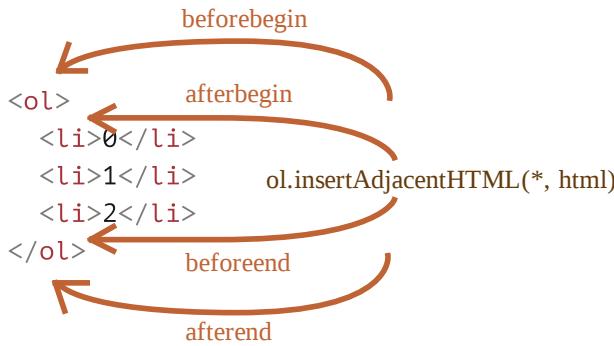
```
<div id="div"></div>  
<script>  
  div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');  
  div.insertAdjacentHTML('afterend', '<p>Пока</p>');  
</script>
```

...Приведёт к:

```
<p>Привет</p>  
<div id="div"></div>  
<p>Пока</p>
```

Так мы можем добавлять произвольный HTML на страницу.

Варианты вставки:



Мы можем легко заметить сходство между этой и предыдущей картинкой. Точки вставки фактически одинаковые, но этот метод вставляет HTML.

У метода есть два брата:

- `elem.insertAdjacentText(where, text)` – такой же синтаксис, но строка `text` вставляется «как текст», вместо HTML,
- `elem.insertAdjacentElement(where, elem)` – такой же синтаксис, но вставляет элемент `elem`.

Они существуют, в основном, чтобы унифицировать синтаксис. На практике часто используется только `insertAdjacentHTML`. Потому что для элементов и текста у нас есть методы `append/prepend/before/after` – их быстрее написать, и они могут вставлять как узлы, так и текст.

Так что, вот альтернативный вариант показа сообщения:

```

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

<script>
document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>`);
</script>

```

Удаление узлов

Для удаления узла есть методы `node.remove()`.

Например, сделаем так, чтобы наше сообщение удалялось через секунду:

```

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

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
</script>

```

Если нам нужно *переместить* элемент в другое место – нет необходимости удалять его со старого.

Все методы вставки автоматически удаляют узлы со старых мест.

Например, давайте поменяем местами элементы:

```

<div id="first">Первый</div>
<div id="second">Второй</div>
<script>
  // нет необходимости вызывать метод remove
  second.after(first); // берёт #second и после него вставляет #first
</script>

```

Клонирование узлов: `cloneNode`

Как вставить ещё одно подобное сообщение?

Мы могли бы создать функцию и поместить код туда. Альтернатива – **клонировать** существующий `div` и изменить текст внутри него (при необходимости).

Иногда, когда у нас есть большой элемент, это может быть быстрее и проще.

- Вызов `elem.cloneNode(true)` создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем `elem.cloneNode(false)`, тогда клон будет без дочерних элементов.

Пример копирования сообщения:

```

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

<div class="alert" id="div">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>

<script>
let div2 = div.cloneNode(true); // клонировать сообщение
div2.querySelector('strong').innerHTML = 'Всем пока!';
// изменить клонированный элемент

div.after(div2); // показать клонированный элемент после существующего div
</script>

```

DocumentFragment

`DocumentFragment` является специальным DOM-узлом, который служит обёрткой для передачи списков узлов.

Мы можем добавить к нему другие узлы, но когда мы вставляем его куда-то, он «исчезает», вместо него вставляется его содержимое.

Например, `getListContent` ниже генерирует фрагмент с элементами ``, которые позже вставляются в ``:

```

<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>

```

Обратите внимание, что на последней строке с (*) мы добавляем `DocumentFragment`, но он «исчезает», поэтому структура будет:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

`DocumentFragment` редко используется. Зачем добавлять элементы в специальный вид узла, если вместо этого мы можем вернуть массив узлов? Переписанный пример:

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append + оператор "..." = друзья!
</script>
```

Мы упоминаем `DocumentFragment` в основном потому, что он используется в некоторых других областях, например, для элемента `template`, который мы рассмотрим гораздо позже.

Устаревшие методы вставки/удаления

Старая школа

Эта информация помогает понять старые скрипты, но не нужна для новой разработки.

Есть несколько других, более старых, методов вставки и удаления, которые существуют по историческим причинам.

Сейчас уже нет причин их использовать, так как современные методы `append`, `prepend`, `before`, `after`, `remove`, `replaceWith` более гибкие и удобные.

Мы упоминаем о них только потому, что их можно найти во многих старых скриптах:

`parentElem.appendChild(node)`

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

Следующий пример добавляет новый `` в конец ``:

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

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

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

`parentElem.insertBefore(node, nextSibling)`

Вставляет `node` перед `nextSibling` в `parentElem`.

Следующий пример вставляет новый элемент перед вторым ``:

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

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

Чтобы вставить `newLi` в начало, мы можем сделать вот так:

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

`parentElem.replaceChild(node, oldChild)`

Заменяет `oldChild` на `node` среди дочерних элементов `parentElem`.

`parentElem.replaceChild(node)`

Удаляет `node` из `parentElem` (предполагается, что он родитель `node`).

Этот пример удалит первый `` из ``:

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

<script>
  let li = list.firstElementChild;
  list.removeChild(li);
</script>
```

Все эти методы возвращают вставленный/удалённый узел. Другими словами, `parentElem.appendChild(node)` вернёт `node`. Но обычно возвращаемое значение не используют, просто вызывают метод.

Несколько слов о «`document.write`»

Есть ещё один, очень древний метод добавления содержимого на веб-страницу: `document.write`.

Синтаксис:

```
<p>Где-то на странице...</p>
<script>
  document.write('<b>Привет из JS</b>');
</script>
<p>Конец</p>
```

Вызов `document.write(html)` записывает `html` на страницу «прямо здесь и сейчас». Стока `html` может быть динамически сгенерирована, поэтому метод достаточно гибкий. Мы можем использовать JavaScript, чтобы создать полноценную веб-страницу и записать её в документ.

Этот метод пришёл к нам со времён, когда ещё не было ни DOM, ни стандартов... Действительно старые времена. Он всё ещё живёт, потому что есть скрипты, которые используют его.

В современных скриптах он редко встречается из-за следующего важного ограничения:

Вызов `document.write` работает только во время загрузки страницы.

Если вызвать его позже, то существующее содержимое документа затрётся.

Например:

```
<p>Через одну секунду содержимое этой страницы будет заменено...</p>
<script>
  // document.write через 1 секунду
  // вызов происходит после того, как страница загрузится, поэтому метод затирает сод
  setTimeout(() => document.write('<b>...Этим.</b>'), 1000);
</script>
```

Так что после того, как страница загружена, он уже непригоден к использованию, в отличие от других методов DOM, которые мы рассмотрели выше.

Это его недостаток.

Есть и преимущество. Технически, когда `document.write` запускается во время чтения HTML браузером, и что-то пишет в документ, то браузер воспринимает это так, как будто это изначально было частью загруженного HTML-документа.

Поэтому он работает невероятно быстро, ведь при этом *нет модификации DOM*. Метод пишет прямо в текст страницы, пока DOM ещё в процессе создания.

Так что, если нам нужно динамически добавить много текста в HTML, и мы находимся на стадии загрузки, и для нас очень важна скорость, это может помочь. Но на практике эти требования редко сочетаются. И обычно мы можем увидеть этот метод в скриптах просто потому, что они старые.

Итого

- Методы для создания узлов:
 - `document.createElement(tag)` – создаёт элемент с заданным тегом,
 - `document.createTextNode(value)` – создаёт текстовый узел (редко используется),
 - `elem.cloneNode(deep)` – клонирует элемент, если `deep==true`, то со всеми дочерними элементами.
- Вставка и удаление:
 - `node.append(...nodes or strings)` – вставляет в `node` в конец,
 - `node.prepend(...nodes or strings)` – вставляет в `node` в начало,
 - `node.before(...nodes or strings)` – вставляет прямо перед `node`,
 - `node.after(...nodes or strings)` – вставляет сразу после `node`,
 - `node.replaceWith(...nodes or strings)` – заменяет `node`.

- `node.remove()` – удаляет `node`.
- Устаревшие методы:
 - `parent.appendChild(node)`
 - `parent.insertBefore(node, nextSibling)`
 - `parent.removeChild(node)`
 - `parent.replaceChild(newElem, node)`

Все эти методы возвращают `node`.

- Если нужно вставить фрагмент HTML, то `elem.insertAdjacentHTML(where, html)` вставляет в зависимости от `where`:
 - "beforebegin" – вставляет `html` прямо перед `elem`,
 - "afterbegin" – вставляет `html` в `elem` в начало,
 - "beforeend" – вставляет `html` в `elem` в конец,
 - "afterend" – вставляет `html` сразу после `elem`.

Также существуют похожие методы `elem.insertAdjacentText` и `elem.insertAdjacentElement`, они вставляют текстовые строки и элементы, но они редко используются.

- Чтобы добавить HTML на страницу до завершения её загрузки:
 - `document.write(html)`

После загрузки страницы такой вызов затирает документ. В основном встречается в старых скриптах.

✓ Задачи

createTextNode vs innerHTML vs textContent

важность: 5

У нас есть пустой DOM-элемент `elem` и строка `text`.

Какие из этих 3-х команд работают одинаково?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`
3. `elem.textContent = text`

[К решению](#)

Очистите элемент

важность: 5

Создайте функцию `clear(elem)`, которая удаляет всё содержимое из `elem`.

```
<ol id="elem">
  <li>Привет</li>
  <li>Мир</li>
</ol>

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

  clear(elem); // очищает список
</script>
```

[К решению](#)

Почему остаётся "aaa"?

важность: 1

В примере ниже вызов `table.remove()` удаляет таблицу из документа.

Но если вы запустите его, вы увидите, что текст "aaa" все еще виден.

Почему так происходит?

```
<table id="table">
  aaa
  <tr>
    <td>Тест</td>
  </tr>
</table>

<script>
  alert(table); // таблица, как и должно быть

  table.remove();
  // почему в документе остался текст "aaa"?
</script>
```

[К решению](#)

Создайте список

важность: 4

Напишите интерфейс для создания списка.

Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью `prompt`.
2. Создавайте элемент `` и добавляйте его к ``.
3. Продолжайте до тех пор, пока пользователь не отменит ввод (нажатием клавиши `Esc` или введя пустую строку).

Все элементы должны создаваться динамически.

Если пользователь вводит HTML-теги, они должны обрабатываться как текст.

[Демо в новом окне](#) ↗

[К решению](#)

Создайте дерево из объекта

важность: 5

Напишите функцию `createTree`, которая создаёт вложенный список `ul/li` из объекта.

Например:

```
let data = {
  "Рыбы": {
    "форель": {},
    "лосось": {}
  },
  "деревья": {
    "Огромные": {
      "секвойя": {},
      "дуб": {}
    },
    "Цветковые": {
      "яблоня": {},
      "магнолия": {}
    }
  }
};
```

Синтаксис:

```
let container = document.getElementById('container');
createTree(container, data); // создаёт дерево в контейнере
```

Результат (дерево):

- Рыбы
 - форель
 - лосось
- Деревья
 - Огромные
 - секвойя
 - дуб
 - Цветковые
 - яблоня
 - магнолия

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](#)

Выведите список потомков в дереве

важность: 5

Есть дерево, организованное в виде вложенных списков `ul/li`.

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](#)

Создайте календарь в виде таблицы

важность: 4

Напишите функцию `createCalendar(elem, year, month)`.

Вызов функции должен создать календарь для заданного месяца `month` в году `year` и вставить его в `elem`.

Календарь должен быть таблицей, где неделя – это `<tr>`, а день – это `<td>`. У таблицы должен быть заголовок с названиями дней недели, каждый день – `<th>`, первым днём недели должен быть понедельник.

Например, `createCalendar(cal, 2012, 9)` сгенерирует в `cal` 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. В этой задаче достаточно сгенерировать календарь, кликабельным его делать не нужно.

[Открыть песочницу для задачи.](#)

[К решению](#)

Цветные часы с использованием setInterval

важность: 4

Создайте цветные часы как в примере ниже:

```
hh:mm:ss  
Start Stop
```

Для стилизации используйте HTML/CSS, JavaScript должен только обновлять время в элементах.

[Открыть песочницу для задачи.](#)

[К решению](#)

Вставьте HTML в список

важность: 5

Напишите код для вставки `23` между этими двумя ``:

```
<ul id="ul">  
  <li id="one">1</li>  
  <li id="two">4</li>  
</ul>
```

[К решению](#)

Сортировка таблицы

важность: 5

Вот таблица:

```
<table>  
<thead>  
  <tr>  
    <th>Name</th><th>Surname</th><th>Age</th>  
  </tr>  
</thead>  
<tbody>
```

```
<tr>
  <td>John</td><td>Smith</td><td>10</td>
</tr>
<tr>
  <td>Pete</td><td>Brown</td><td>15</td>
</tr>
<tr>
  <td>Ann</td><td>Lee</td><td>5</td>
</tr>
<tr>
  <td>...</td><td>...</td><td>...</td>
</tr>
</tbody>
</table>
```

В ней может быть больше строк.

Напишите код для сортировки по столбцу "name".

[Открыть песочницу для задачи.](#)

[К решению](#)

Стили и классы

До того, как начнёте изучать способы работы со стилями и классами в JavaScript, есть одно важное правило. Надеемся, это достаточно очевидно, но мы всё равно должны об этом упомянуть.

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: `<div class="...">`
2. Писать стили непосредственно в атрибуте `style: <div style="...">`.

JavaScript может менять и классы, и свойство `style`.

Классы – всегда предпочтительный вариант по сравнению со `style`. Мы должны манипулировать свойством `style` только в том случае, если классы «не могут справиться».

Например, использование `style` является приемлемым, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript:

```
let top = /* сложные расчёты */;
let left = /* сложные расчёты */;

elem.style.left = left; // например, '123px', значение вычисляется во время работы ск
elem.style.top = top; // например, '456px'
```

В других случаях, например, чтобы сделать текст красным, добавить значок фона – описываем это в CSS и добавляем класс (JavaScript может это сделать). Это более гибкое и лёгкое в поддержке решение.

className и classList

Изменение класса является одним из наиболее часто используемых действий в скриптах.

Когда-то давно в JavaScript существовало ограничение: зарезервированное слово типа "class" не могло быть свойством объекта. Это ограничение сейчас отсутствует, но в то время было невозможно иметь свойство `elem.class`.

Поэтому для классов было введено схожее свойство "className": `elem.className` соответствует атрибуту "class".

Например:

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

Если мы присваиваем что-то `elem.className`, то это заменяет всю строку с классами. Иногда это то, что нам нужно, но часто мы хотим добавить/удалить один класс.

Для этого есть другое свойство: `elem.classList`.

`elem.classList` – это специальный объект с методами для добавления/удаления одного класса.

Например:

```
<body class="main page">
  <script>
    // добавление класса
    document.body.classList.add('article');

    alert(document.body.className); // main page article
  </script>
</body>
```

Так что мы можем работать как со строкой полного класса, используя `className`, так и с отдельными классами, используя `classList`. Выбираем

тот вариант, который нам удобнее.

Методы `classList`:

- `elem.classList.add/remove("class")` – добавить/удалить класс.
- `elem.classList.toggle("class")` – добавить класс, если его нет, иначе удалить.
- `elem.classList.contains("class")` – проверка наличия класса, возвращает `true/false`.

Кроме того, `classList` является перебираемым, поэтому можно перечислить все классы при помощи `for..of`:

```
<body class="main page">
  <script>
    for (let name of document.body.classList) {
      alert(name); // main, затем page
    }
  </script>
</body>
```

Element style

Свойство `elem.style` – это объект, который соответствует тому, что написано в атрибуте `"style"`. Установка стиля `elem.style.width="100px"` работает так же, как наличие в атрибуте `style` строки `width:100px`.

Для свойства из нескольких слов используется camelCase:

```
background-color => elem.style.backgroundColor
z-index        => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Например:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Свойства с префиксом

Стили с браузерным префиксом, например, `-moz-border-radius`, `-webkit-border-radius` преобразуются по тому же принципу: дефис означает заглавную букву.

Например:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Сброс стилей

Иногда нам нужно добавить свойство стиля, а потом, позже, убрать его.

Например, чтобы скрыть элемент, мы можем задать `elem.style.display = "none"`.

Затем мы можем удалить свойство `style.display`, чтобы вернуться к первоначальному состоянию. Вместо `delete elem.style.display` мы должны присвоить ему пустую строку: `elem.style.display = ""`.

```
// если мы запустим этот код, <body> "мигнёт"
document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000); // возврат к нормальному со
```

Если мы установим в `style.display` пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства `style.display` вообще не было.

Полная перезапись `style.cssText`

Обычно мы используем `style.*` для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, `div.style="color: red; width: 100px"`, потому что `div.style` – это объект, и он доступен только для чтения.

Для задания нескольких стилей в одной строке используется специальное свойство `style.cssText`:

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

<script>
    // можем даже устанавливать специальные флаги для стилей, например, "important"
    div.style.cssText = `color: red !important;
        background-color: yellow;
        width: 100px;
        text-align: center;
    `;

    alert(div.style.cssText);
</script>
```

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно ненароком удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда мы точно знаем, что не удалим существующий стиль.

То же самое можно сделать установкой атрибута:

```
div.setAttribute('style', 'color: red...').
```

Следите за единицами измерения

Не забудьте добавить к значениям единицы измерения.

Например, мы должны устанавливать `10px`, а не просто `10` в свойство `elem.style.top`. Иначе это не сработает:

```
<body>
    <script>
        // не работает!
        document.body.style.margin = 20;
        alert(document.body.style.margin); // '' (пустая строка, присваивание игнорируется)

        // сейчас добавим единицу измерения (px) - и заработает
        document.body.style.margin = '20px';
```

```
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
</script>
</body>
```

Пожалуйста, обратите внимание, браузер «распаковывает» свойство `style.margin` в последних строках и выводит `style.marginLeft` и `style.marginTop` из него.

Вычисленные стили: `getComputedStyle`

Итак, изменить стиль очень просто. Но как его прочитать?

Например, мы хотим знать размер, отступы, цвет элемента. Как это сделать?

Свойство `style` оперирует только значением атрибута "style", без учёта CSS-каскада.

Поэтому, используя `elem.style`, мы не можем прочитать ничего, что приходит из классов CSS.

Например, здесь `style` не может видеть отступы:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  Красный текст
  <script>
    alert(document.body.style.color); // пусто
    alert(document.body.style.marginTop); // пусто
  </script>
</body>
```

...Но что, если нам нужно, скажем, увеличить отступ на `20px`? Для начала нужно его текущее значение получить.

Для этого есть метод: `getComputedStyle`.

Синтаксис:

```
getComputedStyle(element, [pseudo])
```

`element`

Элемент, значения для которого нужно получить

pseudo

Указывается, если нужен стиль псевдоэлемента, например `::before`. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на `elem.style`, но с учётом всех CSS-классов.

Например:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

<script>
  let computedStyle = getComputedStyle(document.body);

  // сейчас мы можем прочитать отступ и цвет

  alert( computedStyle.marginTop ); // 5px
  alert( computedStyle.color ); // rgb(255, 0, 0)
</script>

</body>
```

➊ Вычисленное (computed) и окончательное (resolved) значения

Есть две концепции в [CSS ↗](#):

1. *Вычисленное (computed)* значение – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, `height:1em` или `font-size:125%`.
2. *Окончательное (resolved ↗)* значение – непосредственно применяемое к элементу. Значения `1em` или `125%` являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, `height:20px` или `font-size:16px`. Для геометрических свойств разрешённые значения могут иметь плавающую точку, например, `width:50.5px`.

Давным-давно `getComputedStyle` был создан для получения вычисленных значений, но оказалось, что окончательные значения гораздо удобнее, и стандарт изменился.

Так что, в настоящее время `getComputedStyle` фактически возвращает окончательное значение свойства, для геометрии оно обычно в пикселях.



getComputedStyle требует полное свойство!

Для правильного получения значения нужно указать точное свойство. Например: `paddingLeft`, `marginTop`, `borderTopWidth`. При обращении к сокращённому: `padding`, `margin`, `border` – правильный результат не гарантируется.

Например, если есть свойства `paddingLeft/paddingTop`, то что мы получим вызывая `getComputedStyle(elem).padding`? Ничего, или, может быть, «сгенерированное» значение из известных внутренних отступов? Стандарта для этого нет.

Стили, применяемые к посещённым :visited ссылкам, скрываются!

Посещённые ссылки могут быть окрашены с помощью псевдокласса `:visited`.

Но `getComputedStyle` не даёт доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью `:visited`. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к `:visited` CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для «злой» страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

Итого

Для управления классами существуют два DOM-свойства:

- `className` – строковое значение, удобно для управления всем набором классов.
- `classList` – объект с методами `add/remove/toggle/contains`, удобно для управления отдельными классами.

Чтобы изменить стили:

- Свойство `style` является объектом со стилями в формате camelCase. Чтение и запись в него работают так же, как изменение соответствующих свойств в атрибуте `"style"`. Чтобы узнать, как добавить в него `important` и делать некоторые другие редкие вещи –смотрите [документацию ↗](#).
- Свойство `style.cssText` соответствует всему атрибуту `"style"`, полной строке стилей.

Для чтения окончательных стилей (с учётом всех классов, после применения CSS и вычисления окончательных значений) используется:

- Метод `getComputedStyle(elem, [pseudo])` возвращает объект, похожий по формату на `style`. Только для чтения.

✓ Задачи

Создать уведомление

важность: 5

Напишите функцию `showNotification(options)`, которая создаёт уведомление: `<div class="notification">` с заданным содержимым. Уведомление должно автоматически исчезнуть через 1,5 секунды.

Пример объекта `options`:

```
// показывает элемент с текстом "Hello" рядом с правой верхней частью окна.  
showNotification({  
    top: 10, // 10px от верхней границы окна (по умолчанию 0px)  
    right: 10, // 10px от правого края окна (по умолчанию 0px)  
    html: "Hello!", // HTML-уведомление  
    className: "welcome" // дополнительный класс для div (необязательно)  
});
```

[Демо в новом окне ↗](#)

Используйте CSS-позиционирование для отображения элемента в заданных координатах. Исходный документ имеет необходимые стили.

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Размеры и прокрутка элементов

Существует множество JavaScript-свойств, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. В этой главе мы будем называть их «метрики».

Они часто требуются, когда нам нужно передвигать или позиционировать элементы с помощью JavaScript.

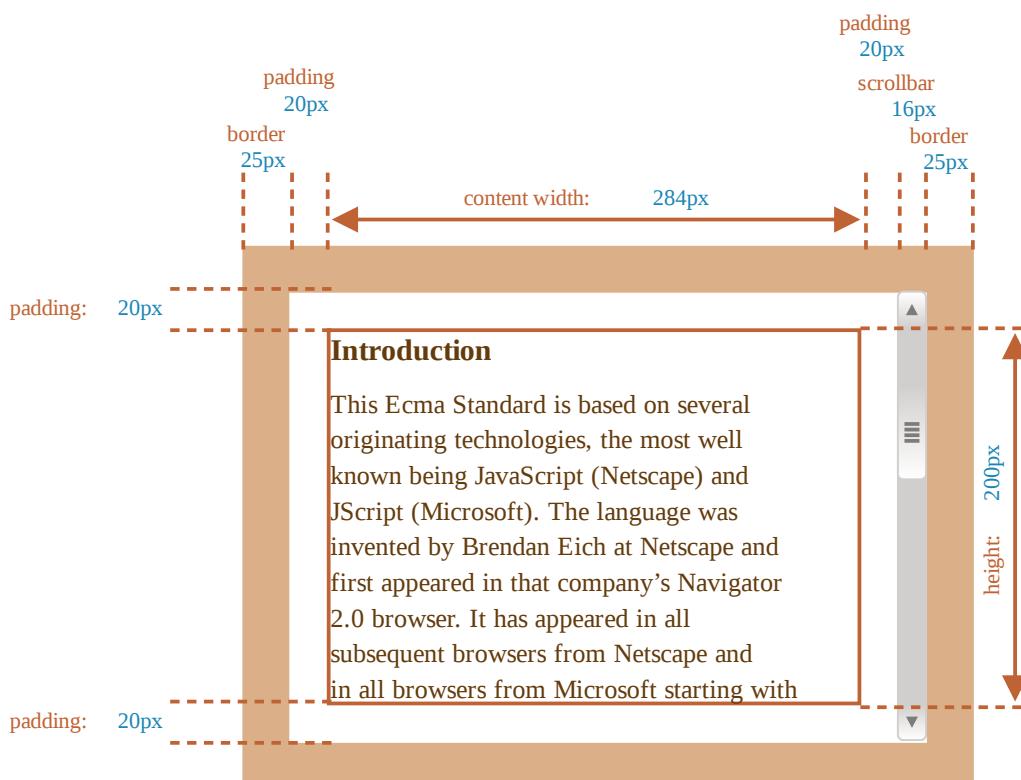
Простой пример

В качестве простого примера демонстрации свойств мы будем использовать следующий элемент:

```
<div id="example">  
    ...Текст...  
</div>  
<style>  
    #example {  
        width: 300px;  
        height: 200px;  
        border: 25px solid #E8C48F;  
        padding: 20px;  
        overflow: auto;  
    }  
</style>
```

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств.

Результат выглядит так:



Вы можете [открыть этот пример в песочнице](#).

Внимание, полоса прокрутки

В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше).

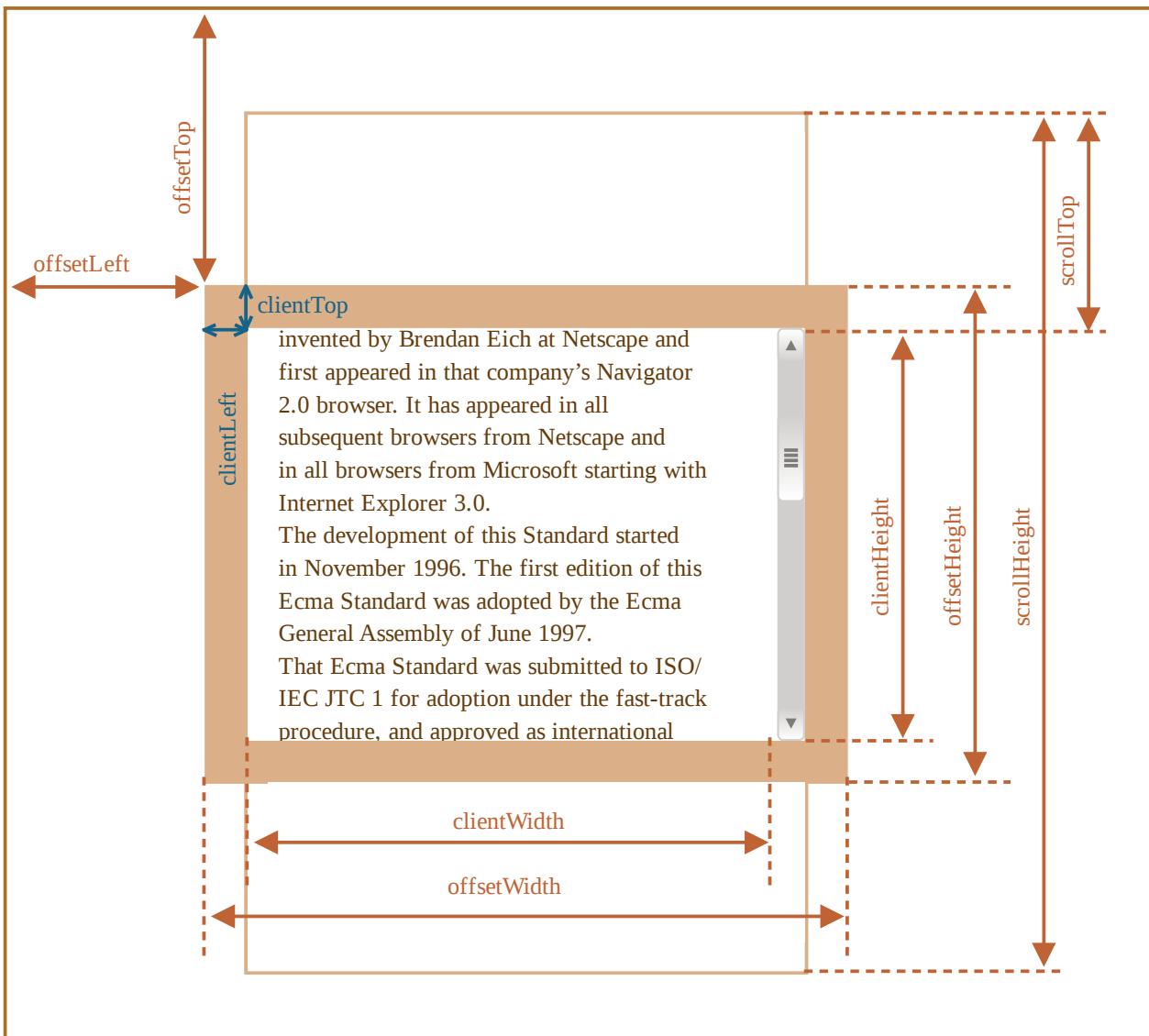
Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только $300 - 16 = 284px$, и мы должны это учитывать. Вот почему примеры в этой главе даны с полосой прокрутки. Без неё некоторые вычисления будут проще.

Область padding-bottom (нижний внутренний отступ) может быть заполнена текстом

Нижние внутренние отступы padding-bottom изображены пустыми на наших иллюстрациях, но если элемент содержит много текста, то он будет перекрывать padding-bottom, это нормально.

Метрики

Вот общая картина с геометрическими свойствами:



Значениями свойств являются числа, подразумевается, что они в пикселях.

Давайте начнём исследовать, начиная снаружи элемента.

offsetParent, offsetLeft/Top

Эти свойства редко используются, но так как они являются «самыми внешними» метриками, мы начнём с них.

В свойстве `offsetParent` находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге.

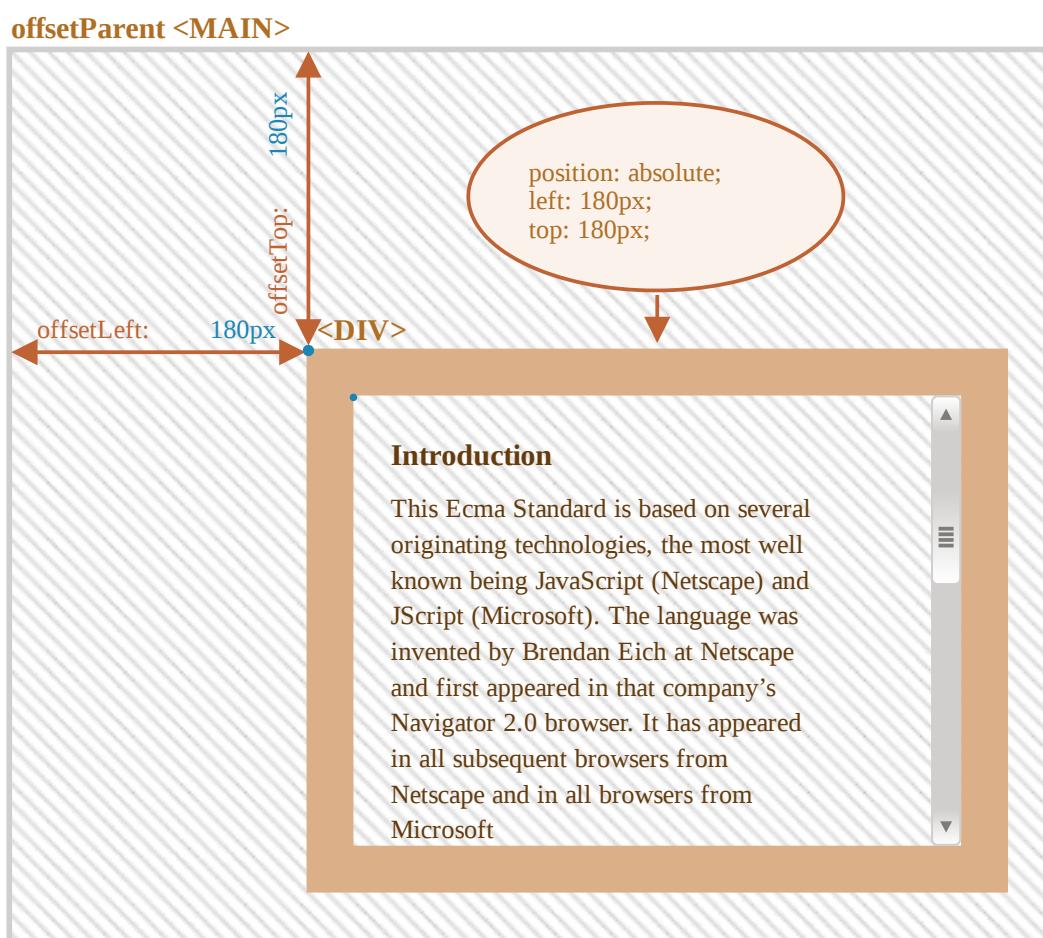
То есть, ближайший предок, который удовлетворяет следующим условиям:

1. Является CSS-позиционированным (CSS-свойство `position` равно `absolute`, `relative`, `fixed` или `sticky`),
2. или `<td>`, `<th>`, `<table>`,
3. или `<body>`.

Свойства `offsetLeft`/`offsetTop` содержат координаты х/у относительно верхнего левого угла `offsetParent`.

В примере ниже внутренний `<div>` имеет элемент `<main>` в качестве `offsetParent`, а свойства `offsetLeft`/`offsetTop` являются сдвигами относительно верхнего левого угла (180):

```
<main style="position: relative" id="main">
  <article>
    <div id="example" style="position: absolute; left: 180px; top: 180px">...</div>
  </article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (обратите внимание: число, а не строка "180px")
  alert(example.offsetTop); // 180
</script>
```



Существует несколько ситуаций, когда `offsetParent` равно `null`:

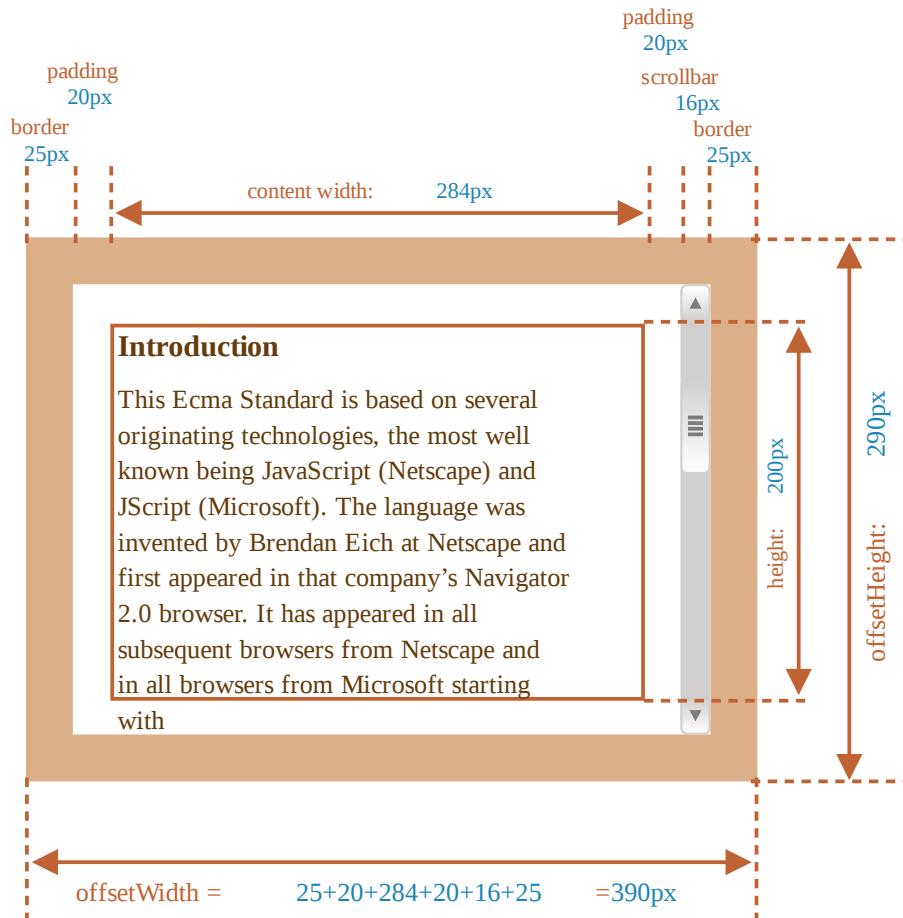
1. Для скрытых элементов (с CSS-свойством `display:none` или когда его нет в документе).
2. Для элементов `<body>` и `<html>`.

3. Для элементов с `position:fixed`.

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.



Для нашего элемента:

- `offsetWidth = 390` – внешняя ширина блока, её можно получить сложением CSS-ширины (300px), внутренних отступов (2 * 20px) и рамок (2 * 25px).
- `offsetHeight = 290` – внешняя высота блока.

Метрики для не показываемых элементов равны нулю.

Координаты и размеры в JavaScript устанавливаются только для видимых элементов.

Если элемент (или любой его родитель) имеет `display:none` или отсутствует в документе, то все его метрики равны нулю (или `null`, если это `offsetParent`).

Например, свойство `offsetParent` равно `null`, а `offsetWidth` и `offsetHeight` равны `0`, когда мы создали элемент, но ещё не вставили его в документ, или если у элемента (или у его родителя) `display:none`.

Мы можем использовать это, чтобы делать проверку на видимость:

```
function isHidden(elem) {
  return !elem.offsetWidth && !elem.offsetHeight;
}
```

Заметим, что функция `isHidden` также вернёт `true` для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые `<div>`).

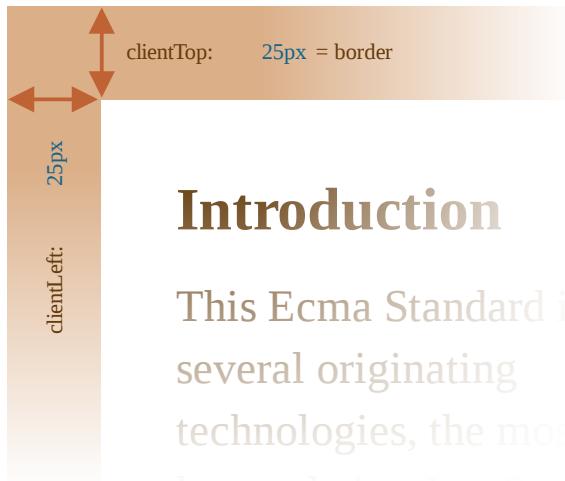
clientTop/Left

Пойдём дальше. 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 based on several originating technologies, the most prominent being:

HTML, CSS, ECMAScript, and DOM.

...Но на самом деле эти свойства – вовсе не ширины рамок, а отступы внутренней части элемента от внешней.

В чём же разница?

Она возникает, 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 ещё и ширину полосы прокрутки.

В этом случае `clientLeft` будет равно `25`, но с прокруткой – `25 + 16 = 41`.

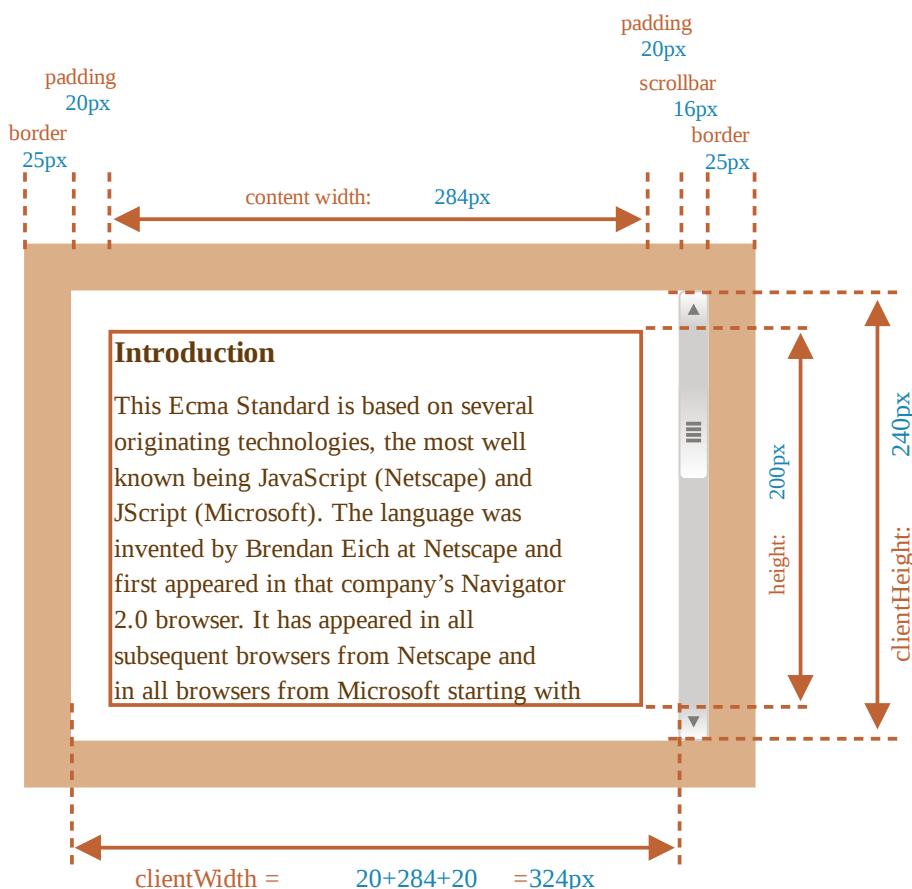
Вот соответствующий пример на иврите:



clientWidth/Height

Эти свойства – размер области внутри рамок элемента.

Они включают в себя ширину области содержимого вместе с внутренними отступами `padding`, но без прокрутки:

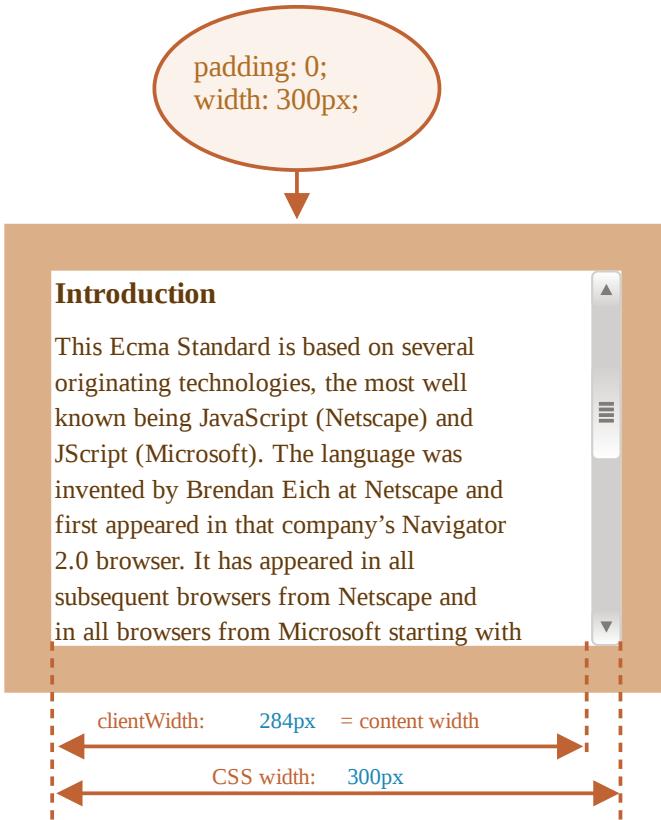


На рисунке выше посмотрим вначале на высоту `clientHeight`.

Горизонтальной прокрутки нет, так что это в точности то, что внутри рамок: CSS-высота `200px` плюс верхние и нижние внутренние отступы ($2 * 20\text{px}$), итого `240px`.

Теперь `clientWidth` – ширина содержимого здесь равна не `300px`, а `284px`, т.к. `16px` отведено для полосы прокрутки. Таким образом: `284px` плюс левый и правый отступы – всего `324px`.

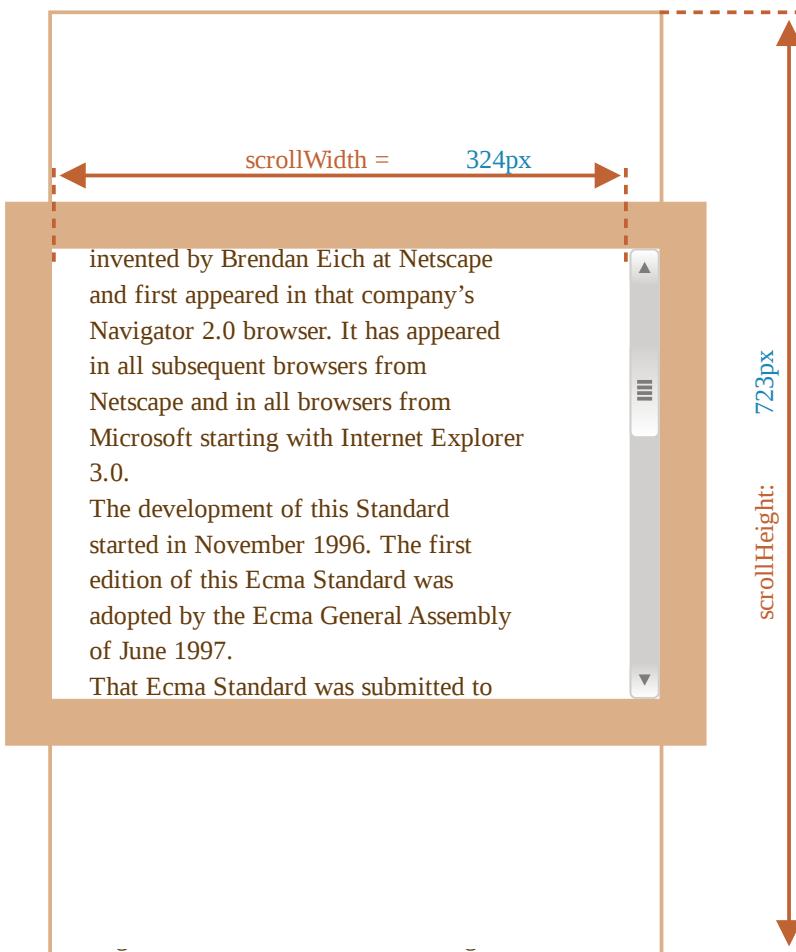
Если нет внутренних отступов `padding`, то `clientWidth/Height` в точности равны размеру области содержимого внутри рамок за вычетом полосы прокрутки (если она есть).



Поэтому в тех случаях, когда мы точно знаем, что отступов нет, можно использовать `clientWidth/clientHeight` для получения размеров внутренней области содержимого.

scrollWidth/Height

Эти свойства – как `clientWidth/clientHeight`, но также включают в себя прокрученную (которую не видно) часть элемента.



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.

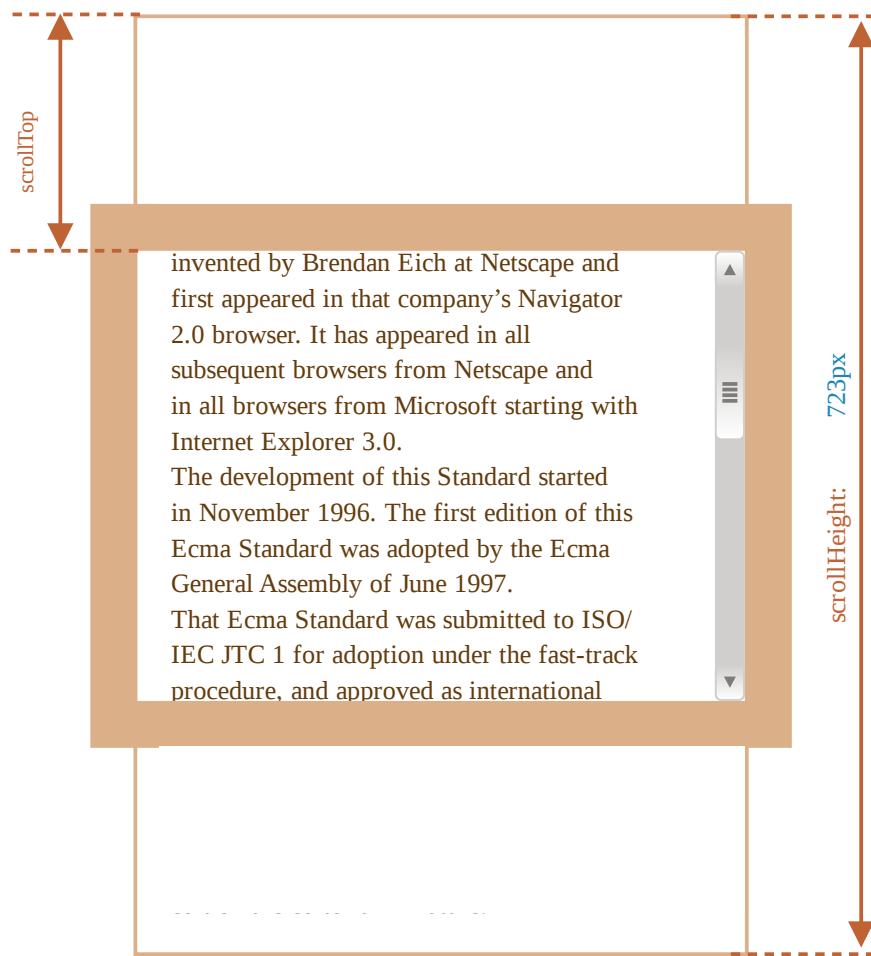
Таким кодом:

```
// распахнуть элемент на всю высоту
element.style.height = `${element.scrollHeight}px`;
```

scrollLeft/scrollTop

Свойства `scrollLeft/scrollTop` – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Следующая иллюстрация показывает значения `scrollHeight` и `scrollTop` для блока с вертикальной прокруткой.



Другими словами, свойство `scrollTop` – это «сколько уже прокручено вверх».

i Свойства `scrollLeft`/`scrollTop` можно изменять

В отличие от большинства свойств, которые доступны только для чтения, значения `scrollLeft`/`scrollTop` можно изменять, и браузер выполнит прокрутку элемента.

Установка значения `scrollTop` на `0` или на большое значение, такое как `1e9`, прокрутит элемент в самый верх/низ соответственно.

Не стоит брать `width/height` из CSS

Мы рассмотрели метрики, которые есть у DOM-элементов, и которые можно использовать для получения различных высот, ширин и прочих расстояний.

Но как мы знаем из главы [Стили и классы](#), CSS-высоту и ширину можно извлечь, используя `getComputedStyle`.

Так почему бы не получать, к примеру, ширину элемента при помощи `getComputedStyle`, вот так?

```
let elem = document.body;

alert( getComputedStyle(elem).width ); // показывает CSS-ширину elem
```

Почему мы должны использовать свойства-метрики вместо этого? На то есть две причины:

1. Во-первых, CSS-свойства `width/height` зависят от другого свойства – `box-sizing`, которое определяет, «что такое», собственно, эти CSS-ширина и высота. Получается, что изменение `box-sizing`, к примеру, для более удобной вёрстки, сломает такой JavaScript.
2. Во-вторых, CSS свойства `width/height` могут быть равны `auto`, например, для инлайнового элемента:

```
<span id="elem">Привет!</span>

<script>
  alert( getComputedStyle(elem).width ); // auto
</script>
```

Конечно, с точки зрения CSS `width:auto` – совершенно нормально, но нам-то в JavaScript нужен конкретный размер в `px`, который мы могли бы использовать для вычислений. Получается, что в данном случае ширина из CSS вообще бесполезна.

Есть и ещё одна причина: полоса прокрутки. Бывает, без полосы прокрутки код работает прекрасно, но стоит ей появиться, как начинают проявляться баги. Так происходит потому, что полоса прокрутки «отъедает» место от области внутреннего содержимого в некоторых браузерах. Таким образом, реальная ширина содержимого *меньше* CSS-ширины. Как раз это и учитывают свойства `clientWidth/clientHeight`.

...Но с `getComputedStyle(elem).width` ситуация иная. Некоторые браузеры (например, Chrome) возвращают реальную внутреннюю ширину с вычетом ширины полосы прокрутки, а некоторые (например, Firefox) – именно CSS-свойство (игнорируя полосу прокрутки). Эти кросбраузерные отличия – ещё один повод не использовать `getComputedStyle`, а использовать свойства-метрики.

Обратите внимание: описанные различия касаются только чтения свойства `getComputedStyle(...).width` из JavaScript, визуальное отображение корректно в обоих случаях.

Итого

У элементов есть следующие геометрические свойства (метрики):

- `offsetParent` – ближайший CSS-позиционированный родитель или ближайший `td`, `th`, `table`, `body`.
- `offsetLeft/offsetTop` – позиция в пикселях верхнего левого угла относительно `offsetParent`.
- `offsetWidth/offsetHeight` – «внешняя» ширина/высота элемента, включая рамки.
- `clientLeft/clientTop` – расстояние от верхнего левого внешнего угла до внутреннего. Для операционных систем с ориентацией слева-направо эти свойства равны ширинам левой/верхней рамки. Если язык ОС таков, что ориентация справа налево, так что вертикальная полоса прокрутки находится не справа, а слева, то `clientLeft` включает в своё значение её ширину.
- `clientWidth/clientHeight` – ширина/высота содержимого вместе с внутренними отступами `padding`, но без полосы прокрутки.
- `scrollWidth/scrollHeight` – ширины/высота содержимого, аналогично `clientWidth/Height`, но учитывают прокрученную, невидимую область элемента.
- `scrollLeft/scrollTop` – ширина/высота прокрученной сверху части элемента, считается от верхнего левого угла.

Все свойства доступны только для чтения, кроме `scrollLeft/scrollTop`, изменение которых заставляет браузер прокручивать элемент.

✓ Задачи

Найти размер прокрутки снизу

важность: 5

Свойство `elem.scrollTop` содержит размер прокрученной области при отсчёте сверху. А как подсчитать размер прокрутки снизу (назовём его `scrollBottom`)?

Напишите соответствующее выражение для произвольного элемента `elem`.

P.S. Проверьте: если прокрутки нет вообще или элемент полностью прокручен – оно должно давать `0`.

[К решению](#)

Узнать ширину полосы прокрутки

важность: 3

Напишите код, который возвращает ширину стандартной полосы прокрутки.

Для Windows она обычно колеблется от `12px` до `20px`. Если браузер не выделяет место под полосу прокрутки (так тоже бывает, она может быть прозрачной над текстом), тогда значение может быть `0px`.

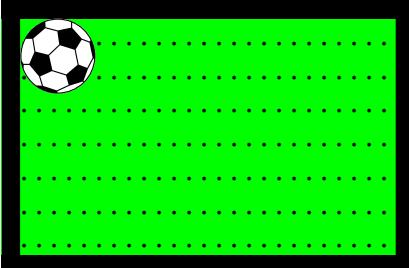
P.S. Ваш код должен работать в любом HTML-документе, независимо от его содержимого.

[К решению](#)

Поместите мяч в центр поля

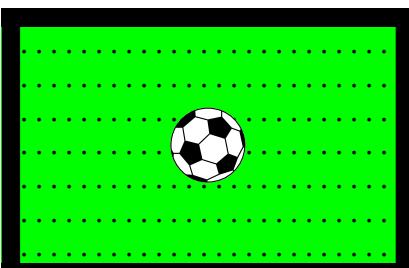
важность: 5

Исходный документ выглядит так:



Каковы координаты центра поля?

Вычислите их и используйте, чтобы поместить мяч в центр поля:



- Элемент должен позиционироваться за счёт JavaScript, а не CSS.
- Код должен работать с любым размером мяча (`10`, `20`, `30` пикселей) и любым размером поля без привязки к исходным значениям.

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](#)

В чём отличие CSS-свойств width и clientWidth

важность: 5

В чём отличие между `getComputedStyle(elem).width` и `elem.clientWidth`?

Укажите хотя бы 3 различия, лучше – больше.

[К решению](#)

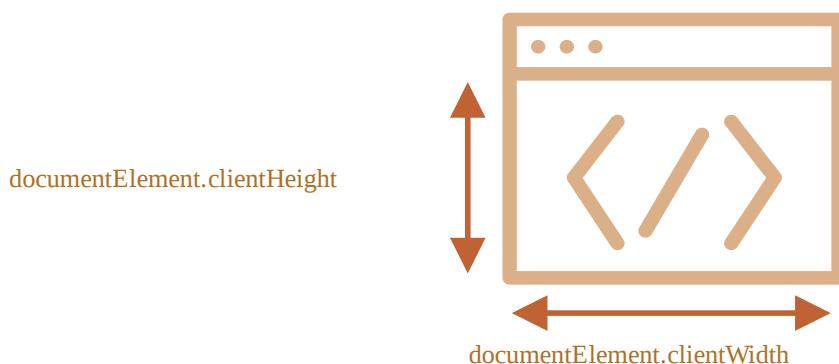
Размеры и прокрутка окна

Как узнать ширину и высоту окна браузера? Как получить полную ширину и высоту документа, включая прокрученную часть? Как прокрутить страницу с помощью JavaScript?

Для большинства таких запросов мы можем использовать корневой элемент документа `document.documentElement`, который соответствует тегу `<html>`. Однако есть дополнительные методы и особенности, которые необходимо учитывать.

Ширина/высота окна

Чтобы получить ширину/высоту окна, можно взять свойства `clientWidth/clientHeight` из `document.documentElement`:



Не `window.innerWidth/Height`

Браузеры также поддерживают свойства `window.innerWidth/innerHeight`. Вроде бы, похоже на то, что нам нужно. Почему же не использовать их?

Если есть полоса прокрутки, и она занимает какое-то место, то свойства `clientWidth/clientHeight` указывают на ширину/высоту документа без неё (за её вычетом). Иными словами, они возвращают высоту/ширину видимой части документа, доступной для содержимого.

А `window.innerWidth/innerHeight` включают в себя полосу прокрутки.

Если полоса прокрутки занимает некоторое место, то эти две строки выведут разные значения:

```
alert( window.innerWidth ); // полная ширина окна  
alert( document.documentElement.clientWidth ); // ширина окна за вычетом полосы пр
```

В большинстве случаев нам нужна доступная ширина окна: для рисования или позиционирования. Полоса прокрутки «отъедает» её часть. Поэтому следует использовать `documentElement.clientHeight/Width`.

DOCTYPE – это важно

Обратите внимание, что геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет `<!DOCTYPE HTML>`. Возможны странности.

В современном HTML мы всегда должны указывать `DOCTYPE`.

Ширина/высота документа

Теоретически, т.к. корневым элементом документа является `documentElement`, и он включает в себя всё содержимое, мы можем получить полный размер документа как `documentElement.scrollWidth/scrollHeight`.

Но именно на этом элементе, для страницы в целом, эти свойства работают не так, как предполагается. В Chrome/Safari/Opera, если нет прокрутки, то `documentElement.scrollHeight` может быть даже меньше, чем `documentElement.clientHeight`! С точки зрения элемента это невозможная ситуация.

Чтобы надёжно получить полную высоту документа, нам следует взять максимальное из этих свойств:

```
let scrollHeight = Math.max(  
    document.body.scrollHeight, document.documentElement.scrollHeight,  
    document.body.offsetHeight, document.documentElement.offsetHeight,  
    document.body.clientHeight, document.documentElement.clientHeight  
);  
  
alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);
```

Почему? Лучше не спрашивайте. Эти несоответствия идут с древних времён. Глубокой логики здесь нет.

Получение текущей прокрутки

Обычные элементы хранят текущее состояние прокрутки в `elem.scrollTop/scrollLeft`.

Что же со страницей? В большинстве браузеров мы можем обратиться к `documentElement.scrollTop/Top`, за исключением основанных на старом WebKit (Safari), где есть ошибка ([5991 ↗](#)), и там нужно использовать `document.body` вместо `document.documentElement`.

К счастью, нам совсем не обязательно запоминать эти особенности, потому что текущую прокрутку можно прочитать из свойств `window.pageYOffset/pageXOffset`:

```
alert('Текущая прокрутка сверху: ' + window.pageYOffset);  
alert('Текущая прокрутка слева: ' + window.pageXOffset);
```

Эти свойства доступны только для чтения.

Прокрутка: `scrollTo`, `scrollBy`, `scrollIntoView`



Важно:

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен.

Например, если мы попытаемся прокрутить страницу из скрипта, подключенного в `<head>`, это не сработает.

Обычные элементы можно прокручивать, изменения `scrollTop/scrollLeft`.

Мы можем сделать то же самое для страницы в целом, используя `document.documentElement.scrollTop/Left` (кроме основанных на

старом WebKit (Safari), где, как сказано выше,
`document.body.scrollTop/Left`).

Есть и другие способы, в которых подобных несовместимостей нет: специальные методы `window.scrollBy(x, y)` и `window.scrollTo(pageX, pageY)` .

- Метод `scrollBy(x, y)` прокручивает страницу относительно её текущего положения. Например, `scrollBy(0, 10)` прокручивает страницу на 10px вниз.
- Метод `scrollTo(pageX, pageY)` прокручивает страницу на абсолютные координаты `(pageX, pageY)` . То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить `scrollLeft scrollTop` . Для прокрутки в самое начало мы можем использовать `scrollTo(0, 0)` .

Эти методы одинаково работают для всех браузеров.

scrollIntoView

Для полноты картины давайте рассмотрим ещё один метод:
[elem.scrollIntoView\(`top`\) ↗](#) .

Вызов `elem.scrollIntoView(top)` прокручивает страницу, чтобы `elem` оказался вверху. У него есть один аргумент:

- если `top=true` (по умолчанию), то страница будет прокручена, чтобы `elem` появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.
- если `top=false` , то страница будет прокручена, чтобы `elem` появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Запретить прокрутку

Иногда нам нужно 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 установить
`document.body.style.overflow = "hidden"` .

Аналогичным образом мы можем «заморозить» прокрутку для других элементов, а не только для `document.body` .

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 выглядит немного странно, но это можно обойти, если сравнить `clientWidth` до и после остановки, и если `clientWidth` увеличится (значит полоса прокрутки исчезла), то добавить `padding` в `document.body` вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

Итого

Размеры:

- Ширина/высота видимой части документа (ширина/высота области содержимого): `document.documentElement.clientWidth/Height`
- Ширина/высота всего документа со всей прокручиваемой областью страницы:

```
let scrollHeight = Math.max(  
    document.body.scrollHeight, document.documentElement.scrollHeight,  
    document.body.offsetHeight, document.documentElement.offsetHeight,  
    document.body.clientHeight, document.documentElement.clientHeight  
)
```

Прокрутка:

- Прокрутку окна можно получить так: `window.pageYOffset/pageXOffset`.
- Изменить текущую прокрутку:
 - `window.scrollTo(pageX, pageY)` – абсолютные координаты,
 - `window.scrollBy(x, y)` – прокрутка относительно текущего места,
 - `elem.scrollIntoView(true)` – прокрутить страницу так, чтобы сделать `elem` видимым (выровнять относительно верхней/нижней части окна).

Координаты

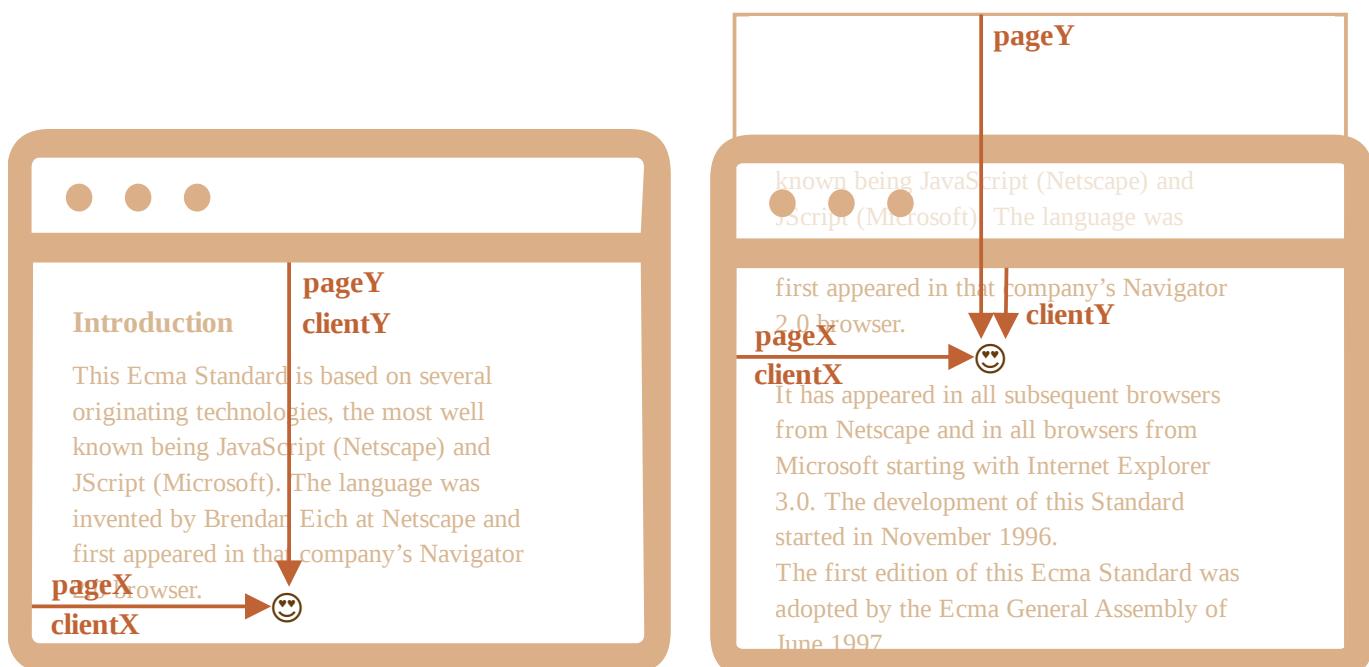
Чтобы передвигать элементы по экрану, нам следует познакомиться с системами координат.

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

- Относительно окна браузера** – как `position:fixed`, отсчёт идёт от верхнего левого угла окна.
 - мы будем обозначать эти координаты как `clientX/clientY`, причина выбора таких имён будет ясна позже, когда мы изучим свойства событий.
- Относительно документа** – как `position:absolute` на уровне документа, отсчёт идёт от верхнего левого угла документа.
 - мы будем обозначать эти координаты как `pageX/pageY`.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На приведённой картинке взята точка в документе и показаны её координаты до прокрутки (слева) и после (справа):



При прокрутке документа:

- `pageY` – координата точки относительно документа осталась без изменений, так как отсчёт по-прежнему ведётся от верхней границы документа (сейчас она прокручена наверх).
- `clientY` – координата точки относительно окна изменилась (стрелка на рисунке стала короче), так как точка стала ближе к верхней границе окна.

Координаты относительно окна: `getBoundingClientRect`

Метод `elem.getBoundingClientRect()` возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент `elem`, в виде объекта встроенного класса `DOMRect`.

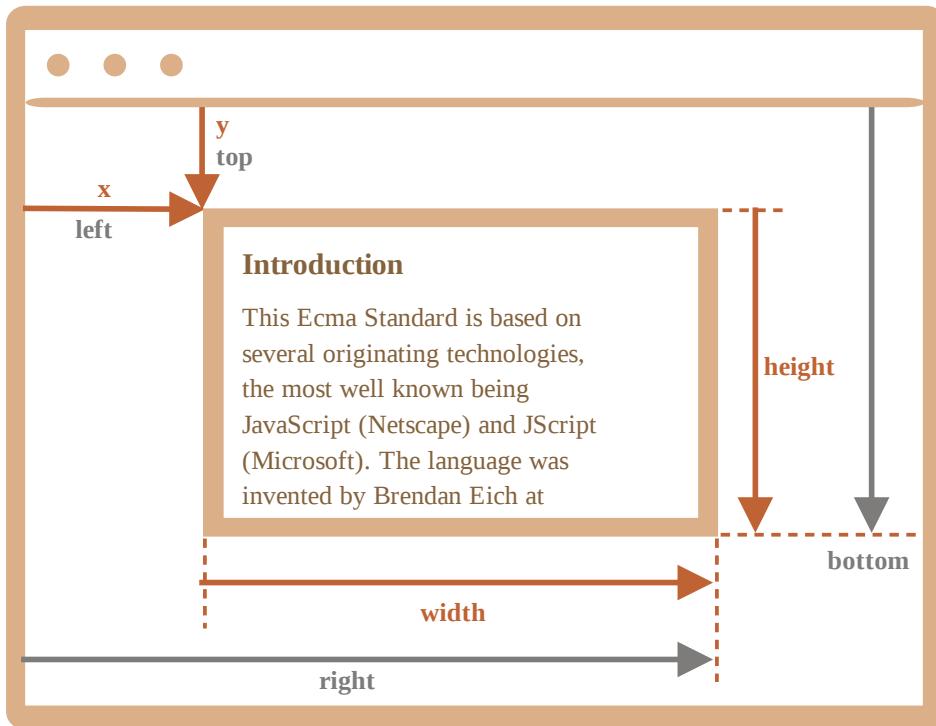
Основные свойства объекта типа `DOMRect`:

- `x/y` – X/Y-координаты начала прямоугольника относительно окна,
- `width/height` – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

- `top/bottom` – Y-координата верхней/нижней границы прямоугольника,
- `left/right` – X-координата левой/правой границы прямоугольника.

Вот картинка с результатами вызова `elem.getBoundingClientRect()`:



Как вы видите, `x/y` и `width/height` уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе:

- `left = x`
- `top = y`
- `right = x + width`
- `bottom = y + height`

Заметим:

- Координаты могут считаться с десятичной частью, например `10.5`. Это нормально, ведь браузер использует дроби в своих внутренних вычислениях. Мы не обязаны округлять значения при установке `style.left/top`.
- Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент `elem` ушёл вверх за пределы окна, то вызов `elem.getBoundingClientRect().top` вернёт отрицательное значение.

i Зачем вообще нужны зависимые свойства? Для чего существуют `top/left`, если есть `x/y`?

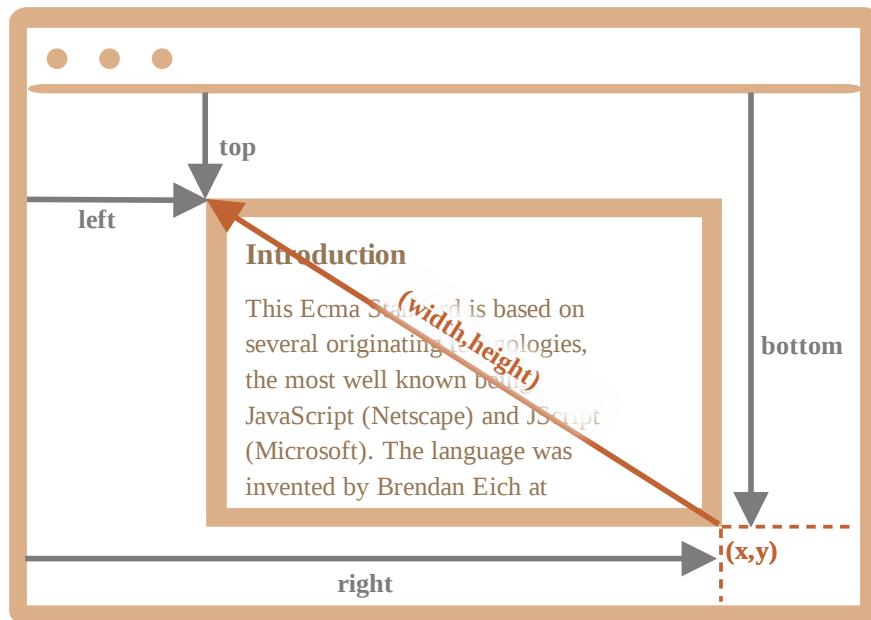
С математической точки зрения, прямоугольник однозначно задаётся начальной точкой (x, y) и вектором направления $(width, height)$.

Так что дополнительные зависимые свойства существуют лишь для удобства.

Что же касается `top/left`, то они на самом деле не всегда равны `x/y`. Технически, значения `width/height` могут быть отрицательными. Это позволяет задать «направленный» прямоугольник, например, для выделения мышью с отмеченным началом и концом.

То есть, отрицательные значения `width/height` означают, что прямоугольник «растет» влево-вверх из правого угла.

Вот прямоугольник с отрицательными `width` и `height` (например, `width=-200, height=-100`):



Как вы видите, свойства `left/top` при этом не равны `x/y`.

Впрочем, на практике результат вызова `elem.getBoundingClientRect()` всегда возвращает положительные значения для ширины/высоты. Здесь мы упомянули отрицательные `width/height` лишь для того, чтобы вы поняли, зачем существуют эти с виду дублирующие свойства.

Internet Explorer и Edge: не поддерживают x/y

Internet Explorer и Edge не поддерживают свойства `x/y` по историческим причинам.

Таким образом, мы можем либо сделать полифил (добавив соответствующие геттеры в `DomRect.prototype`), либо использовать `top/left`, так как это всегда одно и то же при положительных `width/height`, в частности – в результате вызова `elem.getBoundingClientRect()`.

Координаты right/bottom отличаются от одноимённых CSS-свойств

Есть очевидное сходство между координатами относительно окна и CSS `position:fixed`.

Но в CSS свойство `right` означает расстояние от правого края, и свойство `bottom` означает расстояние от нижнего края окна браузера.

Если взглянуть на картинку выше, то видно, что в JavaScript это не так. Все координаты в контексте окна считаются от верхнего левого угла, включая `right/bottom`.

elementFromPoint(x, y)

Вызов `document.elementFromPoint(x, y)` возвращает самый глубоко вложенный элемент в окне, находящийся по координатам `(x, y)`.

Синтаксис:

```
let elem = document.elementFromPoint(x, y);
```

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента, который сейчас в центре окна браузера:

```
let centerX = document.documentElement.clientWidth / 2;
let centerY = document.documentElement.clientHeight / 2;

let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";
alert(elem.tagName);
```

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

 Для координат за пределами окна метод `elementFromPoint` возвращает `null`

Метод `document.elementFromPoint(x, y)` работает, только если координаты (x, y) относятся к видимой части содержимого окна.

Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается `null`.

Вот типичная ошибка, которая может произойти, если в коде нет соответствующей проверки:

```
let elem = document.elementFromPoint(x, y);
// если координаты ведут за пределы окна, то elem = null
elem.style.background = '>'; // Ошибка!
```

Применение для `fixed` позиционирования

Чаще всего нам нужны координаты для позиционирования чего-либо.

Чтобы показать что-то около нужного элемента, мы можем вызвать `getBoundingClientRect`, чтобы получить его координаты элемента, а затем использовать CSS-свойство `position` вместе с `left/top` (или `right/bottom`).

Например, функция `createMessageUnder(elem, html)` ниже показывает сообщение под элементом `elem`:

```
let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {
    // создаём элемент, который будет содержать сообщение
    let message = document.createElement('div');
    // для стилей лучше было бы использовать css-класс здесь
    message.style.cssText = "position:fixed; color: red";

    // устанавливаем координаты элементу, не забываем про "px"!
    let coords = elem.getBoundingClientRect();

    message.style.left = coords.left + "px";
    message.style.top = coords.bottom + "px";

    message.innerHTML = html;

    return message;
}
```

```
// Использование:  
// добавим сообщение на страницу на 5 секунд  
let message = createMessageUnder(elem, 'Hello, world!');  
document.body.append(message);  
setTimeout(() => message.remove(), 5000);
```

Код можно изменить, чтобы показывать сообщение слева, справа, снизу, применять к нему CSS-анимации и так далее. Это просто, так как в нашем распоряжении имеются все координаты и размеры элемента.

Но обратите внимание на одну важную деталь: при прокрутке страницы сообщение уплывает от кнопки.

Причина весьма очевидна: сообщение позиционируется с помощью `position:fixed`, поэтому оно остаётся всегда на том же самом месте в окне при прокрутке страницы.

Чтобы изменить это, нам нужно использовать другую систему координат, где сообщение позиционировалось бы относительно документа, и свойство `position:absolute`.

Координаты относительно документа

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна.

В CSS координаты относительно окна браузера соответствуют свойству `position:fixed`, а координаты относительно документа – свойству `position:absolute` на самом верхнем уровне вложенности.

Мы можем воспользоваться свойствами `position:absolute` и `top/left`, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но мы можем написать его сами.

Две системы координат связаны следующими формулами:

- `pageY = clientY` + высота вертикально прокрученной части документа.
- `pageX = clientX` + ширина горизонтально прокрученной части документа.

Функция `getCoords(elem)` берёт координаты в контексте окна с помощью `elem.getBoundingClientRect()` и добавляет к ним значение соответствующей прокрутки:

```
// получаем координаты элемента в контексте документа
```

```
function getCoords(elem) {
  let box = elem.getBoundingClientRect();

  return {
    top: box.top + window.pageYOffset,
    right: box.right + window.pageXOffset,
    bottom: box.bottom + window.pageYOffset,
    left: box.left + window.pageXOffset
  };
}
```

Если бы в примере выше мы использовали её вместе с `position: absolute`, то при прокрутке сообщение оставалось бы рядом с элементом.

Модифицированная функция `createMessageUnder`:

```
function createMessageUnder(elem, html) {
  let message = document.createElement('div');
  message.style.cssText = "position: absolute; color: red";

  let coords = getCoords(elem);

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;

  return message;
}
```

Итого

Любая точка на странице имеет координаты:

1. Относительно окна браузера – `elem.getBoundingClientRect()`.
2. Относительно документа – `elem.getBoundingClientRect()` плюс текущая прокрутка страницы.

Координаты в контексте окна подходят для использования с `position: fixed`, а координаты относительно документа – для использования с `position: absolute`.

Каждая из систем координат имеет свои преимущества и недостатки. Иногда будет лучше применить одну, а иногда – другую, как это и происходит с позиционированием в CSS, где мы выбираем между `absolute` и `fixed`.

✓ Задачи

Найдите координаты точек относительно окна браузера

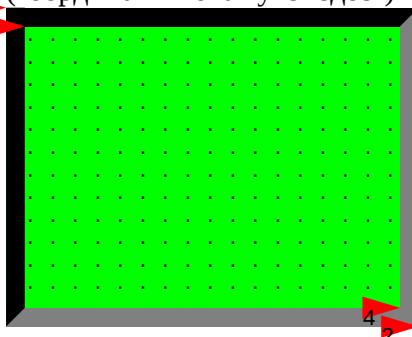
важность: 5

В ифрейме ниже располагается документ с зелёным «полем».

Используйте JavaScript, чтобы найти координаты углов, обозначенных стрелками.

В документе уже реализована функциональность, когда при клике на любом месте показываются соответствующие координаты.

Кликните в любом месте для получения координат в контексте окна.
Это для тестирования, чтобы проверить результат, который вы получили с помощью JavaScript.
(координаты покажутся здесь)



1
3
5
4
2

Ваш код должен при помощи DOM получить четыре пары координат:

1. верхний левый, внешний угол (это просто).
2. нижний правый, внешний угол (тоже просто).
3. верхний левый, внутренний угол (чуть сложнее).
4. нижний правый, внутренний угол (есть несколько способов, выберите один).

Координаты, вычисленные вами, должны совпадать с теми, которые возвращаются по клику мыши.

P.S. Код должен работать, если у элемента другие размеры или есть рамка, без привязки к конкретным числам.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

[Покажите заметку рядом с элементом](#)

важность: 5

Создайте функцию `positionAt(anchor, position, elem)`, которая позиционирует элемент `elem` в зависимости от значения свойства `position` рядом с элементом `anchor`.

Аргумент `position` – строка с одним из 3 значений:

- `"top"` – расположить `elem` прямо над `anchor`
- `"right"` – расположить `elem` непосредственно справа от `anchor`
- `"bottom"` – расположить `elem` прямо под `anchor`

Она используется внутри функции `showNote(anchor, position, html)`, которая уже есть в исходном коде задачи. Она создаёт и показывает элемент-«заметку» с текстом `html` на заданной позиции `position` рядом с элементом `anchor`.

Демо заметки:

The screenshot shows a note-taking interface with three examples of note positioning:

- note above**: A note positioned above the text "Teacher: Why are you late?".
- note at the right**: A note positioned to the right of the text "Teacher: Why are you late?".
- note below**: A note positioned below the text "Teacher: Why are you late?".

The main text area contains placeholder text: "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt 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](#)

Покажите заметку около элемента (абсолютное позиционирование)

важность: 5

Измените код решения [предыдущего задания](#) так, чтобы элемент заметки использовал свойство `position: absolute` вместо `position: fixed`.

Это предотвратит расхождение элементов при прокрутке страницы.

Используйте решение предыдущего задания для начала. Чтобы проверить решение в условиях с прокруткой, добавьте стиль элементу `<body style="height: 2000px">`.

[К решению](#)

Расположите заметку внутри элемента (абсолютное позиционирование)

важность: 5

Усовершенствуйте решение предыдущего задания [Покажите заметку около элемента \(абсолютное позиционирование\)](#): научите функцию `positionAt(anchor, position, elem)` вставлять `elem` внутрь `anchor`.

Новые значения для аргумента `position`:

- `top-out`, `right-out`, `bottom-out` – работают так же, как раньше, они вставляют `elem` сверху/справа/снизу `anchor`.
- `top-in`, `right-in`, `bottom-in` – вставляют `elem` внутрь `anchor`: приклеивают его к верхнему/правому/нижнему краю.

Например:

```
// показывает заметку поверх цитаты
positionAt(blockquote, "top-out", note);

// показывает заметку внутри цитаты вблизи верхнего края элемента
positionAt(blockquote, "top-in", note);
```

Результат:

Rem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidentum 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.

note top-out

note top-in

note right-out

Teacher: Why are you late?

Student: There was a man who lost a hundred dollar bill.

Teacher: That's nice. Were you helping him look for it?

Student: No. I was standing on it.

note bottom-in

Rem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidentum 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.

Для начала возьмите решение задания [Покажите заметку около элемента](#) (абсолютное позиционирование).

[К решению](#)

Введение в события

Введение в браузерные события, их свойства и обработку.

Введение в браузерные события

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

- `click` – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- `contextmenu` – происходит, когда кликнули на элемент правой кнопкой мыши.
- `mouseover / mouseout` – когда мышь наводится на / покидает элемент.
- `mousedown / mouseup` – когда нажали / отжали кнопку мыши на элементе.
- `mousemove` – при движении мыши.

События на элементах управления:

- `submit` – пользователь отправил форму `<form>`.

- `focus` – пользователь фокусируется на элементе, например нажимает на `<input>`.

Клавиатурные события:

- `keydown` и `keyup` – когда пользователь нажимает / отпускает клавишу.

События документа:

- `DOMContentLoaded` – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

- `transitionend` – когда CSS-анимация завершена.

Существует множество других событий. Мы подробно разберём их в последующих главах.

Обработчики событий

Событию можно назначить **обработчик**, то есть функцию, которая сработает, как только событие произошло.

Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Есть несколько способов назначить событию обработчик. Сейчас мы их рассмотрим, начиная с самого простого.

Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется `on<событие>`.

Например, чтобы назначить обработчик события `click` на элементе `input`, можно использовать атрибут `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. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, вот так:

`onclick="alert("Click!")"`, код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и

вызвать её там.

Следующий пример по клику запускает функцию `countRabbits()`:

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

Считать кроликов!

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому `ONCLICK` будет работать так же, как `onClick` и `onCLICK`... Но, как правило, атрибуты пишут в нижнем регистре: `onclick`.

Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента `on<событие>`.

К примеру, `elem.onclick`:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```

Нажми меня!

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство.

Этот способ, по сути, аналогичен предыдущему.

Обработчик всегда хранится в свойстве DOM-объекта, 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">
```

```
Кнопка
```

2. HTML + JS:

```
<input type="button" id="button" value="Кнопка">
<script>
  button.onclick = function() {
    alert('Клик!');
  };
</script>
```

```
Кнопка
```

Так как у элемента DOM может быть только одно свойство с именем `onclick`, то назначить более одного обработчика так нельзя.

В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

```
<input type="button" id="elem" onclick="alert('Было!')" value="Нажми меня">
<script>
  elem.onclick = function() { // перезапишет существующий обработчик
    alert('Станет'); // выведется только это
  };
</script>
```

```
Нажми меня
```

Кстати, обработчиком можно назначить и уже существующую функцию:

```
function sayThanks() {
  alert('Спасибо!');
}

elem.onclick = sayThanks;
```

Убрать обработчик можно назначением `elem.onclick = null`.

Доступ к элементу через `this`

Внутри обработчика события `this` ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже `button` выводит своё содержимое, используя `this.innerHTML`:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

Нажми меня

Частые ошибки

Если вы только начинаете работать с событиями, обратите внимание на следующие моменты.

Функция должна быть присвоена как `sayThanks`, а не `sayThanks()`.

```
// правильно  
button.onclick = sayThanks;  
  
// неправильно  
button.onclick = sayThanks();
```

Если добавить скобки, то `sayThanks()` – это уже вызов функции, результат которого (равный `undefined`, так как функция ничего не возвращает) будет присвоен `onclick`. Так что это не будет работать.

...А вот в разметке, в отличие от свойства, скобки нужны:

```
<input type="button" id="button" onclick="sayThanks()">
```

Это различие просто объяснить. При создании обработчика браузером из атрибута, он автоматически создаёт функцию с телом из значения атрибута: `sayThanks()`.

Так что разметка генерирует такое свойство:

```
button.onclick = function() {  
    sayThanks(); // содержимое атрибута  
};
```

Используйте именно функции, а не строки.

Назначение обработчика строкой `elem.onclick = "alert(1)"` также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не используйте `setAttribute` для обработчиков.

Такой вызов работать не будет:

```
// при нажатии на body будут ошибки,  
// атрибуты всегда строки, и функция станет строкой  
document.body.setAttribute('onclick', function() { alert(1) });
```

Регистр DOM-свойства имеет значение.

Используйте `elem.onclick`, а не `elem.ONCLICK`, потому что DOM-свойства чувствительны к регистру.

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение.

Мы хотим назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

```
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 `addEventListener` и `removeEventListener`. Они свободны от указанного недостатка.

Синтаксис добавления обработчика:

```
element.addEventListener(event, handler, [options]);
```

event

Имя события, например `"click"`.

handler

Ссылка на функцию-обработчик.

options

Дополнительный объект со свойствами:

- `once` : если `true` , тогда обработчик будет автоматически удалён после выполнения.
- `capture` : фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе [Всплытие и погружение](#). Так исторически сложилось, что `options` может быть `false/true` , это то же самое, что `{capture: false/true}` .
- `passive` : если `true` , то указывает, что обработчик никогда не вызовет `preventDefault()` , подробнее об этом будет рассказано в главе [Действия браузера по умолчанию](#).

Для удаления обработчика следует использовать `removeEventListener` :

```
element.removeEventListener(event, handler, [options]);
```

Удаление требует именно ту же функцию

Для удаления нужно передать именно ту функцию-обработчик которая была назначена.

Вот так не сработает:

```
elem.addEventListener( "click" , () => alert('Спасибо!'));
// ....
elem.removeEventListener( "click", () => alert('Спасибо!'));
```

Обработчик не будет удалён, т.к. 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 обработчик не сохранить где-либо, мы не сможем её удалить. Нет метода, который позволяет получить из элемента 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 = () => 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.

⚠️ Обработчики некоторых событий можно назначать только через `addEventListener`

Существуют события, которые нельзя назначить через DOM-свойство, но можно через `addEventListener`.

Например, таково событие `DOMContentLoaded`, которое срабатывает, когда завершена загрузка и построение DOM документа.

```
document.onDOMContentLoaded = function() {
  alert("DOM построен"); // не будет работать
};
```

```
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM построен"); // а вот так сработает
});
```

Так что `addEventListener` более универсален. Хотя заметим, что таких событий меньшинство, это скорее исключение, чем правило.

Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее.

Когда происходит событие, браузер создаёт объект *события*, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

Пример ниже демонстрирует получение координат мыши из 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("Координаты: " + event.clientX + ":" + event.clientY);
}
</script>
```

Некоторые свойства объекта `event`:

`event.type`

Тип события, в данном случае `"click"`.

`event.currentTarget`

Элемент, на котором сработал обработчик. Значение – обычно такое же, как и у `this`, но если обработчик является функцией-стрелкой или при помощи `bind` привязан другой объект в качестве `this`, то мы можем получить элемент из `event.currentTarget`.

`event.clientX / event.clientY`

Координаты курсора в момент клика относительно окна, для событий мыши.

Есть также и ряд других свойств, в зависимости от типа событий, которые мы разберём в дальнейших главах.

Объект события доступен и в HTML

При назначении обработчика в HTML, тоже можно использовать объект `event`, вот так:

```
<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"`, а тело взято из атрибута.

Объект-обработчик: `handleEvent`

Мы можем назначить обработчиком не только функцию, но и объект при помощи `addEventListener`. В этом случае, когда происходит событие, вызывается метод объекта `handleEvent`.

К примеру:

```
<button id="elem">Нажми меня</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " на " + event.currentTarget);
    }
  });
</script>
```

Как видим, если `addEventListener` получает объект в качестве обработчика, он вызывает `object.handleEvent(event)`, когда происходит событие.

Мы также можем использовать класс для этого:

```
<button id="elem">Нажми меня</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
```

```

        elem.innerHTML = "Нажата кнопка мыши";
        break;
    case 'mouseup':
        elem.innerHTML += "...и отжата.";
        break;
    }
}
}

let menu = new Menu();
elem.addEventListener('mousedown', menu);
elem.addEventListener('mouseup', menu);
</script>

```

Здесь один и тот же объект обрабатывает оба события. Обратите внимание, мы должны явно назначить оба обработчика через `addEventListener`. Тогда объект `menu` будет получать события `mousedown` и `mouseup`, но не другие (не назначенные) типы событий.

Метод `handleEvent` не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий, вот так:

```

<button id="elem">Нажми меня</button>

<script>
    class Menu {
        handleEvent(event) {
            // mousedown -> onMousedown
            let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
            this[method](event);
        }

        onMousedown() {
            elem.innerHTML = "Кнопка мыши нажата";
        }

        onMouseup() {
            elem.innerHTML += "...и отжата.";
        }
    }

    let menu = new Menu();
    elem.addEventListener('mousedown', menu);
    elem.addEventListener('mouseup', menu);
</script>

```

Теперь обработка событий разделена по методам, что упрощает поддержку кода.

Итого

Есть три способа назначения обработчиков событий:

1. Атрибут HTML: `onclick="..."`.
2. DOM-свойство: `elem.onclick = function`.
3. Специальные методы: `elem.addEventListener(event, handler[, phase])` для добавления, `removeEventListener` для удаления.

HTML-атрибуты используются редко потому, что JavaScript в HTML-теге выглядит немного странно. К тому же много кода там не напишешь.

DOM-свойства вполне можно использовать, но мы не можем назначить больше одного обработчика на один тип события. Во многих случаях с этим ограничением можно мириться.

Последний способ самый гибкий, однако нужно писать больше всего кода. Есть несколько типов событий, которые работают только через него, к примеру `transitionend` и `DOMContentLoaded`. Также `addEventListener` поддерживает объекты в качестве обработчиков событий. В этом случае вызывается метод объекта `handleEvent`.

Не важно, как вы назначаете обработчик – он получает объект события первым аргументом. Этот объект содержит подробности о том, что произошло.

Мы изучим больше о событиях и их типах в следующих главах.

✓ Задачи

Скрыть элемент по нажатию кнопки

важность: 5

Добавьте JavaScript к кнопке `button`, чтобы при нажатии элемент `<div id="text">` исчезал.

Демо:

Нажмите, чтобы спрятать текст

Текст

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Спрятать себя

важность: 5

Создайте кнопку, которая будет скрывать себя по нажатию.

[К решению](#)

Какой обработчик запустится?

важность: 5

В переменной `button` находится кнопка. Изначально на ней нет обработчиков.

Который из обработчиков запустится? Что будет выведено при клике после выполнения кода?

```
button.addEventListener("click", () => alert("1"));

button.removeEventListener("click", () => alert("1"));

button.onclick = () => alert(2);
```

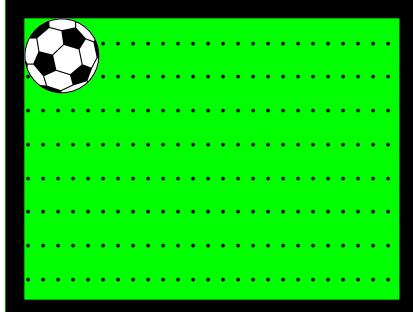
[К решению](#)

Передвигните мяч по полю

важность: 5

Пусть мяч перемещается при клике на поле, туда, куда был клик, вот так:

Нажмите на поле для перемещения мяча.



Требования:

- Центр мяча должен совпадать с местом нажатия мыши (если это возможно без пересечения краёв поля);
- CSS-анимация желательна, но не обязательна;
- Мяч ни в коем случае не должен пересекать границы поля;

- При прокрутке страницы ничего не должно ломаться;

Заметки:

- Код должен уметь работать с различными размерами мяча и поля, не привязываться к каким-либо фиксированным значениям.
- Используйте свойства `event.clientX/event.clientY` для определения координат мыши при клике.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Создать раскрывающееся меню

важность: 5

Создать меню, которое по нажатию открывается либо закрывается:

► Сладости (нажми меня)!

P.S. HTML/CSS исходного документа можно и нужно менять.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Добавить кнопку закрытия

важность: 5

Есть список сообщений.

При помощи JavaScript для каждого сообщения добавьте в верхний правый угол кнопку закрытия.

Результат должен выглядеть, как показано здесь:

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 (лат. *Equus asinus asinus*), или ишак, — одомашненный подвид дикого осла (*Equus asinus*), сыгравший важную историческую роль в развитии хозяйства и культуры человека и по-прежнему широко в хозяйстве многих развивающихся стран.

Кошка

[x]

Кошка, или домашняя кошка (лат. *Felis silvestris catus*), — домашнее животное, одно из наиболее популярных(наряду с собакой) «животных-компаньонов». Являясь одиночным охотником на грызунов и других мелких животных, кошка — социальное животное, использующее для общения широкий диапазон звуковых сигналов.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Карусель

важность: 4

Создайте «Карусель» — ленту изображений, которую можно листать влево-вправо нажатием на стрелочки.



В дальнейшем к ней можно будет добавить анимацию, динамическую подгрузку и другие возможности.

P.S. В этой задаче разработка структуры HTML/CSS составляет 90% решения.

[Открыть песочницу для задачи.](#) ↗

Всплытие и погружение

Давайте начнём с примера.

Этот обработчик для `<div>` сработает, если вы кликните по любому из вложенных тегов, будь то `` или `<code>`:

```
<div onclick="alert('Обработчик!')">
  <em>Если вы кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>
</div>
```

Если вы кликните на EM, сработает обработчик на DIV

Вам не кажется это странным? Почему же сработал обработчик на `<div>`, если клик произошёл на ``?

Всплытие

Принцип всплытия очень простой.

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

Например, есть 3 вложенных элемента `FORM` > `DIV` > `P` с обработчиком на каждом:

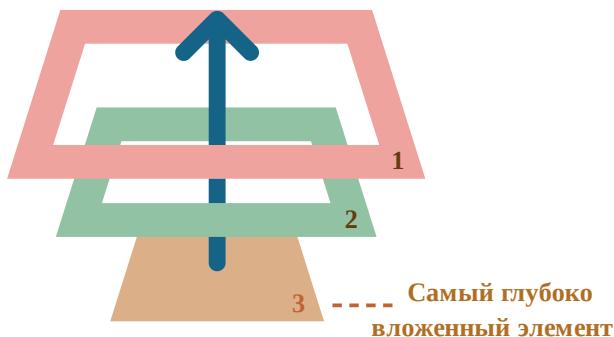
```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```



Клик по внутреннему `<p>` вызовет обработчик `onclick`:

1. Сначала на самом `<p>`.
2. Потом на внешнем `<div>`.
3. Затем на внешнем `<form>`.
4. И так далее вверх по цепочке до самого `document`.



Поэтому если кликнуть на `<p>`, то мы увидим три оповещения: `p → div → form`.

Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде.

⚠ **Почти все события всплывают.**

Ключевое слово в этой фразе – «почти».

Например, событие `focus` не всплывает. В дальнейшем мы увидим и другие примеры. Однако, стоит понимать, что это скорее исключение, чем правило, всё-таки большинство событий всплывают.

event.target

Всегда можно узнать, на каком конкретно элементе произошло событие.

Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через `event.target`.

Отличия от `this` (`= event.currentTarget`):

- `event.target` – это «целевой» 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 внутри обработчика `form.onclick`:

- `this (= event.currentTarget)` всегда будет элемент `<form>`, так как обработчик сработал на ней.
- `event.target` будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

Попробуйте сами:

<https://plnkr.co/edit/i3Q6mJoVe8HyXwdx?p=preview>

Возможна и ситуация, когда `event.target` и `this` – один и тот же элемент, например, если клик был непосредственно на самом элементе `<form>`, а не на его подэлементе.

Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет вспыльвать до элемента `<html>`, а затем до объекта `document`, а иногда даже до `window`, вызывая все обработчики на своём пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод `event.stopPropagation()`.

Например, здесь при клике на кнопку `<button>` обработчик `body.onclick` не сработает:

```
<body onclick="alert(`сюда всплытие 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, `event.stopPropagation()` препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует 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 через `event.stopPropagation()` имеет свои подводные камни, которые со временем могут стать проблемами.

Например:

1. Мы делаем вложенное меню. Каждое подменю обрабатывает клики на своих элементах и делает для них `stopPropagation`, чтобы не срабатывало внешнее меню.
2. Позже мы решили отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще у нас кликают люди. Некоторые системы аналитики так делают. Обычно используют `document.addEventListener('click' ...)`, чтобы отлавливать все клики.
3. Наша аналитика не будет работать над областью, где клики прекращаются `stopPropagation`. Увы, получилась «мёртвая зона».

Зачастую нет никакой необходимости прекращать всплытие. Задача, которая, казалось бы, требует этого, может быть решена иначе. Например, с помощью создания своего уникального события, о том, как это делать, мы поговорим позже. Также мы можем записывать какую-то служебную информацию в объект `event` в одном обработчике, а читать в другом, таким образом мы можем сообщить обработчикам на родительских элементах информацию о том, что событие уже было как-то обработано.

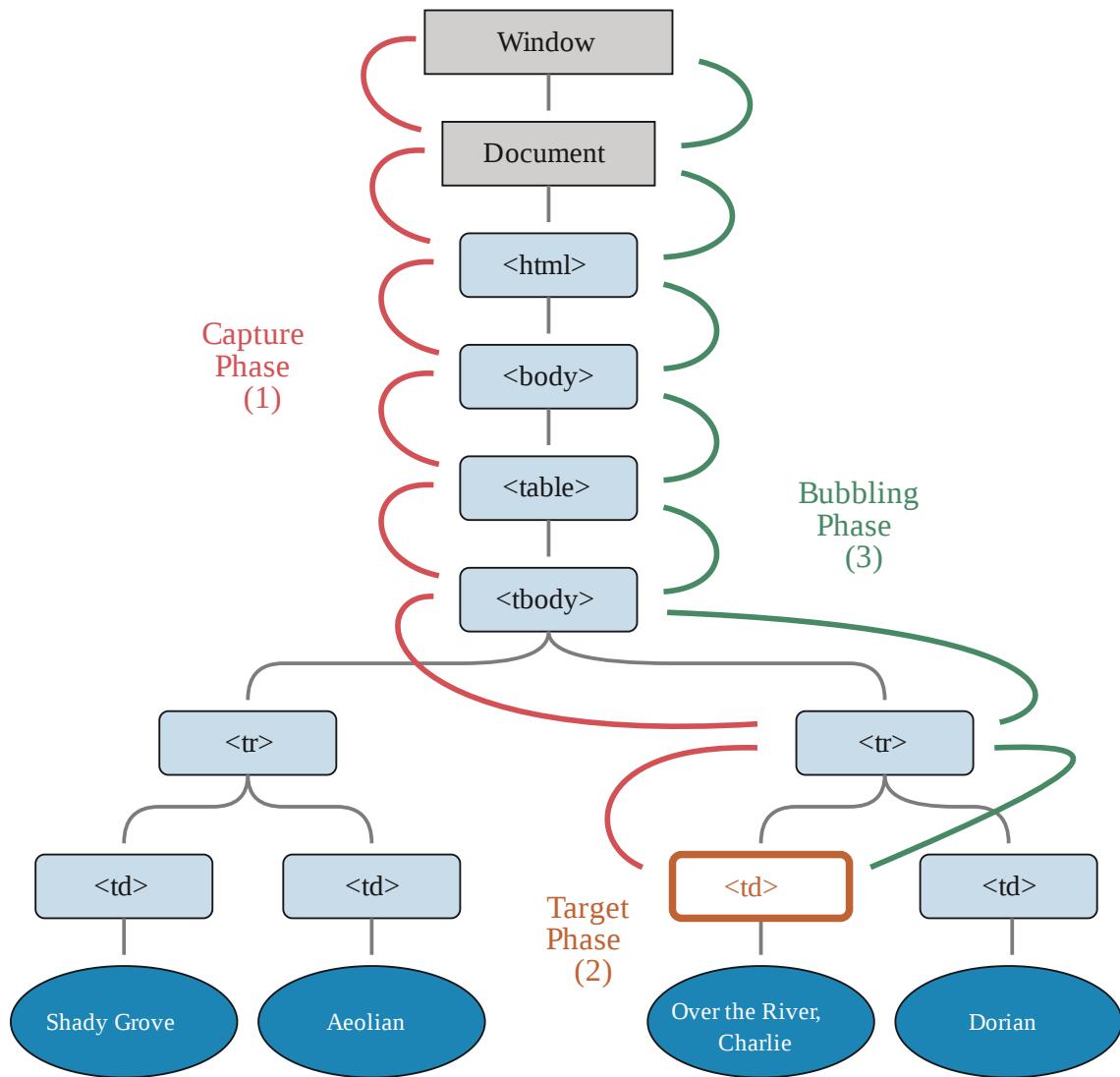
Погружение

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт DOM Events [описывает](#) 3 фазы прохода события:

1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Картина из спецификации демонстрирует, как это работает при клике по ячейке `<td>`, расположенной внутри таблицы:



То есть при клике на `<td>` событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт наверх (всплытие), вызывая по пути обработчики.

Ранее мы говорили только о всплытии, потому что другие стадии, как правило, не используются и проходят незаметно для нас.

Обработчики, добавленные через `on<event>`-свойство или через HTML-атрибуты, или через `addEventListener(event, handler)` с двумя

аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах.

Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент `capture` вот так:

```
elem.addEventListener(..., {capture: true})  
// или просто "true", как сокращение для {capture: true}  
elem.addEventListener(..., true)
```

Существуют два варианта значений опции `capture`:

- Если аргумент `false` (по умолчанию), то событие будет поймано при всплытии.
- Если аргумент `true`, то событие будет перехвачено при погружении.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение.

Давайте посмотрим и всплытие и погружение в действии:

```
<style>  
body * {  
    margin: 10px;  
    border: 1px solid blue;  
}  
</style>  
  
<form>FORM  
  <div>DIV  
    <p>P</p>  
  </div>  
</form>  
  
<script>  
  for(let elem of document.querySelectorAll('*')) {  
    elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);  
    elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));  
  }  
</script>
```

FORM

DIV

P

Здесь обработчики навешиваются на *каждый* элемент в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события.

Если вы кликните по `<p>`, то последовательность следующая:

1. `HTML` → `BODY` → `FORM` → `DIV` (фаза погружения, первый обработчик)
2. `P` (фаза цели, срабатывают обработчики, установленные и на погружение и на всплытие, так что выводится два раза)
3. `DIV` → `FORM` → `BODY` → `HTML` (фаза всплытия, второй обработчик)

Существует свойство `event.eventPhase`, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, мы обычно и так знаем об этом в обработчике.

i Чтобы убрать обработчик `removeEventListener`, нужна та же фаза

Если мы добавили обработчик вот так `addEventListener(..., true)`, то мы должны передать то же значение аргумента `capture` в `removeEventListener(..., true)`, когда снимаем обработчик.

i На каждой фазе разные обработчики на одном элементе срабатывают в порядке назначения

Если у нас несколько обработчиков одного события, назначенных `addEventListener` на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

```
elem.addEventListener("click", e => alert(1)); // всегда сработает перед следующим
elem.addEventListener("click", e => alert(2));
```

Итого

При наступлении события – самый глубоко вложенный элемент, на котором оно произошло, помечается как «целевой» (`event.target`).

- Затем событие сначаладвигается вниз от корня документа к `event.target`, по пути вызывая обработчики, поставленные через `addEventListener(..., true)`, где `true` – это сокращение для `{capture: true}`.
- Далее обработчики вызываются на целевом элементе.
- Далее событиедвигается от `event.target` вверх к корню документа, по пути вызывая обработчики, поставленные через `on<event>` и

`addEventListener` без третьего аргумента или с третьим аргументом равным `false`.

Каждый обработчик имеет доступ к свойствам события `event`:

- `event.target` – самый глубокий элемент, на котором произошло событие.
- `event.currentTarget` (`= this`) – элемент, на котором в данный момент сработал обработчик (тот, на котором «висит» конкретный обработчик)
- `event.eventPhase` – на какой фазе он сработал (погружение=1, фаза цели=2, 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, обычно события обрабатываются во время всплытия. И в этом есть логика.

В реальном мире, когда происходит чрезвычайная ситуация, местные службы реагируют первыми. Они знают лучше всех местность, в которой это произошло, и другие детали. Вышестоящие инстанции подключаются уже после этого и при необходимости.

Тоже самое справедливо для обработчиков событий. Код, который «навесил» обработчик на конкретный элемент, знает максимум деталей об элементе и его предназначении. Например, обработчик на определённом `<td>` скорее всего подходит только для этого конкретного `<td>`, он знает все о нём, поэтому он должен отработать первым. 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.

Всплытие и погружение являются основой для «делегирования событий» – очень мощного приёма обработки событий. Его мы изучим в следующей главе.

Делегирование событий

Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – **делегирование**.

Идея 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.

Рассмотрим пример – [диаграмму Ба-Гуа](#). Это таблица, отражающая древнюю китайскую философию.

Вот она:

Квадрат Bagua: Направление, Элемент, Цвет, Значение		
Северо-Запад Металл Серебро Старейшины	Север Вода Синий Перемены	Северо-Восток Земля Жёлтый Направление
Запад Металл Золото Молодость	Центр Всё Пурпурный Гармония	Восток Дерево Синий Будущее
Юго-Запад Земля Коричневый Спокойствие	Юг Огонь Оранжевый Слава	Юго-Восток Дерево Зелёный Роман

Её HTML (схематично):

```
<table>
  <tr>
    <th colspan="3">Квадрат <em>Bagua</em>: Направление, Элемент, Цвет, Значение</th>
  </tr>
  <tr>
    <td>...<strong>Северо-Запад</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>...ещё 2 строки такого же вида...</tr>
  <tr>...ещё 2 строки такого же вида...</tr>
</table>
```

В этой таблице всего 9 ячеек, но могло бы быть и 99, и даже 9999, не важно.

Наша задача – реализовать подсветку ячейки `<td>` при клике.

Вместо того, чтобы назначать обработчик `onclick` для каждой ячейки `<td>` (их может быть очень много) – мы повесим «единий» обработчик на элемент `<table>`.

Он будет использовать `event.target`, чтобы получить элемент, на котором произошло событие, и подсветить его.

Код будет таким:

```
let selectedTd;
```

```



```

Такому коду нет разницы, сколько ячеек в таблице. Мы можем добавлять, удалять `<td>` из таблицы динамически в любое время, и подсветка будет стablyно работать.

Однако, у текущей версии кода есть недостаток.

Клик может быть не на теге `<td>`, а внутри него.

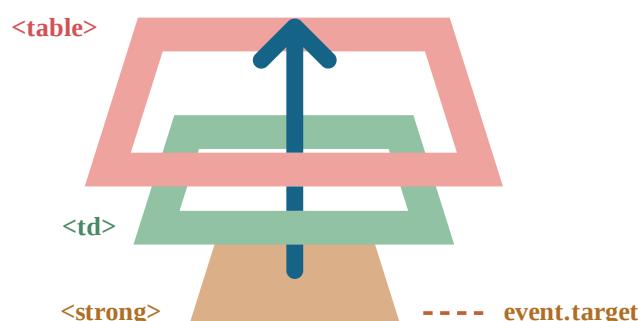
В нашем случае, если взглянуть на HTML-код таблицы внимательно, видно, что ячейка `<td>` содержит вложенные теги, например ``:

```

<td>
  <strong>Северо-Запад</strong>
  ...
</td>

```

Естественно, если клик произойдёт на элементе ``, то он станет значением `event.target`.



Внутри обработчика `table.onclick` мы должны по `event.target` разобраться, был клик внутри `<td>` или нет.

Вот улучшенный код:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};
```

Разберём пример:

1. Метод `elem.closest(selector)` возвращает ближайшего предка, соответствующего селектору. В данном случае нам нужен `<td>`, находящийся выше по дереву от исходного элемента.
2. Если `event.target` не содержится внутри элемента `<td>`, то вызов вернёт `null`, и ничего не произойдёт.
3. Если таблицы вложенные, `event.target` может содержать элемент `<td>`, находящийся вне текущей таблицы. В таких случаях мы должны проверить, действительно ли это `<td>` нашей таблицы.
4. И если это так, то подсвечиваем его.

В итоге мы получили короткий код подсветки, быстрый и эффективный, которому совершенно не важно, сколько всего в таблице `<td>`.

Применение делегирования: действия в разметке

Есть и другие применения делегирования.

Например, нам нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами `save`, `load`, `search`... Как их состыковать?

Первое, что может прийти в голову – это найти каждую кнопку и назначить ей свой обработчик среди методов объекта. Но существует более элегантное решение. Мы можем добавить один обработчик для всего меню и атрибуты `data-action` для каждой кнопки в соответствии с методами, которые они вызывают:

```
<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>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('сохраняю');
    }

    load() {
      alert('загружаю');
    }

    search() {
      alert('ищу');
    }

    onClick(event) {
      let action = event.target.dataset.action;
      if (action) {
        this[action]();
      }
    }
  }

  new Menu(menu);
</script>

```

Обратите внимание, что метод `this.onClick` в строке, отмеченной звёздочкой `(*)`, привязывается к контексту текущего объекта `this`. Это важно, т.к. иначе `this` внутри него будет ссылаться на DOM-элемент (`elem`), а не на объект `Menu`, и `this[action]` будет не тем, что нам нужно.

Так что же даёт нам здесь делегирование?

- Не нужно писать код, чтобы присвоить обработчик каждой кнопке. Достаточно просто создать один метод и поместить его в разметку.
- Структура HTML становится по-настоящему гибкой. Мы можем добавлять/удалять кнопки в любое время.

Мы также можем использовать классы `.action-save`, `.action-load`, но подход с использованием атрибутов `data-action` является более семантическим. Их можно использовать и для стилизации в правилах CSS.

Приём проектирования «поведение»

Делегирование событий можно использовать для добавления элементам «поведения» (`behavior`), декларативно задавая хитрые обработчики установкой специальных HTML-атрибутов и классов.

Приём проектирования «поведение» состоит из двух частей:

1. Элементу ставится пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования ставится обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут, производит соответствующее действие.

Поведение: «Счётчик»

Например, здесь HTML-атрибут `data-counter` добавляет кнопкам поведение: «увеличить значение при клике»:

```
Счётчик: <input type="button" value="1" data-counter>
Ещё счётчик: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {
    if (event.target.dataset.counter != undefined) { // если есть атрибут...
      event.target.value++;
    }
  });
</script>
```

Счётчик: 1 Ещё счётчик: 2

Если нажать на кнопку – значение увеличится. Конечно, нам важны не счётчики, а общий подход, который здесь продемонстрирован.

Элементов с атрибутом `data-counter` может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования мы фактически добавили новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

⚠️ Всегда используйте метод `addEventListener` для обработчиков на уровне документа

Когда мы устанавливаем обработчик событий на объект `document`, мы всегда должны использовать метод `addEventListener`, а не `document.on<событие>`, т.к. в случае последнего могут возникать конфликты: новые обработчики будут перезаписывать уже существующие.

Для реального проекта совершенно нормально иметь много обработчиков на элементе `document`, установленных из разных частей кода.

Поведение: «Переключатель» (Toggler)

Ещё один пример поведения. 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.addEventListener('click', function(event) {  
        let id = event.target.dataset.toggleId;  
        if (!id) return;  
  
        let elem = document.getElementById(id);  
  
        elem.hidden = !elem.hidden;  
    });  
</script>
```

Показать форму подписки

Ещё раз подчеркнём, 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 так себя вести. Просто используем поведение. Обработчики на уровне документа сделают это возможным для элемента в любом месте страницы.

Мы можем комбинировать несколько вариантов поведения на одном элементе.

Шаблон «поведение» может служить альтернативой для фрагментов JS-кода в вёрстке.

Итого

Делегирование событий – это здорово! Пожалуй, это один из самых полезных приёмов для работы с DOM.

Он часто используется, если есть много элементов, обработка которых очень схожа, но не только для этого.

Алгоритм:

1. Вешаем обработчик на контейнер.
2. В обработчике проверяем исходный элемент `event.target`.
3. Если событие произошло внутри нужного нам элемента, то обрабатываем его.

Зачем использовать:

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

Конечно, у делегирования событий есть свои ограничения:

- Во-первых, событие должно всплывать. Некоторые события этого не делают. Также, низкоуровневые обработчики не должны вызывать `event.stopPropagation()`.
- Во-вторых, делегирование создаёт 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, что её даже не стоит принимать во внимание.

✓ Задачи

Спрятать сообщения с помощью делегирования

важность: 5

Дан список сообщений с кнопками для удаления [x]. Заставьте кнопки работать.

В результате должно работать вот так:

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.

Кошка

[x]

Кошка, или домашняя кошка (лат. *Félis silvétris cátus*), — домашнее животное, одно из наиболее популярных (наряду с собакой) "животных-компаньонов". С точки зрения научной систематики, домашняя кошка — млекопитающее семейства кошачьих отряда хищных. Ранее домашнюю кошку нередко рассматривали как отдельный биологический вид.

P.S. Используйте делегирование событий. Должен быть лишь один обработчик на элементе-контейнере для всего.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Раскрывающееся дерево

важность: 5

Создайте дерево, которое по клику на заголовок скрывает-показывает потомков:

- Животные
 - Млекопитающие
 - Коровы
 - Ослы
 - Собаки
 - Тигры
 - Другие
 - Змеи
 - Птицы
 - Ящерицы
- Рыбы
 - Аквариумные
 - Гуппи
 - Скалярии
 - Морские
 - Морская форель

Требования:

- Использовать только один обработчик событий (применить делегирование)
- Клик вне текста заголовка (на пустом месте) ничего делать не должен.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Сортируемая таблица

важность: 4

Сделать таблицу сортируемой: при клике на элемент `<th>` строки таблицы должны сортироваться по соответствующему столбцу.

Каждый элемент `<th>` имеет атрибут `data-type`:

```
<table id="grid">
  <thead>
    <tr>
      <th data-type="number">Возраст</th>
      <th data-type="string">Имя</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>5</td>
      <td>Вася</td>
    </tr>
    <tr>
      <td>10</td>
      <td>Петя</td>
    </tr>
    ...
  </tbody>
</table>
```

```
</tbody>  
</table>
```

В примере выше первый столбец содержит числа, а второй – строки. Функция сортировки должна это учитывать, ведь числа сортируются иначе, чем строки.

Сортировка должна поддерживать только типы "string" и "number".

Работающий пример:

Возраст	Имя
5	Вася
2	Петя
12	Женя
9	Маша
1	Илья

P.S. Таблица может быть большой, с любым числом строк и столбцов.

[Открыть песочницу для задачи.](#)

[К решению](#)

Поведение "подсказка"

важность: 5

Напишите JS-код, реализующий поведение «подсказка».

При наведении мыши на элемент с атрибутом `data-tooltip`, над ним должна показываться подсказка и скрываться при переходе на другой элемент.

Пример HTML с подсказками:

```
<button data-tooltip="эта 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

Короткая кнопка

Ещё кнопка

Прокрутите страницу, чтобы кнопки оказались у верхнего края, а затем проверьте, правильно ли выводятся подсказки.

В этой задаче мы полагаем, что во всех элементах с атрибутом `data-tooltip` – только текст. То есть, в них нет вложенных тегов (пока).

Детали оформления:

- Отступ от подсказки до элемента с `data-tooltip` должен быть `5px` по высоте.
- Подсказка должна быть, по возможности, посередине элемента.
- Подсказка не должна вылезать за границы экрана, в том числе если страница частично прокручена, если нельзя показать сверху – показывать снизу элемента.
- Текст подсказки брать из значения атрибута `data-tooltip`. Это может быть произвольный HTML.

Для решения вам понадобятся два события:

- `mouseover` срабатывает, когда указатель мыши заходит на элемент.
- `mouseout` срабатывает, когда указатель мыши уходит с элемента.

Примените делегирование событий: установите оба обработчика на элемент `document`, чтобы отслеживать «заход» и «уход» курсора на элементы с атрибутом `data-tooltip` и управлять подсказками с их же помощью.

После реализации поведения – люди, даже не знакомые с JavaScript смогут добавлять подсказки к элементам.

P.S. В один момент может быть показана только одна подсказка.

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 её на сервер.
- Зажатие кнопки мыши над текстом и её движение в таком состоянии – 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 `event`. Для отмены действия браузера существует стандартный метод `event.preventDefault()`.
- Если же обработчик назначен через `on<событие>` (не через `addEventListener`), то также можно вернуть `false` из обработчика.

В следующем примере при клике по ссылке переход не произойдёт:

```
<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 `on<событие>`.

В других случаях `return` не нужен, он никак не обрабатывается.

Пример: меню

Рассмотрим меню для сайта, например:

```
<ul id="menu" class="menu">  
  <li><a href="/html">HTML</a></li>  
  <li><a href="/javascript">JavaScript</a></li>
```

```
<li><a href="/css">CSS</a></li>
</ul>
```

Данный пример при помощи CSS может выглядеть так:

HTML

JavaScript

CSS

В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами `<a>`. В этом подходе есть некоторые преимущества, например:

- Некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если мы будем использовать `<button>` или ``, то данное сочетание работать не будет.
- Поисковые движки переходят по ссылкам `` при индексации.

Поэтому в разметке мы используем `<a>`. Но нам необходимо обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить.

Например, вот так:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert(href); // может быть подгрузка с сервера, генерация интерфейса и т.п.

  return false; // отменить действие браузера (переход по ссылке)
};
```

Если мы уберём `return false`, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из `href`. А это нам здесь не нужно, мы обрабатываем клик сами.

Кстати, использование здесь делегирования событий делает наше меню очень гибким. Мы можем добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

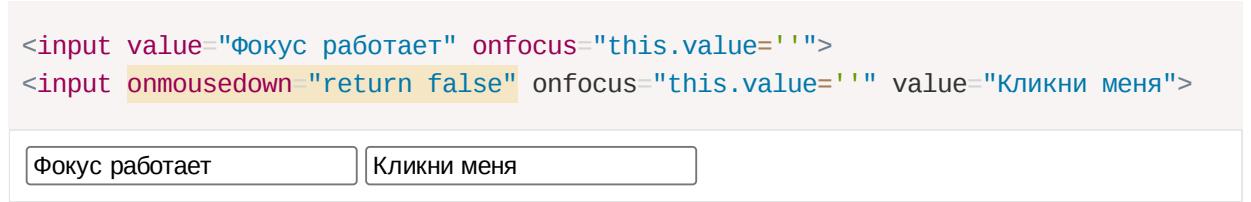
События, вытекающие из других

Некоторые события естественным образом вытекают друг из друга. Если мы отменим первое событие, то последующие не возникнут.

Например, событие `mousedown` для поля `<input>` приводит к фокусировке на нём и запускает событие `focus`. Если мы отменим событие `mousedown`, то фокусирования не произойдёт.

В следующем примере попробуйте нажать на первом `<input>` – происходит событие `focus`. Но если вы нажимаете по второму элементу, то события `focus` не будет.

```
<input value="Фокус работает" onfocus="this.value=''">
<input onmousedown="return false" onfocus="this.value=''" value="Кликни меня">
```



Это потому, что отменено стандартное действие `mousedown`. Впрочем, фокусировка на элементе всё ещё возможна, если мы будем использовать другой способ. Например, нажатием клавиши `Tab` можно перейти от первого поля ввода ко второму. Но только не через клик мышью на элемент, это больше не работает.

Опция «`passive`» для обработчика

Необязательная опция `passive: true` для `addEventListener` сигнализирует браузеру, что обработчик не собирается выполнять `preventDefault()`.

Почему это может быть полезно?

Есть некоторые события, как `touchmove` на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но мы можем отменить это действие, используя `preventDefault()` в обработчике.

Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если `preventDefault` не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция `passive: true` сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие.

Для некоторых браузеров (Firefox, Chrome) опция `passive` по умолчанию включена в `true` для таких событий, как `touchstart` и `touchmove`.

event.defaultPrevented

Свойство `event.defaultPrevented` установлено в `true`, если действие по умолчанию было предотвращено, и `false`, если нет.

Рассмотрим практическое применение этого свойства для улучшения архитектуры.

Помните, в главе [Всплытие и погружение](#) мы говорили о `event.stopPropagation()` и упоминали, что останавливать «всплытие» – плохо?

Иногда вместо этого мы можем использовать `event.defaultPrevented`, чтобы просигналить другим обработчикам, что событие обработано.

Давайте посмотрим практический пример.

По умолчанию браузер при событии `contextmenu` (клик правой кнопкой мыши) показывает контекстное меню со стандартными опциями. Мы можем отменить событие по умолчанию и показать своё меню, как здесь:

```
<button>Правый клик вызывает контекстное меню браузера</button>

<button oncontextmenu="alert('Рисуем наше меню'); return false">
    Правый клик вызывает наше контекстное меню
</button>
```

Правый клик вызывает контекстное меню браузера Правый клик вызывает наше контекстное меню

Теперь в дополнение к этому контекстному меню реализуем контекстное меню для всего документа.

При правом клике должно показываться ближайшее контекстное меню.

```
<p>Правый клик здесь вызывает контекстное меню документа</p>
<button id="elem">Правый клик здесь вызывает контекстное меню кнопки</button>

<script>
    elem.oncontextmenu = function(event) {
        event.preventDefault();
        alert("Контекстное меню кнопки");
    };

    document.oncontextmenu = function(event) {
        event.preventDefault();
    };
</script>
```

```
    alert("Контекстное меню документа");
};

</script>
```

Правый клик здесь вызывает контекстное меню документа

Правый клик здесь вызывает контекстное меню кнопки

Проблема заключается в том, что когда мы кликаем по элементу `elem`, то мы получаем два меню: контекстное меню для кнопки и (событие всплывает вверх) контекстное меню для документа.

Как это поправить? Одно из решений – это подумать: «Когда мы обрабатываем правый клик в обработчике на кнопке, остановим всплытие», и вызвать `event.stopPropagation()`:

```
<p>Правый клик вызывает меню документа</p>
<button id="elem">Правый клик вызывает меню кнопки (добавлен event.stopPropagation)</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    event.stopPropagation();
    alert("Контекстное меню кнопки");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Контекстное меню документа");
  };
</script>
```

Правый клик вызывает меню документа

Правый клик вызывает меню кнопки (добавлен event.stopPropagation)

Теперь контекстное меню для кнопки работает как задумано. Но цена слишком высока. Мы навсегда запретили доступ к информации о правых кликах для любого внешнего кода, включая счётчики, которые могли бы собирать статистику, и т.п. Это слегка неразумно.

Альтернативным решением было бы проверить в обработчике `document`, было ли отменено действие по умолчанию? Если да, тогда событие было обработано, и нам не нужно на него реагировать.

```
<p>Правый клик вызывает меню документа (добавлена проверка event.defaultPrevented)</p>
<button id="elem">Правый клик вызывает меню кнопки</button>

<script>
```

```
elem.oncontextmenu = function(event) {  
    event.preventDefault();  
    alert("Контекстное меню кнопки");  
};  
  
document.oncontextmenu = function(event) {  
    if (event.defaultPrevented) return;  
  
    event.preventDefault();  
    alert("Контекстное меню документа");  
};  
</script>
```

Правый клик вызывает меню документа (добавлена проверка `event.defaultPrevented`)

Правый клик вызывает меню кнопки

Сейчас всё работает правильно. Если у нас есть вложенные элементы и каждый из них имеет контекстное меню, то код также будет работать. Просто убедитесь, что проверяете `event.defaultPrevented` в каждом обработчике `contextmenu`.

ⓘ `event.stopPropagation()` и `event.preventDefault()`

Как мы можем видеть, `event.stopPropagation()` и `event.preventDefault()` (также известный как `return false`) – это две разные функции. Они никак не связаны друг с другом.

ⓘ Архитектура вложенных контекстных меню

Есть также несколько альтернативных путей, чтобы реализовать вложенные контекстные меню. Одним из них является единый глобальный объект с обработчиком `document.oncontextmenu` и методами, позволяющими хранить в нём другие обработчики.

Объект будет перехватывать любой клик правой кнопкой мыши, просматривать сохранённые обработчики и запускать соответствующий.

Но при этом каждый фрагмент кода, которому требуется контекстное меню, должен знать об этом объекте и использовать его вместо собственного обработчика `contextmenu`.

Итого

Действий браузера по умолчанию достаточно много:

- `mousedown` – начинает выделять текст (если двигать мышкой).

- `click` на `<input type="checkbox">` – ставит или убирает галочку в `input`.
- `submit` – при нажатии на `<input type="submit">` или при нажатии клавиши `Enter` в форме данные отправляются на сервер.
- `keydown` – при нажатии клавиши в поле ввода появляется символ.
- `contextmenu` – при правом клике показывается контекстное меню браузера.
- ...и многие другие...

Все эти действия можно отменить, если мы хотим обработать событие исключительно при помощи JavaScript.

Чтобы отменить действие браузера по умолчанию, используйте `event.preventDefault()` или `return false`. Второй метод работает, только если обработчик назначен через `on<событие>`.

Опция `passive: true` для `addEventListener` сообщает браузеру, что действие по умолчанию не будет отменено. Это очень полезно для некоторых событий на мобильных устройствах, таких как `touchstart` и `touchmove`, чтобы сообщить браузеру, что он не должен ожидать выполнения всех обработчиков, а ему следует сразу приступить к выполнению действия по умолчанию, например, к прокрутке.

Если событие по умолчанию отменено, то значение `event.defaultPrevented` становится `true`, иначе `false`.

Сохраняйте семантику, не злоупотребляйте

Технически, отменяя действия браузера по умолчанию и добавляя JavaScript, мы можем настроить поведение любого элемента. Например, мы можем заставить ссылку `<a>` работать как кнопку, а кнопку `<button>` вести себя как ссылка (перенаправлять на другой URL).

Но нам следует сохранять семантическое значение HTML элементов. Например, не кнопки, а тег `<a>` должен применяться для переходов по ссылкам.

Помимо того, что это «хорошо», это делает ваш HTML лучше с точки зрения доступности для людей с ограниченными возможностями и с особых устройств.

Также, если мы рассматриваем пример с тегом `<a>`, то обратите внимание: браузер предоставляет возможность открывать ссылки в новом окне (кликая правой кнопкой мыши или используя другие возможности). И пользователям это нравится. Но если мы заменим ссылку кнопкой и стилизуем её как ссылку, используя CSS, то специфичные функции браузера для тега `<a>` всё равно работать не будут.

✓ Задачи

Почему не работает return false?

важность: 3

Почему в коде ниже `return false` не работает?

```
<script>
  function handler() {
    alert( "...");
    return false;
  }
</script>

<a href="https://w3.org" onclick="handler()">браузер откроет w3.org</a>
```

[браузер откроет w3.org](#)

Браузер переходит по указанной ссылке, но нам этого не нужно.

Как поправить?

[К решению](#)

Поймайте переход по ссылке

важность: 5

Сделайте так, чтобы при клике на ссылки внутри элемента `id="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 насчёт того, чтобы прочитать [Википедию](#) или посетить [W3.org](#) и узнать о современных стандартах?

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 иметь вложенные теги, в том числе *внутри ссылок*, например, `<i>...</i>`.

Открыть песочницу для задачи. ↗

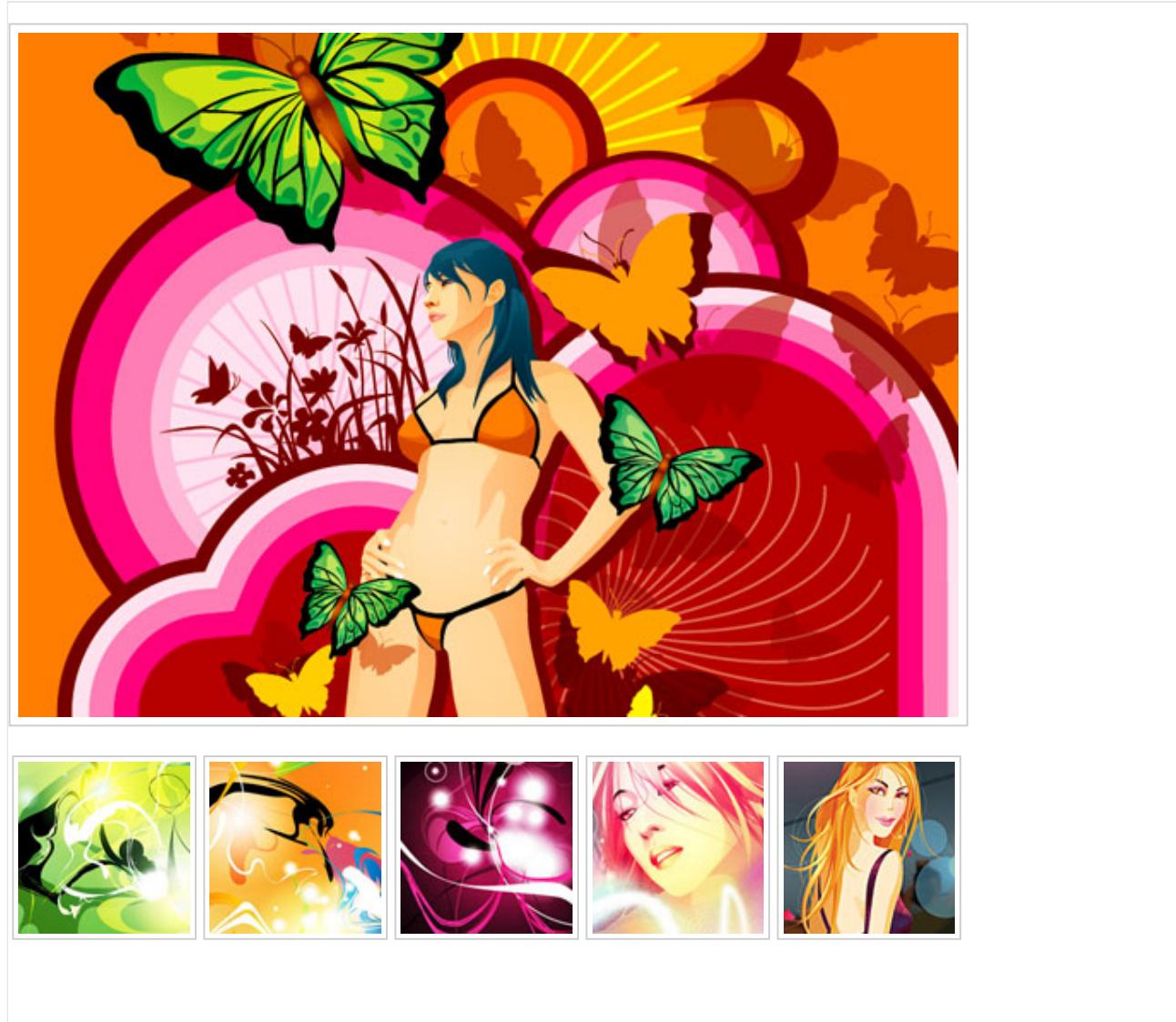
К решению

Галерея изображений

важность: 5

Создайте галерею изображений, в которой основное изображение изменяется при клике на уменьшенный вариант.

Например:



P.S. Используйте делегирование.

Открыть песочницу для задачи. ↗

К решению

Генерация пользовательских событий

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода.

Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент нашего меню, реализованного при помощи JavaScript, может генерировать события, относящиеся к этому меню: `open` (меню раскрыто), `select` (выбран пункт меню) и т.п. А другой код может слушать эти события и узнавать, что происходит с меню.

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как `click`, `mousedown` и другие. Это бывает полезно для автоматического тестирования.

Конструктор Event

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс [Event](#).

Событие встроенного класса `Event` можно создать так:

```
let event = new Event(type[, options]);
```

Где:

- `type` – тип события, строка, например `"click"` или же любой придуманный нами – `"my-event"`.
- `options` – объект с тремя необязательными свойствами:
 - `bubbles: true/false` – если `true`, тогда событие всплывает.
 - `cancelable: true/false` – если `true`, тогда можно отменить действие по умолчанию. Позже мы разберём, что это значит для пользовательских событий.
 - `composed: true/false` – если `true`, тогда событие будет всплывать наружу за пределы Shadow DOM. Позже мы разберём это в [разделе Веб-компоненты](#).

По умолчанию все три свойства установлены в `false`: `{bubbles: false, cancelable: false, composed: false}`.

Метод dispatchEvent

После того, как объект события создан, мы должны запустить его на элементе, вызвав метод `elem.dispatchEvent(event)`.

Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан флаг `bubbles`, то оно будет всплывать.

В примере ниже событие `click` инициируется JavaScript-кодом так, как будто кликнули по кнопке:

```
<button id="elem" onclick="alert('Клик!');>Автоклик</button>

<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

`event.isTrusted`

Можно легко отличить «настоящее» событие от сгенерированного кодом.

Свойство `event.isTrusted` принимает значение `true` для событий, порождаемых реальными действиями пользователя, и `false` для генерируемых кодом.

Пример всплытия

Мы можем создать всплывающее событие с именем `"hello"` и поймать его на `document`.

Всё, что нужно сделать – это установить флаг `bubbles` в `true`:

```
<h1 id="elem">Привет из кода!</h1>

<script>
  // ловим на document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Привет от " + event.target.tagName); // Привет от H1
  });

  // ...запуск события на элементе!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);

  // обработчик на document сработает и выведет сообщение.

</script>
```

Обратите внимание:

- Мы должны использовать `addEventListener` для наших собственных событий, т.к. `on<event>`-свойства существуют только для встроенных событий, то есть `document.onhello` не сработает.
- Мы обязаны передать флаг `bubbles: true`, иначе наше событие не будет всплывать.

Механизм всплытия идентичен как для встроенного события (`click`), так и для пользовательского события (`hello`). Также одинакова работа фаз всплытия и погружения.

MouseEvent, KeyboardEvent и другие

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации [UI Event](#) :

- `UIEvent`
- `FocusEvent`
- `MouseEvent`
- `WheelEvent`
- `KeyboardEvent`
- ...

Стоит использовать их вместо `new Event`, если мы хотим создавать такие события. К примеру, `new MouseEvent("click")`.

Специфический конструктор позволяет указать стандартные свойства для данного типа события.

Например, `clientX/clientY` для события мыши:

```
let event = new MouseEvent("click", {  
    bubbles: true,  
    cancelable: true,  
    clientX: 100,  
    clientY: 100  
});  
  
alert(event.clientX); // 100
```

Обратите внимание: этого нельзя было бы сделать с обычным конструктором `Event`.

Давайте проверим:

```
let event = new Event("click", {
    bubbles: true, // только свойства bubbles и cancelable
    cancelable: true, // работают в конструкторе Event
    clientX: 100,
    clientY: 100
});

alert(event.clientX); // undefined, неизвестное свойство проигнорировано!
```

Впрочем, использование конкретного конструктора не является обязательным, можно обойтись `Event`, а свойства записать в объект отдельно, после создания, вот так: `event.clientX=100`. 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, [MouseEvent ↗](#).

Пользовательские события

Для генерации событий совершенно новых типов, таких как `"hello"`, следует использовать конструктор `new CustomEvent`. Технически [CustomEvent ↗](#) абсолютно идентичен `Event` за исключением одной 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>
    // дополнительная информация приходит в обработчик вместе с событием
    elem.addEventListener("hello", function(event) {
        alert(event.detail.name);
    });

    elem.dispatchEvent(new CustomEvent("hello", {
        detail: { name: "Вася" }
    }));
</script>
```

Свойство `detail` может содержать любые данные. Надо сказать, что никто не мешает и в обычное `new Event` записать любые свойства. Но `CustomEvent` предоставляет специальное поле `detail` во избежание конфликтов с другими свойствами события.

Кроме того, класс события описывает, что это за событие, и если оно не браузерное, а пользовательское, то лучше использовать `CustomEvent`, чтобы явно об этом сказать.

`event.preventDefault()`

Для многих браузерных событий есть «действия по умолчанию», такие как переход по ссылке, выделение и т.п.

Для новых, пользовательских событий браузерных действий, конечно, нет, но код, который генерирует такое событие, может предусматривать какие-то свои действия после события.

Вызов `event.preventDefault()` является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия надо отменить.

Тогда вызов `elem.dispatchEvent(event)` возвратит `false`. И код, сгенерировавший событие, узнает, что продолжать не нужно.

Посмотрим практический пример – прячущегося кролика (могло бы быть скрывающееся меню или что-то ещё).

Ниже вы можете видеть кролика `#rabbit` и функцию `hide()`, которая при вызове генерирует на нём событие `"hide"`, уведомляя всех интересующихся, что кролик собирается спрятаться.

Любой обработчик может узнать об этом, подписавшись на событие `hide` через `rabbit.addEventListener('hide', ...)` и, при желании, отменить действие по умолчанию через `event.preventDefault()`. Тогда кролик не исчезнет:

```
<pre id="rabbit">
  |\_ /|
  \|_|/
  /_. .\
  =\_\_/_=_
  {>o<}
</pre>
<button onclick="hide()">Hide()</button>

<script>
  // hide() будет вызван при щелчке на кнопке
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true // без этого флага preventDefault не сработает
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('Действие отменено обработчиком');
    } else {

```

```
rabbit.hidden = true;
}

}

rabbit.addEventListener('hide', function(event) {
  if (confirm("Вызвать preventDefault?")) {
    event.preventDefault();
  }
});
</script>
```

```
\ \ /|  
 \_/_/  
 / . .\  
 =\ _Y_ /=  
 {>o<}
```

Hide()

Обратите внимание: событие должно содержать флаг `cancelable: true`. Иначе, вызов `event.preventDefault()` будет проигнорирован.

Вложенные события обрабатываются синхронно

Обычно события обрабатываются асинхронно. То есть, если браузер обрабатывает `onclick` и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка `onclick`.

Исключением является ситуация, когда событие инициировано из обработчика другого события.

Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад.

В примере ниже событие `menu-open` обрабатывается синхронно во время обработки `onclick`:

```
<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert("вложенное событие")
    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));

    alert(2);
  };
</script>
```

```
document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>
```

Меню (нажми меня)

Порядок вывода: 1 → вложенное событие → 2.

Обратите внимание, что вложенное событие `menu-open` успевает всплыть и запустить обработчик на `document`. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код (`onclick`).

Это справедливо не только для `dispatchEvent`, но и для других ситуаций. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Если нам это не подходит, то мы можем либо поместить `dispatchEvent` (или любой другой код, инициирующий события) в конец обработчика `onclick`, либо, если это неудобно, можно обернуть генерацию события в `setTimeout` с нулевой задержкой:

```
<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert(2)
    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>
```

Теперь `dispatchEvent` запускается асинхронно после исполнения текущего кода, включая `mouse.onclick`, поэтому обработчики полностью независимы.

Новый порядок вывода: 1 → 2 → вложенное событие.

Итого

Чтобы сгенерировать событие из кода, вначале надо создать объект события.

Базовый конструктор `Event(name, options)` принимает обязательное имя события и `options` – объект с двумя свойствами:

- `bubbles: true` чтобы событие всплывало.
- `cancelable: true` если мы хотим, чтобы `event.preventDefault()` работал.

Особые конструкторы встроенных событий `MouseEvent`, `KeyboardEvent` и другие принимают специфичные для каждого конкретного типа событий свойства. Например, `clientX` для событий мыши.

Для пользовательских событий стоит применять конструктор `CustomEvent`. У него есть дополнительная опция `detail`, с помощью которой можно передавать информацию в объекте события. После чего все обработчики смогут получить к ней доступ через `event.detail`.

Несмотря на техническую возможность генерировать встроенные браузерные события типа `click` или `keydown`, пользоваться ей стоит с большой осторожностью.

Весьма часто, когда разработчик хочет сгенерировать встроенное событие – это вызвано «кривой» архитектурой кода.

Как правило, генерация встроенных событий полезна в следующих случаях:

- 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.

Пользовательские события со своими именами часто создают для улучшения архитектуры, чтобы сообщить о том, что происходит внутри наших меню, слайдеров, каруселей и т.д.

Интерфейсные события

Здесь мы изучим основные события пользовательского интерфейса и как с ними работать.

Основы событий мыши

В этой главе мы более детально рассмотрим события мыши и их свойства.

Сразу заметим: эти события бывают не только из-за мыши, но и эмулируются на других устройствах, в частности, на мобильных, для совместимости.

Типы событий мыши

Мы уже видели некоторые из этих событий:

`mousedown/mouseup`

Кнопка мыши нажата/отпущена над элементом.

`mouseover/mouseout`

Курсор мыши появляется над элементом и уходит с него.

`mousemove`

Каждое движение мыши над элементом генерирует это событие.

`click`

Вызывается при `mousedown`, а затем `mouseup` над одним и тем же элементом, если использовалась левая кнопка мыши.

`dblclick`

Вызывается двойным кликом на элементе.

`contextmenu`

Вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши. Но, заметим, это не совсем событие мыши, оно может вызываться и специальной клавишей клавиатуры.

...Есть также несколько иных типов событий, которые мы рассмотрим позже.

Порядок событий

Как вы можете видеть из приведённого выше списка, действие пользователя может вызвать несколько событий.

Например, клик мышью вначале вызывает `mousedown`, когда кнопка нажата, затем `mouseup` и `click`, когда она отпущена.

В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: `mousedown` → `mouseup` → `click`.

Кнопки мыши

События, связанные с кликом, всегда имеют свойство `button`, которое позволяет получить конкретную кнопку мыши.

Обычно мы не используем его для событий `click` и `contextmenu`, потому что первое происходит только при щелчке левой кнопкой мыши, а второе – только при щелчке правой кнопкой мыши.

С другой стороны, обработчикам `mousedown` и `mouseup` может потребоваться `event.button`, потому что эти события срабатывают на любую кнопку, таким образом `button` позволяет различать «нажатие правой кнопки» и «нажатие левой кнопки».

Возможными значениями `event.button` являются:

Состояние кнопки	<code>event.button</code>
Левая кнопка (основная)	0
Средняя кнопка (вспомогательная)	1
Правая кнопка (вторичная)	2
Кнопка X1 (назад)	3
Кнопка X2 (вперёд)	4

Большинство мышек имеют только левую и правую кнопку, поэтому возможные значения это 0 или 2. Сенсорные устройства также генерируют аналогичные события, когда кто-то нажимает на них.

Также есть свойство `event.buttons`, в котором все нажатые в данный момент кнопки представлены в виде целого числа, по одному биту на кнопку. На практике это свойство используется очень редко, вы можете найти подробную информацию по адресу [MDN ↗](#), если вам это когда-нибудь понадобится.

⚠️ Устаревшее свойство `event.which`

В старом коде вы можете встретить `event.which` свойство – это старый нестандартный способ получения кнопки с возможными значениями:

- `event.which == 1` – левая кнопка,
- `event.which == 2` – средняя кнопка,
- `event.which == 3` – правая кнопка.

На данный момент `event.which` устарел, нам не следует его использовать.

Средняя кнопка сейчас – скорее экзотика, и используется очень редко.

Модификаторы: `shift`, `alt`, `ctrl` и `meta`

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах.

Свойства события:

- `shiftKey : Shift`

- `altKey`: `Alt` (или `Opt` для Mac)
- `ctrlKey`: `Ctrl`
- `metaKey`: `Cmd` для Mac

Они равны `true`, если во время события была нажата соответствующая клавиша.

Например, кнопка внизу работает только при комбинации `Alt+Shift+клик`:

```
<button id="button">Нажми Alt+Shift+Click на мне!</button>

<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Ура!');
    }
  };
</script>
```

Нажми Alt+Shift+Click на мне!

Внимание: обычно на Mac используется клавиша `Cmd` вместо `Ctrl`

В Windows и Linux клавишами-модификаторами являются `Alt`, `Shift` и `Ctrl`. На Mac есть ещё одна: `Cmd`, которой соответствует свойство `metaKey`.

В большинстве приложений, когда в Windows/Linux используется `Ctrl`, на Mac используется `Cmd`.

То есть, когда пользователь Windows нажимает `Ctrl+Enter` и `Ctrl+A`, пользователь Mac нажимает `Cmd+Enter` или `Cmd+A`, и так далее.

Поэтому, если мы хотим поддерживать такие комбинации, как `Ctrl+клик`, то для Mac имеет смысл использовать `Cmd+клик`. Это удобней для пользователей Mac.

Даже если мы и хотели бы заставить людей на Mac использовать именно `Ctrl+клик`, это довольно сложно. Проблема в том, что левый клик в сочетании с `Ctrl` интерпретируется как *правый клик* на MacOS и генерирует событие `contextmenu`, а не `click` как на Windows/Linux.

Поэтому, если мы хотим, чтобы пользователям всех операционных систем было удобно, то вместе с `ctrlKey` нам нужно проверять `metaKey`.

Для JS-кода это означает, что мы должны проверить `if (event.ctrlKey || event.metaKey)`.

Не забывайте про мобильные устройства

Комбинации клавиш хороши в качестве дополнения к рабочему процессу. Так что, если посетитель использует клавиатуру – они работают.

Но если на их устройстве его нет – тогда должен быть способ жить без клавиш-модификаторов.

Координаты: clientX/Y, pageX/Y

Все события мыши имеют координаты двух видов:

1. Относительно окна: `clientX` и `clientY`.
2. Относительно документа: `pageX` и `pageY`.

Мы уже рассмотрели разницу между ними в главе [Координаты](#).

Если вкратце, то относительные координаты документа `pageX/Y` отсчитываются от левого верхнего угла документа и не меняются при прокрутке страницы, в то время как `clientX/Y` отсчитываются от левого верхнего угла текущего окна. Когда страница прокручивается, они меняются.

Например, если у нас есть окно размером 500x500, и курсор мыши находится в левом верхнем углу, то значения `clientX` и `clientY` равны `0`, независимо от того, как прокручивается страница.

А если мышь находится в центре окна, то значения `clientX` и `clientY` равны `250` независимо от того, в каком месте документа она находится и до какого места документ прокручен. В этом они похожи на `position:fixed`.

Координаты относительно документа `pageX`, `pageY` отсчитываются не от окна, а от левого верхнего угла документа. Подробнее о координатах вы можете узнать в главе [Координаты](#).

Отключаем выделение

Двойной клик мыши имеет побочный эффект, который может быть неудобен в некоторых интерфейсах: он выделяет текст.

Например, двойной клик на текст ниже выделяет его в дополнение к нашему обработчику:

```
<span ondblclick="alert('dblclick')">Сделайте двойной клик на мне</span>
```

Сделайте двойной клик на мне

Если зажать левую кнопку мыши и, не отпуская кнопку, провести мышью, то также будет выделение, которое в интерфейсах может быть «не кстати».

Есть несколько способов запретить выделение, о которых вы можете прочитать в главе [Selection и Range](#).

В данном случае самым разумным будет отменить действие браузера по умолчанию при событии `mousedown`, это отменит оба этих выделения:

```
До...
<b ondblclick="alert('Клик!')" onmousedown="return false">
    Сделайте двойной клик на мне
</b>
...После
```

До... Сделайте двойной клик на мне ...После

Теперь выделенный жирным элемент не выделяется при двойном клике, а также на нём нельзя начать выделение, зажав кнопку мыши.

Заметим, что текст внутри него по-прежнему можно выделить, если начать выделение не на самом тексте, а до него или после. Обычно это нормально воспринимается пользователями.

Предотвращение копирования

Если мы хотим отключить выделение для защиты содержимого страницы от копирования, то мы можем использовать другое событие: `oncopy`.

```
<div oncopy="alert('Копирование запрещено!');return false">
    Уважаемый пользователь,
    Копирование информации запрещено для вас.
    Если вы знаете JS или HTML, вы можете найти всю нужную вам информацию в исходном
</div>
```

Уважаемый пользователь, Копирование информации запрещено для вас. Если вы знаете JS или HTML, вы можете найти всю нужную вам информацию в исходном коде страницы.

Если вы попытаетесь скопировать текст в `<div>`, у вас это не получится, потому что срабатывание события `oncopy` по умолчанию запрещено.

Конечно, пользователь имеет доступ к HTML-коду страницы и может взять текст оттуда, но не все знают, как это сделать.

Итого

События мыши имеют следующие свойства:

- Кнопка: `button`.
- Клавиши-модификаторы (`true` если нажаты): `altKey`, `ctrlKey`, `shiftKey` и `metaKey` (Mac).
 - Если вы планируете обработать `Ctrl`, то не забудьте, что пользователи Mac обычно используют `Cmd`, поэтому лучше проверить `if (e.metaKey || e.ctrlKey)`.
- Координаты относительно окна: `clientX/clientY`.
- Координаты относительно документа: `pageX/pageY`.

Действие по умолчанию события `mousedown` – начало выделения, если в интерфейсе оно скорее мешает, его можно отменить.

В следующей главе мы поговорим о событиях, которые возникают при передвижении мыши, и об отслеживании смены элементов под указателем.

✓ Задачи

Выделяемый список

важность: 5

Создайте список, в котором элементы могут быть выделены, как в файловых менеджерах.

- При клике на элемент списка выделяется только этот элемент (добавляется класс `.selected`), отменяется выделение остальных элементов.
- Если клик сделан вместе с `Ctrl` (`Cmd` для Mac), то выделение переключается на элементе, но остальные элементы при этом не изменяются.

Демо:

Кликни на элемент списка, чтобы выделить его.

- Кристофер Робин
- Винни Пух
- Тигра
- Кенга
- Кролик. Просто Кролик.

P.S. В этом задании все элементы списка содержат только текст. Без вложенных тегов.

P.P.S. Предотвратите стандартное для браузера выделение текста при кликах.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Движение мыши: mouseover/out, mouseenter/leave

В этой главе мы более подробно рассмотрим события, возникающие при движении указателя мыши над элементами страницы.

События mouseover/mouseout, relatedTarget

Событие `mouseover` происходит в момент, когда курсор оказывается над элементом, а событие `mouseout` – в момент, когда курсор уходит с элемента.



Эти события являются особенными, потому что у них имеется свойство `relatedTarget`. Оно «дополняет» `target`. Когда мышь переходит с одного элемента на другой, то один из них будет `target`, а другой `relatedTarget`.

Для события `mouseover`:

- `event.target` – это элемент, на который курсор перешёл.
- `event.relatedTarget` – это элемент, с которого курсор ушёл (`relatedTarget → target`).

Для события `mouseout` наоборот:

- `event.target` – это элемент, с которого курсор ушёл.
- `event.relatedTarget` – это элемент, на который курсор перешёл (`target → relatedTarget`).

Свойство `relatedTarget` может быть `null`

Свойство `relatedTarget` может быть `null`.

Это нормально и означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или же, наоборот, ушёл за пределы окна.

Следует держать в уме такую возможность при использовании `event.relatedTarget` в своём коде. Если, например, написать `event.relatedTarget.tagName`, то при отсутствии `event.relatedTarget` будет ошибка.

Пропуск элементов

Событие `mousemove` происходит при движении мыши. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя.

Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события `mousemove`.

Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:

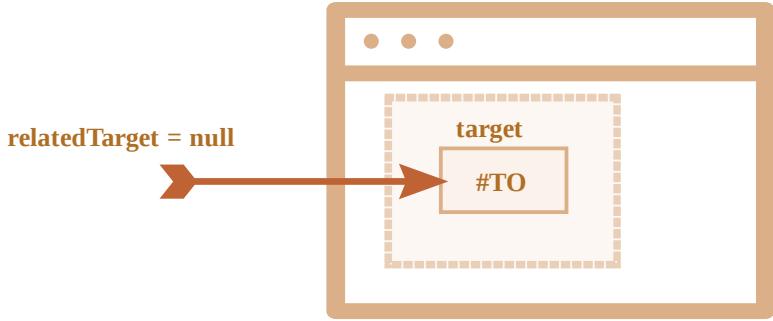


Если курсор мыши передвинуть очень быстро с элемента `#FROM` на элемент `#TO`, как это показано выше, то лежащие между ними элементы `<div>` (или некоторые из них) могут быть пропущены. Событие `mouseout` может запуститься на элементе `#FROM` и затем сразу же сгенерируется `mouseover` на элементе `#TO`.

Это хорошо с точки зрения производительности, потому что если промежуточных элементов много, вряд ли мы действительно хотим обрабатывать вход и выход для каждого.

С другой стороны, мы должны иметь в виду, что указатель мыши не «посещает» все элементы на своём пути. Он может и «прыгать».

В частности, возможно, что указатель запрыгнет в середину страницы из-за пределов окна браузера. В этом случае значение `relatedTarget` будет `null`, так как курсор пришёл «из ниоткуда»:



ⓘ Если был `mouseover`, то будет и `mouseout`

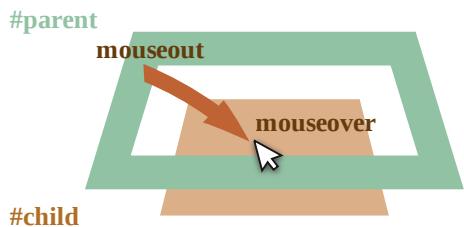
Несмотря на то, что при быстрых переходах промежуточные элементы могут игнорироваться, в одном мы можем быть уверены: элемент может быть пропущен только целиком.

Если указатель «официально» зашёл на элемент, то есть было событие `mouseover`, то при выходе с него обязательно будет `mouseout`.

Событие `mouseout` при переходе на потомка

Важная особенность события `mouseout` – оно генерируется в том числе, когда указатель переходит с элемента на его потомка.

То есть, визуально указатель всё ещё на элементе, но мы получим `mouseout`!



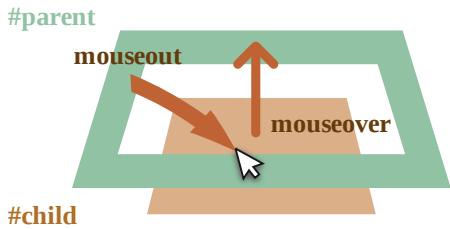
Это выглядит странно, но легко объясняется.

По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным и верхним по z-index.

Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

Обратите внимание на важную деталь.

Событие `mouseover`, происходящее на потомке, всплывает. Поэтому если на родительском элементе есть такой обработчик, то оно его вызовет.



При переходе с родителя элемента на потомка – на родителе сработают два обработчика: и `mouseout` и `mouseover`:

```
parent.onmouseout = function(event) {
  /* event.target: внешний элемент */
};

parent.onmouseover = function(event) {
  /* event.target: внутренний элемент (всплыло) */
};
```

Если код внутри обработчиков не смотрит на `target`, то он подумает, что мышь ушла с элемента `parent` и вернулась на него обратно. Но это не так! Мышикуда не уходила, она просто перешла на потомка.

Если при уходе с элемента что-то происходит, например, запускается анимация, то такая интерпретация происходящего может давать нежелательные побочные эффекты.

Чтобы этого избежать, можно смотреть на `relatedTarget` и, если мышь всё ещё внутри элемента, то игнорировать такие события.

Или же можно использовать другие события: `mouseenter` и `mouseleave`, которые мы сейчас изучим, с ними такая проблема не возникает.

События `mouseenter` и `mouseleave`

События `mouseenter/mouseleave` похожи на `mouseover/mouseout`. Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его.

Но есть и пара важных отличий:

1. Переходы внутри элемента, на его потомки и с них, не считаются.
2. События `mouseenter/mouseleave` не всплывают.

События `mouseenter/mouseleave` предельно просты и понятны.

Когда указатель появляется над элементом – генерируется `mouseenter`, причём не имеет значения, где именно указатель: на самом элементе или на его потомке.

Событие `mouseleave` происходит, когда курсор покидает элемент.

Делегирование событий

События `mouseenter/leave` просты и легки в использовании. Но они не всплывают. Таким образом, мы не можем их делегировать.

Представьте ситуацию, когда мы хотим обрабатывать события, сгенерированные при движении курсора по ячейкам таблицы. И в таблице сотни ячеек.

Очевидное решение – определить обработчик на родительском элементе `<table>` и там обрабатывать возникающие события. Но, так как `mouseenter/leave` не всплывают, то если событие происходит на ячейке `<td>`, то только обработчик на `<td>` может поймать его.

Обработчики событий `mouseenter/leave` на `<table>` срабатывают, если курсор оказывается над таблицей в целом или же уходит с неё. Невозможно получить какую-либо информацию о переходах между ячейками внутри таблицы.

Что ж, не проблема – будем использовать `mouseover/mouseout`.

Начнём с простых обработчиков, которые выделяют текущий элемент под указателем мыши:

```
// выделим элемент под мышью
table.onmouseover = function(event) {
  let target = event.target;
  target.style.background = 'pink';
};

table.onmouseout = function(event) {
  let target = event.target;
  target.style.background = '';
};
```

В нашем случае мы хотим обрабатывать переходы именно между ячейками `<td>`: вход на ячейку и выход с неё. Прочие переходы, в частности, внутри ячейки `<td>` или вообще вне любых ячеек, нас не интересуют, хорошо бы их отфильтровать.

Можно достичь этого так:

- Запоминать текущую ячейку `<td>` в переменную, которую назовём `currentElem`.
- На `mouseover` – игнорировать событие, если мы всё ещё внутри той же самой ячейки `<td>`.
- На `mouseout` – игнорировать событие, если это не уход с текущей ячейки `<td>`.

Вот пример кода, учитывающего все ситуации:

```

// ячейка <td> под курсором в данный момент (если есть)
let currentElem = null;

table.onmouseover = function(event) {
    // перед тем, как войти на следующий элемент, курсор всегда покидает предыдущий
    // если currentElem есть, то мы ещё не ушли с предыдущего <td>,
    // это переход внутри - игнорируем такое событие
    if (currentElem) return;

    let target = event.target.closest('td');

    // переход не на <td> - игнорировать
    if (!target) return;

    // переход на <td>, но вне нашей таблицы (возможно при вложенных таблицах)
    // игнорировать
    if (!table.contains(target)) return;

    // ура, мы зашли на новый <td>
    currentElem = target;
    target.style.background = 'pink';
};

table.onmouseout = function(event) {
    // если мы вне <td>, то игнорируем уход мыши
    // это какой-то переход внутри таблицы, но вне <td>,
    // например с <tr> на другой <tr>
    if (!currentElem) return;

    // мы покидаем элемент – но куда? Возможно, на потомка?
    let relatedTarget = event.relatedTarget;

    while (relatedTarget) {
        // поднимаемся по дереву элементов и проверяем – внутри ли мы currentElem или нет
        // если да, то это переход внутри элемента – игнорируем
        if (relatedTarget == currentElem) return;

        relatedTarget = relatedTarget.parentNode;
    }

    // мы действительно покинули элемент
    currentElem.style.background = '';
    currentElem = null;
};

```

Итого

Мы рассмотрели события `mouseover`, `mouseout`, `mousemove`, `mouseenter` и `mouseleave`.

Особенности, на которые стоит обратить внимание:

- При быстром движении мыши события события не будут возникать на промежуточных элементах.
- События `mouseover/out` и `mouseenter/leave` имеют дополнительное свойство: `relatedTarget`. Оно дополняет свойство `target` и содержит ссылку на элемент, с/на который мы переходим.

События `mouseover/out` возникают, даже когда происходит переход с родительского элемента на потомка. С точки зрения браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным.

События `mouseenter/leave` в этом отличаются. Они генерируются, когда курсор переходит на элемент в целом или уходит с него. Также они не всплывают.

✓ Задачи

Улучшенная подсказка

важность: 5

Напишите JavaScript код, который показывает подсказку над элементом с атрибутом `data-tooltip`. Значение атрибута должно становиться текстом подсказки.

Это похоже на задачу [Поведение "подсказка"](#), но здесь элементы с подсказками могут быть вложены друг в друга. Показываться должна подсказка на самом глубоко вложенном элементе.

Только одна подсказка может быть показана в любой момент времени.

Например:

```
<div data-tooltip="Здесь - домашний интерьер" id="house">
  <div data-tooltip="Здесь - крыша" id="roof"></div>
  ...
  <a href="https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B8_%D0%BF%D0%BE%D1%80%D0%BE%
</div>
```

Результат в iframe:

Жили-были на свете три поросенка. Три брата. Все одинакового роста, кругленькие, розовые, с одинаковыми веселыми хвостиками.

Даже имена у них были похожи. Звали поросят: Ниф-Ниф, Нуф-Нуф и Наф-Наф. Все лето они кувыркались в зеленой траве, грелись на солнышке, нежились в лужах.

Но вот наступила осень. Солнце уже не так сильно припекало, серые облака тянулись над пожелтевшим лесом.

- Пора нам подумать о зиме, - сказал как-то Наф-Наф.
[Наведи курсор на меня](#)

Открыть песочницу для задачи. ↗

[К решению](#)

"Умная" подсказка

важность: 5

Напишите функцию, которая показывает подсказку над элементом только в случае, когда пользователь передвигает мышь *на него*, но не *через него*.

Другими словами, если пользователь подвинул курсор на элементе и остановился – показывать подсказку. А если он просто быстро провёл курсором по элементу, то не надо ничего показывать. Кому понравится лишнее мелькание?

Технически, мы можем измерять скорость прохода курсора мыши над элементом, и если она низкая, то можно посчитать, что пользователь остановил курсор над элементом, и показать ему подсказку. А если скорость высокая, то тогда не показывать.

Создайте для этого универсальный объект `new HoverIntent(options)`.

Его настройки `options`:

- `elem` – отслеживаемый элемент.
- `over` – функция, вызываемая, при заходе на элемент, считаем что заход – это когда курсор медленно двигается или остановился над элементом.
- `out` – функция, вызываемая при уходе курсора с элемента (если был заход).

Пример использования такого объекта для показа подсказки:

```

// пример подсказки
let tooltip = document.createElement('div');
tooltip.className = "tooltip";
tooltip.innerHTML = "Tooltip";

// объект будет отслеживать движение мыши и вызывать функции over/out
new HoverIntent({
  elem,
  over() {
    tooltip.style.left = elem.getBoundingClientRect().left + 'px';
    tooltip.style.top = elem.getBoundingClientRect().bottom + 5 + 'px';
    document.body.append(tooltip);
  },
  out() {
    tooltip.remove();
  }
});

```

Демо:

The screenshot shows a browser developer tools interface. On the left, there is a yellow box containing the text "12 : 30 : 00". To the right of this box, the text "passes: 5 failures: 0 duration: 4.49s 100%" is displayed. Below these, the word "hoverIntent" is shown, followed by a list of three green checkmarks: "mouseover -> immediately no tooltip", "mouseover -> pause shows tooltip", and "mouseover -> fast mouseout no tooltip". There are also three small circular arrows on the right side of the list.

Если двигать курсор над «часами» быстро, то ничего не произойдёт, а если вы замедлите движение курсора над элементом или остановите его, то будет показана подсказка.

Обратите внимание: подсказка не должна пропадать (мигать), когда курсор переходит между дочерними элементами часов.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Drag'n'Drop с событиями мыши

Drag'n'Drop – отличный способ улучшить интерфейс. Захват элемента мышкой и его перенос визуально упростят что угодно: от копирования и перемещения документов (как в файловых менеджерах) до оформления заказа (« положить в корзину »).

В современном стандарте HTML5 есть [раздел о Drag and Drop](#) – и там есть специальные события именно для Drag'n'Drop переноса, такие как `dragstart`,

`dragend` и так далее.

Они интересны тем, что позволяют легко решать простые задачи. Например, можно перетащить файл в браузер, так что JS получит доступ к его содержимому.

Но у них есть и 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 встроенным событиями не реализуемы. Кроме того, мобильные устройства плохо их поддерживают.

Здесь мы будем рассматривать Drag'n'Drop при помощи событий мыши.

Алгоритм Drag'n'Drop

Базовый алгоритм Drag'n'Drop выглядит так:

1. При `mousedown` – готовим элемент к перемещению, если необходимо (например, создаём его копию).
2. Затем при `mousemove` передвигаем элемент на новые координаты путём смены `left/top` и `position:absolute`.
3. При `mouseup` – остановить перенос элемента и произвести все действия, связанные с окончанием Drag'n'Drop.

Это и есть основа Drag'n'Drop. Позже мы сможем расширить этот алгоритм, например, подсветив элементы при наведении на них мыши.

В следующем примере эти шаги реализованы для переноса мяча:

```
ball.onmousedown = function(event) { // (1) отследить нажатие

    // (2) подготовить к перемещению:
    // разместить поверх остального содержимого и в абсолютных координатах
    ball.style.position = 'absolute';
    ball.style.zIndex = 1000;
    // переместим в body, чтобы мяч был точно не внутри position:relative
    document.body.append(ball);
    // и установим абсолютно спозиционированный мяч под курсор

    moveAt(event.pageX, event.pageY);

    // передвинуть мяч под координаты курсора
    // и сдвинуть на половину ширины/высоты для центрирования
    function moveAt(pageX, pageY) {
        ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
        ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
    }

    function onMouseMove(event) {
```

```
moveAt(event.pageX, event.pageY);  
}  
  
// (3) перемещать по экрану  
document.addEventListener('mousemove', onMouseMove);  
  
// (4) положить мяч, удалить более ненужные обработчики событий  
ball.onmouseup = function() {  
    document.removeEventListener('mousemove', onMouseMove);  
    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».

Всё 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 деталь – 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.

Но, как мы помним, событие `mousemove` возникает хоть и часто, но не для каждого пикселя. Поэтому из-за быстрого движения указатель может спрыгнуть с мяча и оказаться где-нибудь в середине документа (или даже за пределами окна).

Вот почему мы должны отслеживать `mousemove` на всём `document`, чтобы поймать его.

Правильное позиционирование

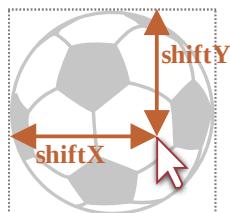
В примерах выше мяч позиционируется так, что его центр оказывается под указателем мыши:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';  
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Неплохо, но есть побочные эффекты. Мы, для начала переноса, можем нажать мышью на любом месте мяча. Если мячик «взят» за самый край – то в начале переноса он резко «прыгает», центрируясь под указателем мыши.

Было бы лучше, если бы изначальный сдвиг курсора относительно элемента сохранялся.

Где захватили, за ту «часть элемента» и переносим:



Обновим наш алгоритм:

- Когда человек нажимает на мячик (`mousedown`) – запомним расстояние от курсора до левого верхнего угла шара в переменных `shiftX`/`shiftY`.
Далее будем удерживать это расстояние при перетаскивании.

Чтобы получить этот сдвиг, мы можем вычесть координаты:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

- Далее при переносе мяча мы позиционируем его с тем же сдвигом относительно указателя мыши, вот так:

```
// onmousemove
// ball has position:absolute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';
```

Итоговый код с правильным позиционированием:

```
ball.onmousedown = function(event) {

  let shiftX = event.clientX - ball.getBoundingClientRect().left;
  let shiftY = event.clientY - ball.getBoundingClientRect().top;

  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  document.body.append(ball);
```

```

moveAt(event.pageX, event.pageY);

// переносит мяч на координаты (pageX, pageY),
// дополнительно учитывая изначальный сдвиг относительно указателя мыши
function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
}

function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
}

// передвигаем мяч при событии mousemove
document.addEventListener('mousemove', onMouseMove);

// отпустить мяч, удалить ненужные обработчики
ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    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.

Цели переноса (droppable)

В предыдущих примерах мяч можно было бросить просто где угодно в пределах окна. В реальности мы обычно берём один элемент и перетаскиваем в другой. Например, «файл» в «папку» или что-то ещё.

Абстрактно говоря, мы берём перетаскиваемый (draggable) элемент и помещаем его в другой элемент «цель переноса» (droppable).

Нам нужно знать:

- куда пользователь положил элемент в конце переноса, чтобы обработать его окончание
- и, желательно, над какой потенциальной целью (элемент, куда можно положить, например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

Решение довольно интересное и немного хитрое, давайте рассмотрим его.

Какой может быть первая мысль? Возможно, установить обработчики событий `mouseover/mouseup` на элемент – потенциальную цель переноса?

Но это не работает.

Проблема в том, что при перемещении перетаскиваемый элемент всегда находится поверх других элементов. А события мыши срабатывают только на верхнем элементе, но не на нижнем.

Например, у нас есть два элемента `<div>`: красный поверх синего (полностью перекрывает). Не получится поймать событие на синем, потому что красный сверху:

```
<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>


</div>


</div>


```



То же самое с перетаскиваемым элементом. Мяч всегда находится поверх других элементов, поэтому события срабатывают на нём. Какие бы обработчики мы ни ставили на нижние элементы, они не будут выполнены.

Вот почему первоначальная идея поставить обработчики на потенциальные цели переноса нереализуема. Обработчики не сработают.

Так что же делать?

Существует метод `document.elementFromPoint(clientX, clientY)`. Он возвращает наиболее глубоко вложенный элемент по заданным координатам окна (или `null`, если указанные координаты находятся за пределами окна).

Мы можем использовать его, чтобы из любого обработчика событий мыши выяснить, над какой мы потенциальной целью переноса, вот так:

```
// внутри обработчика события мыши
ball.hidden = true; // (*) прячем переносимый элемент

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow - элемент под мячом (возможная цель переноса)
```

```
ball.hidden = false;
```

Заметим, нам нужно спрятать мяч перед вызовом функции (*). В противном случае по этим координатам мы будем получать мяч, ведь это и есть элемент непосредственно под указателем: elemBelow=ball. Так что мы прячем его и тут же показываем обратно.

Мы можем использовать этот код для проверки того, над каким элементом мы «летим», в любое время. И обработать окончание переноса, когда оно случится.

Расширенный код `onMouseMove` с поиском потенциальных целей переноса:

```
// потенциальная цель переноса, над которой мы пролетаем прямо сейчас
let currentDroppable = null;

function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);

    ball.hidden = true;
    let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
    ball.hidden = false;

    // событие mousemove может произойти и когда указатель за пределами окна
    // (мяч перетащили за пределы экрана)

    // если clientX/clientY за пределами окна, elementFromPoint вернёт null
    if (!elemBelow) return;

    // потенциальные цели переноса помечены классом droppable (может быть и другая логика)
    let droppableBelow = elemBelow.closest('.droppable');

    if (currentDroppable != droppableBelow) {
        // мы либо залетаем на цель, либо улетаем из неё
        // внимание: оба значения могут быть null
        // currentDroppable=null,
        // если мы были не над droppable до этого события (например, над пустым пространством)
        // droppableBelow=null,
        // если мы не над droppable именно сейчас, во время этого события

        if (currentDroppable) {
            // логика обработки процесса "вылета" из droppable (удаляем подсветку)
            leaveDroppable(currentDroppable);
        }
        currentDroppable = droppableBelow;
        if (currentDroppable) {
            // логика обработки процесса, когда мы "влетаем" в элемент droppable
            enterDroppable(currentDroppable);
        }
    }
}
```

В приведённом ниже примере, когда мяч перетаскивается через футбольные ворота, ворота подсвечиваются.

<https://plnkr.co/edit/Q1afzDb6iz4aJfWp?p=preview>

Теперь в течение всего процесса в переменной `currentDroppable` мы храним текущую потенциальную цель переноса, над которой мы сейчас, можем её подсветить или сделать что-то ещё.

Итого

Мы рассмотрели основной алгоритм Drag'n'Drop.

Ключевые идеи:

1. Поток событий: `ball.mousedown` → `documentmousemove` → `ball.mouseup` (не забудьте отменить браузерный `ondragstart`).
2. В начале перетаскивания: запоминаем начальное смещение указателя относительно элемента: `shiftX/shiftY` – и сохраняем его при перетаскивании.
3. Выявляем потенциальные цели переноса под указателем с помощью `document.elementFromPoint`.

На этой основе можно сделать многое.

- На `mouseup` – по-разному завершать перенос: изменять данные, перемещать элементы.
- Можно подсвечивать элементы, пока мышь «пролетает» над ними.
- Можно ограничить перетаскивание определённой областью или направлением.
- Можно использовать делегирование событий для `mousedown/up`. Один обработчик событий на большой зоне, который проверяет `event.target`, может управлять Drag'n'Drop для сотен элементов.
- И так далее.

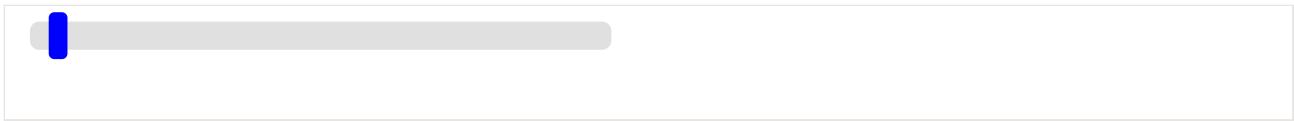
Существуют фреймворки, которые строят архитектуру поверх этого алгоритма, создавая такие классы, как `DragZone`, `Droppable`, `Draggable`. Большинство из них делают вещи, аналогичные описанным выше. Вы можете и сами создать вашу собственную реализацию переноса, как видите, это достаточно просто, возможно, проще, чем адаптация чего-то готового.

✓ Задачи

Слайдер

важность: 5

Создайте слайдер:



Захватите мышкой синий бегунок и двигайте его.

Важные детали:

- Слайдер должен нормально работать при резком движении мыши влево или вправо за пределы полосы. При этом бегунок должен останавливаться чётко в нужном конце полосы.
- При нажатом бегунке мышь может выходить за пределы полосы слайдера, но слайдер пусть всё 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 и DOM.

Сделайте так, чтобы элементы с классом `draggable` можно было переносить мышкой. Как мяч в этой главе.

Требования к реализации:

- Используйте делегирование событий для отслеживания начала перетаскивания: только один обработчик событий `mousedown` на документе.
- Если элементы подносят к верхней/нижней границе окна – оно должно прокручиваться вверх/вниз, чтобы позволить дальнейшее перетаскивание.
- Горизонтальная прокрутка отсутствует (чуть-чуть упрощает задачу, её просто добавить).
- Элемент при переносе, даже при резких движениях мышкой, не должен даже частично попасть вне окна.

Демо слишком велико для размещения здесь, перейдите по ссылке ниже.

[Демо в новом окне](#) ↗

[Открыть песочницу для задачи.](#) ↗

Клавиатура: keydown и keyup

Прежде чем перейти к клавиатуре, обратите внимание, что на современных устройствах есть и другие способы «ввести что-то». Например, распознавание речи (это особенно актуально на мобильных устройствах) или Копировать/Вставить с помощью мыши.

Поэтому, если мы хотим корректно отслеживать ввод в поле `<input>`, то одних клавиатурных событий недостаточно. Существует специальное событие `input`, чтобы отслеживать любые изменения в поле `<input>`. И оно справляется с такой задачей намного лучше. Мы рассмотрим его позже в главе [События: change, input, cut, copy, paste](#).

События клавиатуры же должны использоваться, если мы хотим обрабатывать взаимодействие пользователя именно с клавиатурой (в том числе виртуальной). К примеру, если нам нужно реагировать на стрелочные клавиши `Up` и `Down` или горячие клавиши (включая комбинации клавиш).

Тестовый стенд

Для того, чтобы лучше понять, как работают события клавиатуры, можно использовать [тестовый стенд](#).

События keydown и keyup

Событие `keydown` происходит при нажатии клавиши, а `keyup` – при отпускании.

`event.code` и `event.key`

Свойство `key` объекта события позволяет получить символ, а свойство `code` – «физический код клавиши».

К примеру, одну и ту же клавишу `Z` можно нажать с клавишей `Shift` и без неё. В результате получится два разных символа: `z` в нижнем регистре и `Z` в верхнем регистре.

Свойство `event.key` – это непосредственно символ, и он может различаться. Но `event.code` всегда будет тот же:

Клавиша	<code>event.key</code>	<code>event.code</code>
<code>Z</code>	<code>z</code> (нижний регистр)	<code>KeyZ</code>
<code>Shift+Z</code>	<code>Z</code> (Верхний регистр)	<code>KeyZ</code>

Если пользователь работает с разными языками, то при переключении на другой язык символ изменится с "Z" на совершенно другой. Получившееся станет новым значением `event.key`, тогда как `event.code` останется тем же: "KeyZ".

«KeyZ» и другие клавишиные коды

У каждой клавиши есть код, который зависит от её расположения на клавиатуре. Подробно о клавишиных кодах можно прочитать в [спецификации о кодах событий UI](#).

Например:

- Буквенные клавиши имеют коды по типу "Key<буква>": "KeyA", "KeyB" и т.д.
- Коды числовых клавиш строятся по принципу: "Digit<число>": "Digit0", "Digit1" и т.д.
- Код специальных клавиш – это их имя: "Enter", "Backspace", "Tab" и т.д.

Существует несколько широко распространённых раскладок клавиатуры, и в спецификации приведены клавишиные коды к каждой из них.

Можно их прочитать в [разделе спецификации, посвящённом буквенно-цифровым клавишам](#) или просто нажмите нужную клавишу на [тестовом стенде](#) выше и посмотрите.

Регистр важен: "KeyZ", а не "keyZ"

Выглядит очевидно, но многие всё равно ошибаются.

Пожалуйста, избегайте опечаток: правильно `KeyZ`, а не `keyZ`. Условие `event.code=="keyZ"` работать не будет: первая буква в слове "Key" должна быть заглавная.

А что, если клавиша не буквенно-цифровая? Например, `Shift` или `F1`, или какая-либо другая специальная клавиша? В таких случаях значение свойства `event.key` примерно тоже, что и у `event.code`:

Клавиша	<code>event.key</code>	<code>event.code</code>
<code>F1</code>	<code>F1</code>	<code>F1</code>
<code>Backspace</code>	<code>Backspace</code>	<code>Backspace</code>
<code>Shift</code>	<code>Shift</code>	<code>ShiftRight</code> или <code>ShiftLeft</code>

Обратите внимание, что `event.code` точно указывает, какая именно клавиша нажата. Так, большинство клавиатур имеют по две клавиши `Shift`: слева и справа. `event.code` уточняет, какая именно из них была нажата, в то время как `event.key` сообщает о «смысле» клавиши: что вообще было нажато (`Shift`).

Допустим, мы хотим обработать горячую клавишу `Ctrl+Z` (или `Cmd+Z` для Mac). Большинство текстовых редакторов к этой комбинации подключают действие «Отменить». Мы можем поставить обработчик событий на `keydown` и проверять, какая клавиша была нажата.

Здесь возникает дилемма: в нашем обработчике стоит проверять значение `event.key` или `event.code`?

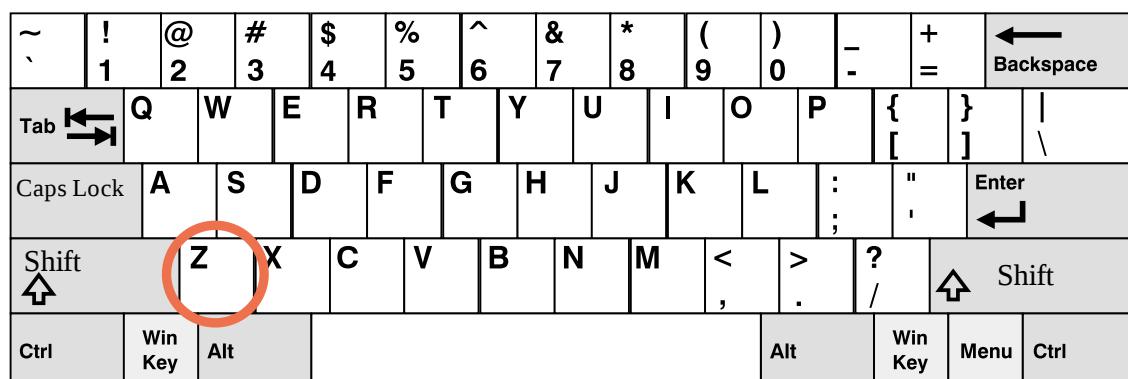
С одной стороны, значение `event.key` – это символ, он изменяется в зависимости от языка, и если у пользователя установлено в ОС несколько языков, и он переключается между ними, нажатие на одну и ту же клавишу будет давать разные символы. Так что имеет смысл проверять `event.code`, ведь его значение всегда одно и тоже.

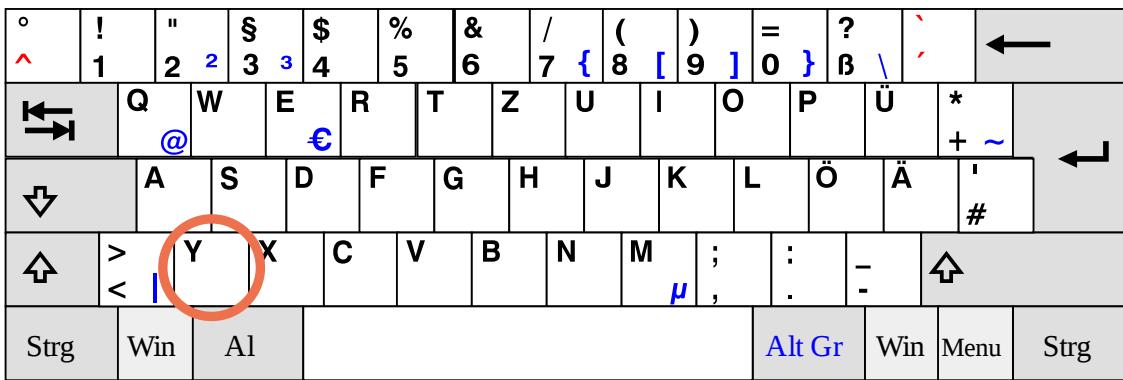
Вот пример кода:

```
document.addEventListener('keydown', function(event) {
  if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
    alert('Отменить!')
  }
});
```

С другой стороны, с `event.code` тоже есть проблемы. На разных раскладках к одной и той же клавише могут быть привязаны разные символы.

Например, вот схема стандартной (US) раскладки («QWERTY») и под ней немецкой («QWERTZ») раскладки (из Википедии):





Для одной и той же клавиши в американской раскладке значение `event.code` равно «`Z`», в то время как в немецкой «`Y`».

Буквально, для пользователей с немецкой раскладкой `event.code` при нажатии на `Y` будет равен `KeyZ`.

Если мы будем проверять в нашем коде `event.code == 'KeyZ'`, то для людей с немецкой раскладкой такая проверка сработает, когда они нажимают `Y`.

Звучит очень странно, но это и в самом деле так. В [спецификации ↗](#) прямо упоминается такое поведение.

Так что `event.code` может содержать неправильный символ при неожиданной раскладке. Одни и те же буквы на разных раскладках могут сопоставляться с разными физическими клавишами, что приводит к разным кодам. К счастью, это происходит не со всеми кодами, а с несколькими, например `KeyA`, `KeyQ`, `KeyZ` (как мы уже видели), и не происходит со специальными клавишами, такими как `Shift`. Вы можете найти полный список проблемных кодов в [спецификации ↗](#).

Чтобы отслеживать символы, зависящие от раскладки, `event.key` надёжнее.

С другой стороны, преимущество `event.code` заключается в том, что его значение всегда остаётся неизменным, будучи привязанным к физическому местоположению клавиши, даже если пользователь меняет язык. Так что горячие клавиши, использующие это свойство, будут работать даже в случае переключения языка.

Хотим поддерживать клавиши, меняющиеся при раскладке? Тогда `event.key` – верный выбор.

Или мы хотим, чтобы горячая клавиша срабатывала даже после переключения на другой язык? Тогда `event.code` может быть лучше.

Автоповтор

При долгом нажатии клавиши возникает автоповтор: `keydown` срабатывает снова и снова, и когда клавишу отпускают, то отрабатывает `keyup`. Так что ситуация, когда много `keydown` и один `keyup`, абсолютно нормальна.

Для событий, вызванных автоповтором, у объекта события свойство `event.repeat` равно `true`.

Действия по умолчанию

Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре.

Для примера:

- Появление символа (самое очевидное).
- Удаление символа (клавиша `Delete`).
- Прокрутка страницы (клавиша `PageDown`).
- Открытие диалогового окна браузера «Сохранить» (`Ctrl+S`)
- ...и так далее.

Предотвращение стандартного действия с помощью

`event.preventDefault()` работает практически во всех сценариях, кроме тех, которые происходят на уровне операционной системы. Например, комбинация `Alt+F4` инициирует закрытие браузера в Windows, что бы мы ни делали в JavaScript.

Для примера, `<input>` ниже ожидает телефонный номер, так что ничего кроме чисел, `+`, `(` или `-` принято не будет:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key ==
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Введите телефон" type="text">
```

Введите телефон

Заметьте, что специальные клавиши, такие как `Backspace`, `Left`, `Right`, `Ctrl+V`, в этом поле для ввода не работают. Это побочный эффект чересчур жёсткого фильтра `checkPhoneKey`.

Добавим ему немного больше свободы:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key ==
    key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' || key == 'Backspace'
}
```

```
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Введите телефон" type="text">
```

Введите телефон

Теперь стрелочки и удаление прекрасно работают.

...Впрочем, мы всё равно можем ввести в `<input>` что угодно с помощью правого клика мыши и пункта «Вставить» контекстного меню. Так что такой фильтр не обладает 100% надёжностью. Мы можем просто оставить всё как есть, потому что в большинстве случаев это работает. Альтернатива – отслеживать событие `input`, оно генерируется после любых изменений в поле `<input>`, и мы можем проверять новое значение и подчёркивать/изменять его, если оно не подходит.

«Дела минувших дней»

В прошлом существовало также событие `keypress`, а также свойства `keyCode`, `charCode`, `which` у объекта события.

Но количество браузерных несовместимостей при работе с ними было столь велико, что у разработчиков спецификации не было другого выхода, кроме как объявить их устаревшими и создать новые, современные события (которые и описываются в этой главе). Старый код ещё работает, так как браузеры продолжают поддерживать и `keypress`, и `keyCode` с `charCode`, и `which`, но более нет никакой необходимости в их использовании.

Итого

Нажатие клавиши всегда генерирует клавиатурное событие, будь то буквенно-цифровая клавиша или специальная типа `Shift` или `Ctrl` и т.д.

Единственным исключением является клавиша `Fn`, которая присутствует на клавиатуре некоторых ноутбуков. События на клавиатуре для неё нет, потому что она обычно работает на уровне более низком, чем даже ОС.

События клавиатуры:

- `keydown` – при нажатии на клавишу (если клавиша остаётся нажатой, происходит автоповтор),
- `keyup` – при отпускании клавиши.

Главные свойства для работы с клавиатурными событиями:

- `code` – «код клавиши» (`"KeyA"`, `"ArrowLeft"` и так далее), особый код, привязанный к физическому расположению клавиши на клавиатуре.

- `key` – символ ("A", "a" и так далее), для не буквенно-цифровых групп клавиш (таких как `Esc`) обычно имеет то же значение, что и `code`.

В прошлом события клавиатуры иногда использовались для отслеживания ввода данных пользователем в полях формы. Это ненадёжно, потому как ввод данных не обязательно может осуществляться с помощью клавиатуры.

Существуют события `input` и `change` специально для обработки ввода (рассмотренные позже в главе [События: change, input, cut, copy, paste](#)). Они срабатывают в результате любого ввода, включая Копировать/Вставить мышью и распознавание речи.

События клавиатуры же должны использоваться только по назначению – для клавиатуры. Например, чтобы реагировать на горячие или специальные клавиши.

✓ Задачи

Отследить одновременное нажатие

важность: 5

Создайте функцию `runOnKeys(func, code1, code2, ... code_n)`, которая запускает `func` при одновременном нажатии клавиш с кодами `code1`, `code2`, ..., `code_n`.

Например, код ниже выведет `alert` при одновременном нажатии клавиш "Q" и "W" (в любом регистре, в любой раскладке)

```
runOnKeys(  
  () => alert("Привет!"),  
  "KeyQ",  
  "KeyW"  
);
```

[Демо в новом окне ↗](#)

[К решению](#)

События указателя

События указателя (Pointer events) – это современный способ обработки ввода с помощью различных указывающих устройств, таких как мышь, перо/стилус, сенсорный экран и так далее.

Краткая история

Сделаем небольшой обзор, чтобы вы поняли общую картину и место событий указателя среди других типов событий.

- Давным-давно, в прошлом, существовали только события мыши

Затем получили широкое распространение сенсорные устройства, в частности телефоны и планшеты. Чтобы скрипты корректно работали, они генерировали (и до сих пор генерируют) события мыши. Например, касание сенсорного экрана генерирует событие `mousedown`. Таким образом, сенсорные устройства позволяли работать с существующими веб-страницами.

Но сенсорные устройства во многих аспектах мощнее, чем мышь. Например, они позволяют касаться экрана сразу в нескольких местах («мульти-тач»). Однако, события мыши не имеют необходимых свойств для обработки таких прикосновений.

- Поэтому появились события касания (Touch events), такие как `touchstart`, `touchend`, `touchmove`, которые имеют специфичные для касаний свойства (мы не будем здесь рассматривать их подробно, потому что события указателя ещё лучше).

Но и этих событий оказалось недостаточно, так как существует много других устройств, таких как перо, у которых есть свои особенности. Кроме того, универсальный код, который отслеживал бы и события касаний и события мыши, неудобно писать.

- Для решения этих задач был внедрён стандарт Pointer Events («События Указателя»). Он предоставляет единый набор событий для всех типов указывающих устройств.

К настоящему времени спецификация [Pointer Events Level 2](#) поддерживается всеми основными браузерами, а [Pointer Events Level 3](#) находится в разработке и почти полностью совместима с Pointer Events Level 2.

Если вы не разрабатываете под старые браузеры, такие как Internet Explorer 10, Safari 12, или более ранние версии, больше нет необходимости использовать события мыши или касаний – можно переходить сразу на события указателя.

При этом ваш код будет корректно работать и с сенсорными устройствами и с мышью. Впрочем, у событий указателя есть важные особенности, которые нужно знать, чтобы их правильно использовать, без лишних сюрпризов. Мы отметим их в этой статье.

Типы событий указателя

Схема именований событий указателя похожа на события мыши:

Событие указателя	Аналогичное событие мыши
pointerdown	mousedown
pointerup	mouseup
pointermove	mousemove
pointerover	mouseover
pointerout	mouseout
pointerenter	mouseenter
pointerleave	mouseleave
pointercancel	-
gotpointercapture	-
lostpointercapture	-

Как мы видим, для каждого `mouse<события>` есть соответствующее `pointer<событие>`, которое играет аналогичную роль. Также есть 3 дополнительных события указателя, у которых нет соответствующего аналога `mouse...`, скоро мы их разберём.

ℹ Замена `mouse<событий>` на `pointer<события>` в коде

Мы можем заменить события `mouse...` на аналогичные `pointer...` в коде и быть уверенными, что с мышью по-прежнему всё будет работать нормально.

При этом поддержка сенсорных устройств «волшебным» образом улучшится. Хотя, возможно, кое-где понадобится добавить `touch-action: none` в CSS. Подробнее мы разберём это ниже, в секции про `pointercancel`.

Свойства событий указателя

События указателя содержат те же свойства, что и события мыши, например `clientX/Y`, `target` и т.п., и несколько дополнительных:

- `pointerId` – уникальный идентификатор указателя, вызвавшего событие.
Идентификатор генерируется браузером. Это свойство позволяет обрабатывать несколько указателей, например сенсорный экран со стилусом и мульти-тач (увидим примеры ниже).
- `pointerType` – тип указывающего устройства. Должен быть строкой с одним из значений: «`mouse`», «`pen`» или «`touch`».
Мы можем использовать это свойство, чтобы определять разное поведение для разных типов указателей.

- `isPrimary` – равно `true` для основного указателя (первый палец в мульти-тач).

Некоторые устройства измеряют область контакта и степень надавливания, например пальца на сенсорном экране, для этого есть дополнительные свойства:

- `width` – ширина области соприкосновения указателя (например, пальца) с устройством. Если не поддерживается, например мышью, то всегда равно `1`.
- `height` – высота области соприкосновения указателя с устройством. Если не поддерживается, например мышью, то всегда равно `1`.
- `pressure` – степень давления указателя в диапазоне от 0 до 1. Для устройств, которые не поддерживают давление, принимает значение `0.5` (нажато) либо `0`.
- `tangentialPressure` – нормализованное тангенциальное давление.
- `tiltX`, `tiltY`, `twist` – специфичные для пера свойства, описывающие положение пера относительно сенсорной поверхности.

Эти свойства большинством устройств не поддерживаются, поэтому редко используются. При необходимости, подробности о них можно найти в [спецификации](#).

Мульти-тач

Одной из функций, которую абсолютно не поддерживают события мыши, является мульти-тач: возможность касаться сразу нескольких мест на телефоне или планшете или выполнять специальные жесты.

События указателя позволяют обрабатывать мульти-тач с помощью свойств `pointerId` и `isPrimary`.

Вот что происходит, когда пользователь касается сенсорного экрана в одном месте, а затем в другом:

1. При касании первым пальцем:

- происходит событие `pointerdown` со свойством `isPrimary=true` и некоторым `pointerId`.

2. При касании вторым и последующими пальцами (при остающемся первом):

- происходит событие `pointerdown` со свойством `isPrimary=false` и уникальным `pointerId` для каждого касания.

Обратите внимание: `pointerId` присваивается не на всё устройство, а для каждого касающегося пальца. Если коснуться экрана 5 пальцами одновременно, получим 5 событий `pointerdown`, каждое со своими координатами и индивидуальным `pointerId`.

События, связанные с первым пальцем, всегда содержат свойство `isPrimary=true`.

Мы можем отслеживать несколько касающихся экрана пальцев, используя их `pointerId`. Когда пользователь перемещает, а затем убирает палец, получаем события `pointermove` и `pointerup` с тем же `pointerId`, что и при событии `pointerdown`.

Событие: `pointercancel`

Событие `pointercancel` происходит, когда текущее действие с указателем по какой-то причине прерывается, и события указателя больше не генерируются.

К таким причинам можно отнести:

- Указывающее устройство было физически выключено.
- Изменилась ориентация устройства (перевернули планшет).
- Браузер решил сам обработать действие, считая его жестом мыши, масштабированием и т.п.

Мы продемонстрируем `pointercancel` на практическом примере, чтобы увидеть, как это влияет на нас.

Допустим, мы реализуем перетаскивание («drag-and-drop») для нашего мяча, как в начале статьи [Drag'n'Drop с событиями мыши](#).

Вот последовательность действий пользователя и соответствующие события:

1. Пользователь нажимает на изображении, чтобы начать перетаскивание
 - происходит событие `pointerdown`
2. Затем он перемещает указатель, двигая изображение
 - происходит событие `pointermove` (возможно, несколько раз)
3. И тут происходит сюрприз! Браузер имеет встроенную поддержку «Drag'n'Drop» для изображений, которая запускает и перехватывает процесс перетаскивания, генерируя при этом событие `pointercancel`.
 - Теперь браузер сам обрабатывает перетаскивание изображения. У пользователя есть возможность перетащить изображение мяча даже за пределы браузера, в свою почтовую программу или файловый менеджер.
 - Событий `pointermove` для нас больше не генерируются.

Таким образом, браузер «перехватывает» действие: в начале переноса drag-and-drop запускается событие `pointercancel`, и после этого события `pointermove` больше не генерируются.

Мы бы хотели реализовать перетаскивание самостоятельно, поэтому давайте скажем браузеру не перехватывать его.

Предотвращайте действие браузера по умолчанию, чтобы избежать `pointercancel`.

Нужно сделать две вещи:

1. Предотвратить запуск встроенного drag'n'drop

- Мы можем сделать это, задав `ball.ondragstart = () => false`, как описано в статье [Drag'n'Drop с событиями мыши](#).
- Это работает для событий мыши.

2. Для устройств с сенсорным экраном существуют другие действия браузера, связанные с касаниями, кроме drag'n'drop. Чтобы с ними не возникало проблем:

- Мы можем предотвратить их, добавив в CSS свойство `#ball { touch-action: none }`.
- Затем наш код начнёт корректно работать на устройствах с сенсорным экраном

После того, как мы это сделаем, события будут работать как и ожидается, браузер не будет перехватывать процесс и не будет вызывать событие `pointercancel`.

Теперь мы можем добавить код для перемещения мяча и наш drag'n'drop будет работать и для мыши и для устройств с сенсорным экраном.

Захват указателя

Захват указателя – особая возможность событий указателя.

Общая идея очень проста, но поначалу может показаться странной, так как для других событий ничего подобного просто нет.

Основной метод:

- `elem.setPointerCapture(pointerId)` – привязывает события с данным `pointerId` к `elem`. После такого вызова все события указателя с таким `pointerId` будут иметь `elem` в качестве целевого элемента (как будто произошли над `elem`), вне зависимости от того, где в документе они произошли.

Другими словами, `elem.setPointerCapture(pointerId)` меняет `target` всех дальнейших событий с данным `pointerId` на `elem`.

Эта привязка отменяется:

- автоматически, при возникновении события `pointerup` или `pointercancel`,
- автоматически, если `elem` удаляется из документа,

- при вызове `elem.releasePointerCapture(pointerId)`.

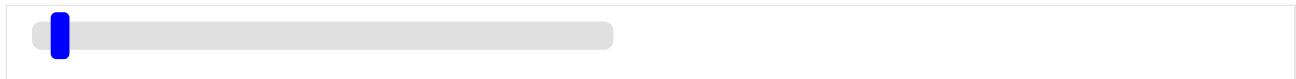
Захват указателя используется для упрощения операций с переносом (drag'n'drop) элементов.

В качестве примера давайте вспомним реализацию слайдера из статьи [Drag'n'Drop с событиями мыши](#).

Мы делаем элемент для слайдера – полоску с «ползунком» (`thumb`) внутри:

```
<div class="slider">
  <div class="thumb"></div>
</div>
```

Со стилями, это выглядит следующим образом:



Затем он работает так:

1. Пользователь сначала нажимает на ползунок `thumb` – срабатывает `pointerdown`.
2. Затем двигает его указателем – срабатывает `pointermove`, и наш код перемещает элемент `thumb`.
 - ...Причём, по мере движения, указатель может покидать ползунок – перемещаться выше или ниже. При этом ползунок должен передвигаться строго по горизонтали, на одной линии с указателем.

В решении, основанном на событиях мыши, для отслеживания всех движений указателя, включая те, которые происходят выше/ниже элемента `thumb`, мы должны были назначить обработчик события `mousemove` на весь документ `document`.

Однако это не самое правильное решение. Одна из проблем – это то, что движения указателя по документу могут вызвать сторонние эффекты, заставить работать другие обработчики (например, `mouseover`), не имеющие отношения к слайдеру.

Именно здесь вступает в игру `setPointerCapture`:

- Мы можем вызывать `thumb.setPointerCapture(event.pointerId)` в обработчике `pointerdown`,
- Тогда дальнейшие события указателя до `pointerup/cancel` будут привязаны к `thumb`.
- Затем, когда произойдёт `pointerup` (передвижение завершено), привязка будет автоматически удалена, нам об этом не нужно беспокоиться.

Так что, даже если пользователь будет двигать указателем по всему документу, обработчики событий будут вызваны на `thumb`. Причём все свойства объекта события, такие как `clientX/clientY`, будут корректны – захват указателя влияет только на `target/currentTarget`.

Вот основной код:

```
thumb.onpointerdown = function(event) {
    // перенацелить все события указателя (до pointerup) на thumb
    thumb.setPointerCapture(event.pointerId);
    // начать отслеживание перемещения указателя
    thumb.onpointermove = function(event) {
        // перемещение слайдера: отслеживание thumb, т.к все события указателя перенацелены
        let newLeft = event.clientX - slider.getBoundingClientRect().left;
        thumb.style.left = newLeft + 'px';
    };
    // если сработало событие pointerup, завершить отслеживание перемещения указателя
    thumb.onpointerup = function(event) {
        thumb.onpointermove = null;
        thumb.onpointerup = null;
        // ...при необходимости также обработайте "конец перемещения"
    };
};
// примечание: нет необходимости вызывать thumb.releasePointerCapture,
// это происходит автоматически при pointerup
```

Таким образом, мы имеем два бонуса:

1. Код становится чище, поскольку нам больше не нужно добавлять/удалять обработчики для всего документа. Удаление привязки происходит автоматически.
2. Если в документе есть какие-то другие обработчики `pointermove`, то они не будут нечаянно вызваны, пока пользователь находится в процессе перетаскивания слайдера.

События при захвате указателя

Существует два связанных с захватом события:

- `gotpointercapture` срабатывает, когда элемент использует `setPointerCapture` для включения захвата.
- `lostpointercapture` срабатывает при освобождении от захвата: явно с помощью `releasePointerCapture` или автоматически, когда происходит событие `pointerup` / `pointercancel`.

Итого

События указателя позволяют одновременно обрабатывать действия с помощью мыши, касания и пера, в едином фрагменте кода.

События указателя расширяют события мыши. Мы можем заменить `mouse` на `pointer` в названиях событий и код продолжит работать для мыши, при этом получив лучшую поддержку других типов устройств.

При обработке переносов и сложных касаний, которые браузер может попытаться обработать сам, не забывайте отменять действие браузера и ставить `touch-action: none` в CSS для элементов, с которыми мы взаимодействуем.

Дополнительные возможности событий указателя:

- Поддержка мультитач с помощью `pointerId` и `isPrimary`.
- Особые свойства для определённых устройств, такие как `pressure`, `width/height` и другие.
- Захват указателя: мы можем перенаправить все события указателя на определённый элемент до наступления события `pointerup` / `pointercancel`.

На данный момент события указателя поддерживаются в основных браузерах, поэтому мы можем безопасно переходить на них, особенно если нет необходимости в поддержке IE10 и Safari 12. И даже для этих браузеров есть полифили, которые добавляют эту поддержку.

Прокрутка

Событие прокрутки `scroll` позволяет реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые при этом можно сделать.

Например:

- Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
- Подгрузить данные, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

```
window.addEventListener('scroll', function() {
  document.getElementById('showScroll').innerHTML = pageYOffset + 'px';
});
```

Событие `scroll` работает как на `window`, так и на других элементах, на которых включена прокрутка.

Предотвращение прокрутки

Как можно сделать что-то непрокручиваемым?

Нельзя предотвратить прокрутку, используя `event.preventDefault()` в обработчике `onscroll`, потому что он срабатывает *после* того, как прокрутка уже произошла.

Но можно предотвратить прокрутку, используя `event.preventDefault()` на событии, которое вызывает прокрутку, например, на событии `keydown` для клавиш `pageUp` и `pageDown`.

Если поставить на них обработчики, в которых вызвать `event.preventDefault()`, то прокрутка не начнётся.

Способов инициировать прокрутку много, поэтому более надёжный способ – использовать CSS, свойство `overflow`.

Вот несколько задач, которые вы можете решить или просмотреть, чтобы увидеть применение `onscroll`.

Задачи

Бесконечная страница

важность: 5

Создайте бесконечную страницу. Когда посетитель прокручивает её до конца, она автоматически добавляет текущие время и дату в текст (чтобы посетитель мог прокрутить ещё).

Как тут:

Прокрути меня

Date: Sun Apr 02 2023 05:50:02 GMT+0300 (Moscow Standard Time)

Date: Sun Apr 02 2023 05:50:02 GMT+0300 (Moscow Standard Time)

Date: Sun Apr 02 2023 05:50:02 GMT+0300 (Moscow Standard Time)

Date: Sun Apr 02 2023 05:50:02 GMT+0300 (Moscow Standard Time)

Пожалуйста, обратите внимание на две важные особенности прокрутки:

- 1. Прокрутка «эластична».** Можно прокрутить немного дальше начала или конца документа на некоторых браузерах/устройствах (после появляется пустое место, а затем документ автоматически «отскакивает» к нормальному состоянию).

2. Прокрутка неточна. Если прокрутить страницу до конца, можно оказаться в 0-50px от реальной нижней границы документа.

Таким образом, «прокрутка до конца» должна означать, что посетитель находится на расстоянии не более 100px от конца документа.

P.S. В реальной жизни мы можем захотеть показать «больше сообщений» или «больше товаров».

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Кнопка вверх/вниз

важность: 5

Создайте кнопку «наверх», чтобы помочь с прокруткой страницы.

Она должна работать следующим образом:

- Пока страница не прокручена вниз хотя бы на высоту окна – кнопка невидима.
- Когда страница прокручена вниз больше, чем на высоту окна – появляется стрелка «наверх» в левом верхнем углу. Если страница прокручивается назад, стрелка исчезает.
- Когда нажимается стрелка, страница прокручивается вверх.

Как тут (слева-сверху, прокрутите):

```
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

Допустим, у нас есть клиент с низкой скоростью соединения, и мы хотим сэкономить его трафик.

Для этого мы решили не показывать изображения сразу, а заменять их на «макеты», как тут:

```

```

То есть, изначально, все изображения — `placeholder.svg`. Когда страница прокручивается до того положения, где пользователь может увидеть изображение — мы меняем `src` на значение из `data-src`, и таким образом изображение загружается.

Вот пример в `iframe`:

Текст и картинки взяты с <https://wikipedia.org>.

Все изображения с `data-src` загружаются, когда становятся видимыми.

Солнечная система

Солнечная система — планетная система, включает в себя центральную звезду — Солнце — и все естественные космические объекты, врачающиеся вокруг Солнца. Она сформировалась путём гравитационного сжатия газопылевого облака примерно 4,57 млрд лет назад.

Большая часть массы объектов Солнечной системы приходится на Солнце; остальная часть содержится в восьми относительно уединённых планетах, имеющих почти круговые орбиты и располагающихся в пределах почти плоского диска — плоскости эклиптики. Общая масса системы составляет около 1,0014. При таком распределении масс особенностью кинематики

Прокрутите его, чтобы увидеть загрузку изображений «по требованию».

Требования:

- При загрузке страницы те изображения, которые уже видимы, должны загружаться сразу же, не ожидая прокрутки.
- Некоторые изображения могут быть обычными, без `data-src`. Код не должен касаться их.
- Если изображение один раз загрузилось, оно не должно больше перезагружаться при прокрутке.

P.S. Если можете, реализуйте более продвинутое решение, которое будет загружать изображения на одну страницу ниже/после текущей позиции.

P.P.S. Достаточно обрабатывать вертикальную прокрутку, горизонтальную не требуется.

[Открыть песочницу для задачи.](#)

[К решению](#)

Формы, элементы управления

Особые свойства, методы и события для работы с формами и элементами ввода: `<input>`, `<select>` и другими.

Свойства и методы формы

Формы и элементы управления, такие как `<input>`, имеют множество специальных свойств и событий.

Работать с формами станет намного удобнее, когда мы их изучим.

Навигация: формы и элементы

Формы в документе входят в специальную коллекцию `document.forms`.

Это так называемая «именованная» коллекция: мы можем использовать для получения формы как её имя, так и порядковый номер в документе.

```
document.forms.my - форма с именем "my" (name="my")
document.forms[0] - первая форма в документе
```

Когда мы уже получили форму, любой элемент доступен в именованной коллекции `form.elements`.

Например:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // получаем форму
  let form = document.forms.my; // <form name="my"> element

  // получаем элемент
  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1
</script>
```

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями `radio`.

В этом случае `form.elements[name]` является коллекцией, например:

```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы управления формы, как бы глубоко они не находились в форме, доступны в коллекции `form.elements`.

`<fieldset>` как «подформа»

Форма может содержать один или несколько элементов `<fieldset>` внутри себя. Они также поддерживают свойство `elements`, в котором находятся элементы управления внутри них.

Например:

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>

  <script>
    alert(form.elements.login); // <input name="login">

    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement

    // мы можем достать элемент по имени как из формы, так и из fieldset с ним
    alert(fieldset.elements.login == form.elements.login); // true
  </script>
</body>
```

Сокращённая форма записи: `form.name`

Есть более короткая запись: мы можем получить доступ к элементу через `form[index/name]`.

Другими словами, вместо `form.elements.login` мы можем написать `form.login`.

Это также работает, но есть небольшая проблема: если мы получаем элемент, а затем меняем его свойство `name`, то он всё ещё будет доступен под старым именем (также, как и под новым).

В этом легче разобраться на примере:

```
<form id="form">
  <input name="login">
</form>

<script>
  alert(form.elements.login == form.login); // true, ведь это одинаковые <input>

  form.login.name = "username"; // изменяем свойство name у элемента input

  // form.elements обновили свои имена:
  alert(form.elements.login); // undefined
  alert(form.elements.username); // input

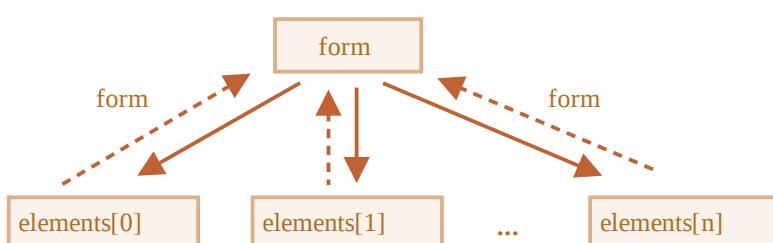
  // а в form мы можем использовать оба имени: новое и старое
  alert(form.username == form.login); // true
</script>
```

Обычно это не вызывает проблем, так как мы редко меняем имена у элементов формы.

Обратная ссылка: `element.form`

Для любого элемента форма доступна через `element.form`. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму.

Вот иллюстрация:



Пример:

```
<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

Элементы формы

Рассмотрим элементы управления, используемые в формах.

input и textarea

К их значению можно получить доступ через свойство `input.value` (строка) или `input.checked` (булево значение) для чекбоксов.

Вот так:

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

input.checked = true; // для чекбоксов и переключателей
```



⚠ Используйте `textarea.value` вместо `textarea.innerHTML`

Обратим внимание: хоть элемент `<textarea>...</textarea>` и хранит своё значение как вложенный HTML, нам не следует использовать `textarea.innerHTML` для доступа к нему.

Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

select и option

Элемент `<select>` имеет 3 важных свойства:

1. `select.options` – коллекция из подэлементов `<option>`,
2. `select.value` – значение выбранного в данный момент `<option>`,
3. `select.selectedIndex` – номер выбранного `<option>`.

Они дают три разных способа установить значение в `<select>`:

- Найти соответствующий элемент `<option>` и установить в `option.selected` значение `true`.
- Установить в `select.value` значение нужного `<option>`.
- Установить в `select.selectedIndex` номер нужного `<option>`.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе.

Вот эти способы на примере:

```
<select id="select">
  <option value="apple">Яблоко</option>
  <option value="pear">Груша</option>
  <option value="banana">Банан</option>
</select>

<script>
  // все три строки делают одно и то же
  select.options[2].selected = true;
  select.selectedIndex = 2;
  select.value = 'banana';
</script>
```

В отличие от большинства других элементов управления, `<select>` позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут `multiple`. Эту возможность используют редко, но в этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство `selected` у подэлементов `<option>`.

Их коллекцию можно получить как `select.options`, например:

```
<select id="select" multiple>
  <option value="blues" selected>Блюз</option>
  <option value="rock" selected>Рок</option>
  <option value="classic">Классика</option>
</select>

<script>
  // получаем все выбранные значения из select с multiple
  let selected = Array.from(select.options)
    .filter(option => option.selected)
    .map(option => option.value);

  alert(selected); // blues, rock
</script>
```

Полное описание элемента `<select>` доступно в спецификации <https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

new Option

Элемент `<option>` редко используется сам по себе, но и здесь есть кое-что интересное.

В спецификации есть красивый короткий синтаксис для создания элемента `<option>`:

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

Параметры:

- `text` – текст внутри `<option>`,
- `value` – значение,
- `defaultSelected` – если `true`, то ставится HTML-атрибут `selected`,
- `selected` – если `true`, то элемент `<option>` будет выбранным.

Тут может быть небольшая путаница с `defaultSelected` и `selected`. Всё просто: `defaultSelected` задаёт HTML-атрибут, его можно получить как `option.getAttribute('selected')`, а `selected` – выбрано значение или нет, именно его важно поставить правильно. Впрочем, обычно ставят оба этих значения в `true` или не ставят вовсе (т.е. `false`).

Пример:

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

Тот же элемент, но выбранный:

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

Элементы `<option>` имеют свойства:

`option.selected`

Выбрана ли опция.

`option.index`

Номер опции среди других в списке `<select>`.

`option.value`

Значение опции.

option.text

Содержимое опции (то, что видит посетитель).

Ссылки

- Спецификация: [https://html.spec.whatwg.org/multipage/forms.html ↗](https://html.spec.whatwg.org/multipage/forms.html).

Итого

Свойства для навигации по формам:

document.forms

Форма доступна через `document.forms[name/index]`.

form.elements

Элементы формы доступны через `form.elements[name/index]`, или можно просто использовать `form[name/index]`. Свойство `elements` также работает для `<fieldset>`.

element.form

Элементы хранят ссылку на свою форму в свойстве `form`.

Значения элементов формы доступны через `input.value`, `textarea.value`, `select.value` и т.д. либо `input.checked` для чекбоксов и переключателей.

Для элемента `<select>` мы также можем получить индекс выбранного пункта через `select.selectedIndex`, либо используя коллекцию пунктов `select.options`.

Это были основы для начала работы с формами. Далее в учебнике мы встретим много примеров.

В следующей главе мы рассмотрим такие события, как `focus` и `blur`, которые могут происходить на любом элементе, но чаще всего обрабатываются в формах.

✓ Задачи

Добавьте пункт к выпадающему списку

важность: 5

Имеется `<select>`:

```
<select id="genres">
  <option value="rock">Рок</option>
  <option value="blues" selected>Блюз</option>
</select>
```

Используя JavaScript:

1. Выведите значение и текст выбранного пункта.
2. Добавьте пункт: `<option value="classic">Классика</option>`.
3. Сделайте его выбранным.

[К решению](#)

Фокусировка: focus/blur

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу `Tab`. Также существует HTML-атрибут `autofocus`, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса, о них – далее.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализовать или загрузить что-нибудь.

Момент потери фокуса («`blur`») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает `Tab`, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса, о них – далее.

Потеря фокуса обычно означает «данные введены», и мы можем выполнить проверку введённых данных или даже отправить эти данные на сервер и так далее.

В работе с событиями фокусировки есть важные особенности. Мы постараемся разобрать их далее.

События focus/blur

Событие `focus` вызывается в момент фокусировки, а `blur` – когда элемент теряет фокус.

Используем их для валидации(проверки) введённых данных.

В примере ниже:

- Обработчик `blur` проверяет, введён ли `email`, и если нет – показывает ошибку.

- Обработчик `focus` скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

```

<style>
    .invalid { border-color: red; }
    #error { color: red }
</style>

Ваш email: <input type="email" id="input">

<div id="error"></div>

<script>
input.onblur = function() {
    if (!input.value.includes('@')) { // не email
        input.classList.add('invalid');
        error.innerHTML = 'Пожалуйста, введите правильный email.';
    }
};

input.onfocus = function() {
    if (this.classList.contains('invalid')) {
        // удаляем индикатор ошибки, т.к. пользователь хочет ввести данные заново
        this.classList.remove('invalid');
        error.innerHTML = "";
    }
};
</script>

```

Ваш email:

Современный HTML позволяет делать валидацию с помощью атрибутов `required`, `pattern` и т.д. Иногда – это всё, что нам нужно. JavaScript можно использовать, когда мы хотим больше гибкости. А ещё мы могли бы отправлять изменённое значение на сервер, если оно правильное.

Методы `focus/blur`

Методы `elem.focus()` и `elem.blur()` устанавливают/снимают фокус.

Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:

```

<style>
.error {
    background: red;
}
</style>

```

```
Ваш email: <input type="email" id="input">
<input type="text" style="width:280px" placeholder="введите неверный email и кликните

<script>
  input.onblur = function() {
    if (!this.value.includes('@')) { // не email
      // показать ошибку
      this.classList.add("error");
      // ...и вернуть фокус обратно
      input.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>
```

Ваш email: введите неверный email и кликните сюда

Это сработает во всех браузерах, кроме Firefox ([bug ↗](#)).

Если мы что-нибудь введём и нажмём `Tab` или кликнем в другое место, тогда `onblur` вернёт фокус обратно.

Отметим, что мы не можем «отменить потерю фокуса», вызвав `event.preventDefault()` в обработчике `onblur` потому, что `onblur` срабатывает после потери фокуса элементом.

Однако на практике следует хорошо подумать, прежде чем внедрять что-то подобное, потому что мы обычно должны показывать ошибки пользователю, но они не должны мешать пользователю при заполнении нашей формы. Ведь, вполне возможно, что он захочет сначала заполнить другие поля.

Потеря фокуса, вызванная JavaScript

Потеря фокуса может произойти по множеству причин.

Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

- `alert` переводит фокус на себя – элемент теряет фокус (событие `blur`), а когда `alert` закрывается – элемент получает фокус обратно (событие `focus`).
- Если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики `focus/blur` могут сработать тогда, когда это не требуется.

Используя эти события, нужно быть осторожным. Если мы хотим отследить потерю фокуса, которую инициировал пользователь, тогда нам следует избегать её самим.

Включаем фокусировку на любом элементе: `tabindex`

Многие элементы по умолчанию не поддерживают фокусировку.

Какие именно – зависит от браузера, но одно всегда верно: поддержка `focus/blur` гарантирована для элементов, с которыми посетитель может взаимодействовать: `<button>`, `<input>`, `<select>`, `<a>` и т.д.

С другой стороны, элементы форматирования `<div>`, ``, `<table>` – по умолчанию не могут получить фокус. Метод `elem.focus()` не работает для них, и события `focus/blur` никогда не срабатывают.

Это можно изменить HTML-атрибутом `tabindex`.

Любой элемент поддерживает фокусировку, если имеет `tabindex`. Значение этого атрибута – порядковый номер элемента, когда клавиша `Tab` (или что-то аналогичное) используется для переключения между элементами.

То есть: если у нас два элемента, первый имеет `tabindex="1"`, а второй `tabindex="2"`, то находясь в первом элементе и нажав `Tab` – мы переместимся во второй.

Порядок перебора таков: сначала идут элементы со значениями `tabindex` от `1` и выше, в порядке `tabindex`, а затем элементы без `tabindex` (например, обычный `<input>`).

При совпадающих `tabindex` элементы перебираются в том порядке, в котором идут в документе.

Есть два специальных значения:

- `tabindex="0"` ставит элемент в один ряд с элементами без `tabindex`. То есть, при переключении такие элементы будут после элементов с `tabindex ≥ 1`.
- `tabindex="-1"` позволяет фокусироваться на элементе только программно. Клавиша `Tab` проигнорирует такой элемент, но метод `elem.focus()` будет действовать.

Например, список ниже. Кликните первый пункт в списке и нажмите `Tab`:

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из iframe с примером.

```
<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 первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из iframe с примером.

- Один
- Ноль
- Два
- Минус один

Порядок такой: `1 - 2 - 0`. Обычно `` не поддерживает фокусировку, но `tabindex` включает её, а также события и стилизацию псевдоклассом `:focus`.

Свойство `elem.tabIndex` тоже работает

Мы можем добавить `tabindex` из JavaScript, используя свойство `elem.tabIndex`. Это даст тот же эффект.

События `focusin/focusout`

События `focus` и `blur` не всплывают.

Например, мы не можем использовать `onfocus` на `<form>`, чтобы подсветить её:

```
<!-- добавить класс при фокусировке на форме -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>
```



Пример выше не работает, потому что когда пользователь перемещает фокус на `<input>`, событие `focus` срабатывает только на этом элементе. Это событие не всплывает. Следовательно, `form.onfocus` никогда не срабатывает.

У этой проблемы два решения.

Первое: забавная особенность – `focus/blur` не всплывают, но передаются вниз на фазе перехвата.

Это сработает:

```
<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  // установить обработчик на фазе перехвата (последний аргумент true)
  form.addEventListener("focus", () => form.classList.add('focused'), true);
  form.addEventListener("blur", () => form.classList.remove('focused'), true);
</script>
```



Второе решение: события `focusin` и `focusout` – такие же, как и `focus/blur`, но они всплывают.

Заметьте, что эти события должны использоваться с `elem.addEventListener`, но не с `on<event>`.

Второй рабочий вариант:

```
<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  form.addEventListener("focusin", () => form.classList.add('focused'));
  form.addEventListener("focusout", () => form.classList.remove('focused'));
</script>
```



Итого

События `focus` и `blur` срабатывают на фокусировке/потере фокуса элемента.

Их особенности:

- Они не всплывают. Но можно использовать фазу перехвата или `focusin/focusout`.
 - Большинство элементов не поддерживают фокусировку по умолчанию. Используйте `tabindex`, чтобы сделать фокусируемым любой элемент.

Текущий элемент с фокусом можно получить из `document.activeElement`.

Задачи

Редактируемый div

важность: 5

Создайте `<div>`, который превращается в `<textarea>`, если на него кликнуть.

`<textarea>` позволяет редактировать HTML в элементе `<div>`.

Когда пользователь нажимает `Enter` или переводит фокус, `<textarea>` превращается обратно в `<div>`, и его содержимое становится HTML-кодом в `<div>`.

[Демо в новом окне ↗](#)

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Редактирование TD по клику

важность: 5

Сделайте ячейки таблицы редактируемыми по клику.

- По клику – ячейка должна стать «редактируемой» (`textarea` появляется внутри), мы можем изменять HTML. Изменение размера ячейки должно быть отключено.
- Кнопки ОК и ОТМЕНА появляются ниже ячейки и, соответственно, завершают/отменяют редактирование.
- Только одну ячейку можно редактировать за один раз. Пока `<td>` в «режиме редактирования», клики по другим ячейкам игнорируются.
- Таблица может иметь множество ячеек. Используйте делегирование событий.

Демо:

Кликните на ячейку таблицы, чтобы редактировать её. Нажмите ОК или ОТМЕНА, когда закончите.

Квадрат Вадиа: Направление, Элемент, Цвет, Значение

Северо-Запад Металл Серебро Старейшины	Север Вода Синий Перемены	Северо-Восток Земля Жёлтый Направление
Запад Металл Золото Молодость	Центр Всё Пурпурный Гармония	Восток Дерево Синий Будущее
Юго-Запад Земля Коричневый Спокойствие	Юг Огонь Оранжевый Слава	Юго-Восток Дерево Зеленый Роман

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

Мышь, управляемая клавиатурой

важность: 4

Установите фокус на мышь. Затем используйте клавиши со стрелками, чтобы её двигать:

[Демо в новом окне ↗](#)

P.S. Не добавляйте обработчики никуда, кроме элемента `#mouse`.

P.P.S. Не изменяйте HTML/CSS, подход должен быть общим и работать с любым элементом.

[Открыть песочницу для задачи. ↗](#)

[К решению](#)

События: change, input, cut, copy, paste

Давайте рассмотрим различные события, сопровождающие обновлению данных.

Событие: change

Событие `change` срабатывает по окончании изменения элемента.

Для текстовых `<input>` это означает, что событие происходит при потере фокуса.

Пока мы печатаем в текстовом поле в примере ниже, событие не происходит. Но когда мы перемещаем фокус в другое место, например, нажимая на кнопку, то произойдёт событие `change`:

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



Для других элементов: `select`, `input type=checkbox/radio` событие запускается сразу после изменения значения:

```
<select onchange="alert(this.value)">
```

```
<option value="">Выберите что-нибудь</option>
<option value="1">Вариант 1</option>
<option value="2">Вариант 2</option>
<option value="3">Вариант 3</option>
</select>
```

Выберите что-нибудь ▾

Событие: input

Событие `input` срабатывает каждый раз при изменении значения.

В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста.

Например:

```
<input type="text" id="input" oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```

oninput:

Если мы хотим обрабатывать каждое изменение в `<input>`, то это событие является лучшим выбором.

С другой стороны, событие `input` не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш `←`, `→` и подобных при фокусе на текстовом поле не вызовут это событие.

ⓘ Нельзя ничего предотвратить в `oninput`

Событие `input` происходит после изменения значения.

Поэтому мы не можем использовать `event.preventDefault()` там – будет уже слишком поздно, никакого эффекта не будет.

События: cut, copy, paste

Эти события происходят при вырезании/копировании/вставке данных.

Они относятся к классу [ClipboardEvent](#) и обеспечивают доступ к копируемым/вставляемым данным.

Мы также можем использовать `event.preventDefault()` для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится.

Например, код, приведённый ниже, предотвращает все подобные события и показывает, что мы пытаемся вырезать/копировать/вставить:

```
<input type="text" id="input">
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));
    return false;
  };
</script>
```



Технически, мы можем скопировать/вставить всё. Например, мы можем скопировать файл из файловой системы и вставить его.

Существует список методов [в спецификации](#) для работы с различными типами данных, чтения/записи в буфер обмена.

Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий `onclick`.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи `dispatchEvent` во всех браузерах, кроме Firefox.

Итого

События изменения данных:

Событие	Описание	Особенности
<code>change</code>	Значение было изменено.	Для текстовых полей срабатывает при потере фокуса.
<code>input</code>	Срабатывает при каждом изменении значения.	Запускается немедленно, в отличие от <code>change</code> .

Событие	Описание	Особенности
cut/copy/paste	Действия по вырезанию/копированию/вставке.	Действие можно предотвратить. Свойство <code>event.clipboardData</code> предоставляет доступ на чтение/запись в буфер обмена...

✓ Задачи

Депозитный калькулятор

Создайте интерфейс, позволяющий ввести сумму банковского вклада и процент, а затем рассчитать, какая это будет сумма через заданный промежуток времени.

Демо-версия:

Депозитный калькулятор.

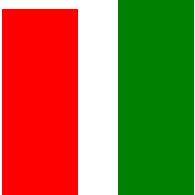
Первоначальный депозит

Срок вклада?

Годовая процентная ставка?

Было: Станет:

10000 10500



Любое изменение введённых данных должно быть обработано немедленно.

Формула:

```
// initial: начальная сумма денег
// interest: проценты, например, 0.05 означает 5% в год
// years: сколько лет ждать
let result = Math.round(initial * (1 + interest) ** years);
```

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Отправка формы: событие и метод submit

При отправке формы срабатывает событие `submit`, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод `form.submit()` позволяет инициировать отправку формы из JavaScript. Мы можем использовать его для динамического создания и отправки наших собственных форм на сервер.

Давайте посмотрим на них подробнее.

Событие: submit

Есть два основных способа отправить форму:

1. Первый – нажать кнопку `<input type="submit">` или `<input type="image">`.
2. Второй – нажать `Enter`, находясь на каком-нибудь поле.

Оба действия генерируют событие `submit` на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать `event.preventDefault()`, тогда форма не будет отправлена на сервер.

В примере ниже:

1. Перейдите в текстовое поле и нажмите `Enter`.
2. Нажмите `<input type="submit">`.

Оба действия показывают `alert` и форма не отправится благодаря `return false`:

```
<form onsubmit="alert('submit!');return false">
    Первый пример: нажмите Enter: <input type="text" value="Текст"><br>
    Второй пример: нажмите на кнопку "Отправить": <input type="submit" value="Отправить">
</form>
```

Первый пример: нажмите Enter:
Второй пример: нажмите на кнопку "Отправить":

Взаимосвязь между `submit` и `click`

При отправке формы по нажатию `Enter` в текстовом поле, генерируется событие `click` на кнопке `<input type="submit">`.

Это довольно забавно, учитывая что никакого клика не было.

Пример:

```
<form onsubmit="alert('submit!');return false">
  <input type="text" size="30" value="Установите фокус здесь и нажмите Enter">
  <input type="submit" value="Отправить" onclick="alert('click')">
</form>
```

Установите фокус здесь и нажмите Enter

Метод: `submit`

Чтобы отправить форму на сервер вручную, мы можем вызвать метод `form.submit()`.

При этом событие `submit` не генерируется. Предполагается, что если программист вызывает метод `form.submit()`, то он уже выполнил всю соответствующую обработку.

Иногда это используют для генерации формы и отправки её вручную, например так:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// перед отправкой формы, её нужно вставить в документ
document.body.append(form);

form.submit();
```

Задачи

Модальное диалоговое окно с формой

важность: 5

Создайте функцию `showPrompt(html, callback)`, которая выводит форму с сообщением (`html`), полем ввода и кнопками `OK/ОТМЕНА`.

- Пользователь должен ввести что-то в текстовое поле и нажать `Enter` или кнопку «OK», после чего должна вызываться функция `callback(value)` со значением поля.
- Если пользователь нажимает `Esc` или кнопку «ОТМЕНА», тогда вызывается `callback(null)`.

В обоих случаях нужно завершить процесс ввода и закрыть диалоговое окно с формой.

Требования:

- Форма должна быть в центре окна.
- Форма является *модальным окном*, это значит, что никакое взаимодействие с остальной частью страницы невозможно, пока пользователь не закроет его.
- При показе формы, фокус должен находиться сразу внутри `<input>`.
- Клавиши `Tab` / `Shift+Tab` должны переключать фокус между полями формы, не позволяя ему переходить к другим элементам страницы.

Пример использования:

```
showPrompt("Введите что-нибудь<br>...умное :)", function(value) {  
  alert(value);  
});
```

Демо во фрейме:

Кликните на кнопку ниже

Кликните, чтобы увидеть форму

P.S. HTML/CSS исходного кода к этой задаче содержит форму с фиксированным позиционированием, но вы должны сделать её модальной.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Загрузка документа и ресурсов

Страница: DOMContentLoaded, load, beforeunload, unload

У жизненного цикла HTML-страницы есть три важных события:

- `DOMContentLoaded` – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки `` и стили, могут быть ещё не загружены.
- `load` – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.).
- `beforeunload/unload` – пользователь покидает страницу.

Каждое из этих событий может быть полезно:

- Событие `DOMContentLoaded` – DOM готов, так что обработчик может искать DOM-узлы и инициализировать интерфейс.
- Событие `load` – внешние ресурсы были загружены, стили применены, размеры картинок известны и т.д.
- Событие `beforeunload` – пользователь покидает страницу. Мы можем проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти.
- `unload` – пользователь почти ушёл, но мы всё ещё можем запустить некоторые операции, например, отправить статистику.

Давайте рассмотрим эти события подробнее.

DOMContentLoaded

Событие `DOMContentLoaded` срабатывает на объекте `document`.

Мы должны использовать `addEventListener`, чтобы поймать его:

```
document.addEventListener("DOMContentLoaded", ready);
// не "document.onDOMContentLoaded = ..."
```

Например:

```
<script>
  function ready() {
    alert('DOM готов');

    // изображение ещё не загружено (если не было закешировано), так что размер будет
    alert(`Размер изображения: ${img.offsetWidth}x${img.offsetHeight}`);
  }

```

```
document.addEventListener("DOMContentLoaded", ready);
</script>


```

В этом примере обработчик `DOMContentLoaded` запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже ``.

Но он не дожидается, пока загрузится изображение. Поэтому `alert` покажет нулевой размер.

На первый взгляд событие `DOMContentLoaded` очень простое. DOM-дерево готово – получаем событие. Хотя тут есть несколько особенностей.

DOMContentLoaded и скрипты

Когда браузер обрабатывает HTML-документ и встречает тег `<script>`, он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM или даже дописать в него (`document.write`), так что `DOMContentLoaded` должен подождать.

Поэтому `DOMContentLoaded` определённо случится после таких скриптов:

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM готов!");
  });
</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></scri
<script>
  alert("Библиотека загружена, встроенный скрипт выполнен");
</script>
```

В примере выше мы сначала увидим «Библиотека загружена...», а затем «DOM готов!» (все скрипты выполнены).

⚠ Скрипты, которые не блокируют `DOMContentLoaded`

Есть два исключения из этого правила:

1. Скрипты с атрибутом `async`, который мы рассмотрим [немного позже](#), не блокируют `DOMContentLoaded`.
2. Скрипты, сгенерированные динамически при помощи `document.createElement('script')` и затем добавленные на страницу, также не блокируют это событие.

DOMContentLoaded и стили

Внешние таблицы стилей не затрагивают DOM, поэтому `DOMContentLoaded` их не ждёт.

Но здесь есть подводный камень. Если после стилей у нас есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
    // скрипт не выполняется, пока не загрузятся стили
    alert(getComputedStyle(document.body).marginTop);
</script>
```

Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся.

Так как `DOMContentLoaded` дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

Встроенное в браузер автозаполнение

Firefox, Chrome и Opera автоматически заполняют поля при наступлении `DOMContentLoaded`.

Например, если на странице есть форма логина и пароля и браузер запомнил значения, то при наступлении `DOMContentLoaded` он попытается заполнить их (если получил разрешение от пользователя).

Так что, если `DOMContentLoaded` откладывается из-за долгой загрузки скриптов, в свою очередь – откладывается автозаполнение. Вы наверняка замечали, что на некоторых сайтах (если вы используете автозаполнение в браузере) поля логина и пароля не заполняются мгновенно, есть некоторая задержка до полной загрузки страницы. Это и есть ожидание события `DOMContentLoaded`.

window.onload

Событие `load` на объекте `window` наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы. Это событие доступно через свойство `onload`.

В примере ниже правильно показаны размеры картинки, потому что `window.onload` дожидается всех изображений:

```
<script>
    window.onload = function() { // можно также использовать window.addEventListener('l
        alert('Страница загружена');
```

```
// к этому моменту картинка загружена
alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
};

</script>


```

window.onunload

Когда посетитель покидает страницу, на объекте `window` генерируется событие `unload`. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон.

Обычно здесь отсылают статистику.

Предположим, мы собрали данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее.

Естественно, событие `unload` – это тот момент, когда пользователь нас покидает и мы хотим сохранить эти данные.

Для этого существует специальный метод `navigator.sendBeacon(url, data)`, описанный в спецификации <https://w3c.github.io/beacon/>.

Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет `sendBeacon`.

Его можно использовать вот так:

```
let analyticsData = { /* объект с собранными данными */ };

window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

- Отсылается POST-запрос.
- Мы можем послать не только строку, но так же формы и другие форматы, как описано в главе [Fetch](#), но обычно это строковый объект.
- Размер данных ограничен 64 Кб.

К тому моменту, как `sendBeacon` завершится, браузер наверняка уже покинет страницу, так что возможности обработать ответ сервера не будет (для статистики он обычно пустой).

Для таких запросов с закрывающейся страницей есть специальный флаг `keepalive` в методе `fetch` для общих сетевых запросов. Вы можете найти больше информации в главе [Fetch API](#).

Если мы хотим отменить переход на другую страницу, то здесь мы этого сделать не сможем. Но сможем в другом месте – в событии `onbeforeunload`.

window.onbeforeunload

Если посетитель собирается уйти со страницы или закрыть окно, обработчик `beforeunload` попросит дополнительное подтверждение.

Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.

Вы можете попробовать это, запустив следующий код и затем перезагрузив страницу:

```
window.onbeforeunload = function() {
  return false;
};
```

По историческим причинам возврат непустой строки так же считается отменой события. Когда-то браузеры использовали её в качестве сообщения, но, как указывает [современная спецификация ↗](#), они не должны этого делать.

Вот пример:

```
window.onbeforeunload = function() {
  return "Есть несохранённые изменения. Всё равно уходим?";
};
```

Поведение было изменено, потому что некоторые веб-разработчики злоупотребляли этим обработчиком события, показывая вводящие в заблуждение и надоедливые сообщения. Так что, прямо сейчас старые браузеры всё ещё могут показывать строку как сообщение, но в остальных – нет возможности настроить показ сообщения пользователям.

readyState

Что произойдёт, если мы установим обработчик `DOMContentLoaded` после того, как документ загрузился?

Естественно, он никогда не запустится.

Есть случаи, когда мы не уверены, готов документ или нет. Мы бы хотели, чтобы наша функция исполнилась, когда DOM загрузился, будь то сейчас или позже.

Свойство `document.readyState` показывает нам текущее состояние загрузки.

Есть три возможных значения:

- "loading" – документ загружается.
- "interactive" – документ был полностью прочитан.
- "complete" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Так что мы можем проверить `document.readyState` и, либо установить обработчик, либо, если документ готов, выполнить код сразу же.

Например, вот так:

```
function work() { /*...*/ }

if (document.readyState == 'loading') {
  // ещё загружается, ждём события
  document.addEventListener('DOMContentLoaded', work);
} else {
  // DOM готов!
  work();
}
```

Также есть событие `readystatechange`, которое генерируется при изменении состояния, так что мы можем вывести все эти состояния таким образом:

```
// текущее состояние
console.log(document.readyState);

// вывести изменения состояния
document.addEventListener('readystatechange', () => console.log(document.readyState))
```

Событие `readystatechange` – альтернативный вариант отслеживания состояния загрузки документа, который появился очень давно. На сегодняшний день он используется редко.

Для полноты картины давайте посмотрим на весь поток событий:

Здесь документ с `<iframe>`, `` и обработчиками, которые логируют события:

```
<script>
  log('начальный readyState:' + document.readyState);

  document.addEventListener('readystatechange', () => log('readyState:' + document.readyState));
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));

  window.onload = () => log('window onload');
</script>
```

```
<iframe src="iframe.html" onload="log('iframe onload')"></iframe>


<script>
  img.onload = () => log('img onload');
</script>
```

Рабочий пример есть [в песочнице](#).

Типичный вывод:

1. [1] начальный readyState:loading
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

Цифры в квадратных скобках обозначают примерное время события. События, отмеченные одинаковой цифрой, произойдут примерно в одно и то же время (\pm несколько миллисекунд).

- `document.readyState` станет `interactive` прямо перед `DOMContentLoaded`. Эти две вещи, на самом деле, обозначают одно и то же.
- `document.readyState` станет `complete`, когда все ресурсы (`iframe` и `img`) загрузятся. Здесь мы видим, что это произойдёт примерно в одно время с `img.onload` (`img` последний ресурс) и `window.onload`. Переключение на состояние `complete` означает то же самое, что и `window.onload`. Разница заключается в том, что `window.onload` всегда срабатывает после всех `load` других обработчиков.

Итого

События загрузки страницы:

- `DOMContentLoaded` генерируется на `document`, когда DOM готов. Мы можем применить JavaScript к элементам на данном этапе.
 - Скрипты, вроде `<script>...</script>` или `<script src="..."></script>` блокируют `DOMContentLoaded`, браузер ждёт, пока они выполняются.
 - Изображения и другие ресурсы тоже всё ещё могут продолжать загружаться.

- Событие `load` на `window` генерируется, когда страница и все ресурсы загружены. Мы редко его используем, потому что обычно нет нужды ждать так долго.
- Событие `beforeunload` на `window` генерируется, когда пользователь покидает страницу. Если мы отменим событие, браузер спросит, на самом деле пользователь хочет уйти (например, у нас есть несохранённые изменения).
- Событие `unload` на `window` генерируется, когда пользователь окончательно уходит, в обработчике мы можем делать только простые вещи, которые ни о чём не спрашивают пользователя и не заставляют его ждать. Из-за этих ограничений оно редко используется. Мы можем послать сетевой запрос с помощью `navigator.sendBeacon`.
- `document.readyState` – текущее состояние документа, изменения можно отследить с помощью события `readystatechange`:
 - `loading` – документ грузится.
 - `interactive` – документ прочитан, происходит примерно в то же время, что и `DOMContentLoaded`, но до него.
 - `complete` – документ и ресурсы загружены, происходит примерно в то же время, что и `window.onload`, но до него.

Скрипты: `async`, `defer`

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются.

Когда браузер загружает HTML и доходит до тега `<script>...</script>`, он не может продолжать строить DOM. Он должен сначала выполнить скрипт. То же самое происходит и с внешними скриптами `<script src="..."></script>`: браузер должен подождать, пока загрузится скрипт, выполнить его, и только затем обработать остальную страницу.

Это ведёт к двум важным проблемам:

- Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
- Если вверху страницы объёмный скрипт, он «блокирует» страницу.

Пользователи не видят содержимое страницы, пока он не загрузится и не запустится:

```
<p>...содержимое перед скриптом...</p>

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></sc
<!-- Это не отобразится, пока скрипт не загрузится -->
<p>...содержимое после скрипта...</p>
```

Конечно, есть пути, как это обойти. Например, мы можем поместить скрипт внизу страницы. Тогда он сможет видеть элементы над ним и не будет препятствовать отображению содержимого страницы:

```
<body>
  ...всё содержимое над скриптом...
  <script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
</body>
```

Но это решение далеко от идеального. Например, браузер замечает скрипт (и может начать загружать его) только после того, как он полностью загрузил HTML-документ. В случае с длинными HTML-страницами это может создать заметную задержку.

Такие вещи незаметны людям, у кого очень быстрое соединение, но много кто в мире имеет медленное подключение к интернету или использует не такой хороший мобильный интернет.

К счастью, есть два атрибута тега `<script>`, которые решают нашу проблему: `defer` и `async`.

defer

Атрибут `defer` сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда DOM дерево будет полностью построено.

Вот тот же пример, что и выше, но с `defer`:

```
<p>...содержимое перед скриптом...</p>
<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1">
<!-- отображается сразу же -->
<p>...содержимое после скрипта...</p>
```

- Скрипты с `defer` никогда не блокируют страницу.
- Скрипты с `defer` всегда выполняются, когда дерево DOM готово, но до события `DOMContentLoaded`.

Следующий пример это показывает:

```
<p>...содержимое до скрипта...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("Дерево DOM готово после выполнения скрипта"))
</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...содержимое после скрипта...</p>
```

1. Содержимое страницы отобразится мгновенно.
2. Событие `DOMContentLoaded` подождёт отложенный скрипт. Оно будет сгенерировано, только когда скрипт (2) будет загружен и выполнен.

Отложенные с помощью `defer` скрипты сохраняют порядок относительно друг друга, как и обычные скрипты.

Поэтому, если сначала загружается большой скрипт, а затем меньшего размера, то последний будет ждать.

```
<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>
<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>
```

i Маленький скрипт загрузится первым, но выполнится вторым

Браузеры сканируют страницу на предмет скриптов и загружают их параллельно в целях увеличения производительности. Поэтому и в примере выше оба скрипта скачиваются параллельно. `small.js` скорее всего загрузится первым.

Но спецификация требует последовательного выполнения скриптов согласно порядку в документе, поэтому он подождёт выполнения `long.js`.

i Атрибут `defer` предназначен только для внешних скриптов

Атрибут `defer` будет проигнорирован, если в теге `<script>` нет `src`.

async

Атрибут `async` означает, что скрипт абсолютно независим:

- Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается.
- Событие `DOMContentLoaded` и асинхронные скрипты не ждут друг друга:

- `DOMContentLoaded` может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет готова),
- ...так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)
- Остальные скрипты не ждут `async`, и скрипты с `async` не ждут другие скрипты.

Так что если у нас есть несколько скриптов с `async`, они могут выполняться в любом порядке. То, что первое загрузится – запустится в первую очередь:

```
<p>...содержимое перед скриптами...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("DOM готов!"));
</script>

<script async src="https://javascript.info/article/script-async-defer/long.js"></script>
<script async src="https://javascript.info/article/script-async-defer/small.js"></script>

<p>...содержимое после скриптов...</p>
```

1. Содержимое страницы отображается сразу же : `async` его не блокирует.
2. `DOMContentLoaded` может произойти как до, так и после `async`, никаких гарантий нет.
3. Асинхронные скрипты не ждут друг друга. Меньший скрипт `small.js` идёт вторым, но скорее всего загрузится раньше `long.js`, поэтому и запустится первым. То есть, скрипты выполняются в порядке загрузки.

Асинхронные скрипты очень полезны для добавления на страницу сторонних скриптов: счётчиков, рекламы и т.д. Они не зависят от наших скриптов, и мы тоже не должны ждать их:

```
<!-- Типичное подключение скрипта Google Analytics -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

Динамически загружаемые скрипты

Мы можем также добавить скрипт и динамически, с помощью JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

Скрипт начнёт загружаться, как только он будет добавлен в документ (*).

Динамически загружаемые скрипты по умолчанию ведут себя как «async».

То есть:

- Они никого не ждут, и их никто не ждёт.
- Скрипт, который загружается первым – запускается первым (в порядке загрузки).

Мы можем изменить относительный порядок скриптов с «первый загрузился – первый выполнился» на порядок, в котором они идут в документе (как в обычных скриптах) с помощью явной установки свойства `async` в `false`:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";

script.async = false;

document.body.append(script);
```

Например, здесь мы добавляем два скрипта. Без `script.async=false` они запускались бы в порядке загрузки (`small.js` скорее всего запустился бы раньше). Но с этим флагом порядок будет как в документе:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}

// long.js запускается первым, так как async=false
loadScript("/article/script-async-defer/long.js");
loadScript("/article/script-async-defer/small.js");
```

Итого

У `async` и `defer` есть кое-что общее: они не блокируют отрисовку страницы. Так что пользователь может просмотреть содержимое страницы и ознакомиться с ней сразу же.

Но есть и значимые различия:

Порядок

DOMContentLoaded

Порядок	DOMContentLoaded
async	Порядок загрузки (кто загрузится первым, тот и сработает).
defer	Порядок документа (как расположены в документе).

⚠ Страница без скриптов должна быть рабочей

Пожалуйста, помните, что когда вы используете `defer`, страница видна до того, как скрипт загрузится.

Пользователь может знакомиться с содержимым страницы, читать её, но графические компоненты пока отключены.

Поэтому обязательно должна быть индикация загрузки, нерабочие кнопки – отключены с помощью CSS или другим образом. Чтобы пользователь явно видел, что уже готово, а что пока нет.

На практике `defer` используется для скриптов, которым требуется доступ ко всему DOM и/или важен их относительный порядок выполнения.

А `async` хорош для независимых скриптов, например счётчиков и рекламы, относительный порядок выполнения которых не играет роли.

Загрузка ресурсов: onload и onerror

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ifреймов, изображений и др.

Для этого существуют два события:

- `load` – успешная загрузка,
- `error` – во время загрузки произошла ошибка.

Загрузка скриптов

Допустим, нам нужно загрузить сторонний скрипт и вызвать функцию, которая объявлена в этом скрипте.

Мы можем загрузить этот скрипт динамически:

```
let script = document.createElement('script');
script.src = "my.js";
```

```
document.head.append(script);
```

...Но как нам вызвать функцию, которая объявлена внутри того скрипта? Нам нужно подождать, пока скрипт загрузится, и только потом мы можем её вызвать.

i На заметку:

Для наших собственных скриптов мы можем использовать [JavaScript-модули](#), но они не слишком широко распространены в сторонних библиотеках.

script.onload

Главный помощник – это событие `load`. Оно срабатывает после того, как скрипт был загружен и выполнен.

Например:

```
let script = document.createElement('script');

// мы можем загрузить любой скрипт с любого домена
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
document.head.append(script);

script.onload = function() {
    // в скрипте создаётся вспомогательная переменная с именем "_"
    alert(_.VERSION); // отображает версию библиотеки
};
```

Таким образом, в обработчике `onload` мы можем использовать переменные, вызывать функции и т.д., которые предоставляет нам сторонний скрипт.

...А что если во время загрузки произошла ошибка? Например, такого скрипта нет (ошибка 404), или сервер был недоступен.

script.onerror

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события `error`.

Например, давайте запросим скрипт, которого не существует:

```
let script = document.createElement('script');
script.src = "https://example.com/404.js"; // такого файла не существует
document.head.append(script);

script.onerror = function() {
    alert("Ошибка загрузки " + this.src); // Ошибка загрузки https://example.com/404.js
};
```

Обратите внимание, что мы не можем получить описание HTTP-ошибки. Мы не знаем, была ли это ошибка 404 или 500, или какая-то другая. Знаем только, что во время загрузки произошла ошибка.

⚠️ Важно:

Обработчики `onload` / `onerror` отслеживают только сам процесс загрузки.

Ошибки обработки и выполнения загруженного скрипта ими не отслеживаются. Чтобы «поймать» ошибки в скрипте, нужно воспользоваться глобальным обработчиком `window.onerror`.

Другие ресурсы

События `load` и `error` также срабатывают и для других ресурсов, а вообще, для любых ресурсов, у которых есть внешний `src`.

Например:

```
let img = document.createElement('img');
img.src = "https://js.cx/clipart/train.gif"; // (*)

img.onload = function() {
  alert(`Изображение загружено, размеры ${img.width}x${img.height}`);
};

img.onerror = function() {
  alert("Ошибка во время загрузки изображения");
};
```

Однако есть некоторые особенности:

- Большинство ресурсов начинают загружаться после их добавления в документ. За исключением тега ``. Изображения начинают загружаться, когда получают `src` (*).
- Для `<iframe>` событие `load` срабатывает по окончании загрузки как в случае успеха, так и в случае ошибки.

Такое поведение сложилось по историческим причинам.

Ошибка в скрипте с другого источника

Есть правило: скрипты с одного сайта не могут получить доступ к содержимому другого сайта. Например, скрипт с `https://facebook.com` не может прочитать почту пользователя на `https://gmail.com`.

Или, если быть более точным, один источник (домен/порт/протокол) не может получить доступ к содержимому с другого источника. Даже поддомен или просто другой порт будут считаться разными источниками, не имеющими доступа друг к другу.

Это правило также касается ресурсов с других доменов.

Если мы используем скрипт с другого домена, и в нем имеется ошибка, мы не сможем узнать детали этой ошибки.

Для примера давайте возьмём мини-скрипт `error.js`, который состоит из одного-единственного вызова функции, которой не существует:

```
// error.js
noSuchFunction();
```

Теперь загрузим этот скрипт с того же сайта, на котором он лежит:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(`#${message}\n#${url}, ${line}:${col}`);
};
</script>
<script src="/article/onload-onerror/crossorigin/error.js"></script>
```

Мы видим нормальный отчёт об ошибке:

```
Uncaught ReferenceError: noSuchFunction is not defined
https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1
```

А теперь загрузим этот же скрипт с другого домена:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(`#${message}\n#${url}, ${line}:${col}`);
};
</script>
<script src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>
```

Отчёт отличается:

```
Script error.
, 0:0
```

Детали отчёта могут варьироваться в зависимости от браузера, но основная идея остаётся неизменной: любая информация о внутреннем устройстве скрипта, включая стек ошибки, спрятана. Именно потому, что скрипт загружен с другого домена.

Зачем нам могут быть нужны детали ошибки?

Существует много сервисов (и мы можем сделать наш собственный), которые обрабатывают глобальные ошибки при помощи `window.onerror`, сохраняют отчёт о них и предоставляют доступ к этому отчёту для анализа. Это здорово, потому что мы можем увидеть реальные ошибки, которые случились у наших пользователей. Но если скрипт – с другого домена, то информации об ошибках в нём почти нет, как мы только что видели.

Похожая кросс-доменная политика (CORS) внедрена и в отношении других ресурсов.

Чтобы разрешить кросс-доменный доступ, нам нужно поставить тегу `<script>` атрибут `crossorigin`, и, кроме того, удалённый сервер должен поставить специальные заголовки.

Существует три уровня кросс-доменного доступа:

1. Атрибут `crossorigin` отсутствует – доступ запрещён.
2. `crossorigin="anonymous"` – доступ разрешён, если сервер отвечает с заголовком `Access-Control-Allow-Origin` со значениями `*` или наш домен. Браузер не отправляет авторизационную информацию и куки на удалённый сервер.
3. `crossorigin="use-credentials"` – доступ разрешён, если сервер отвечает с заголовками `Access-Control-Allow-Origin` со значением наш домен и `Access-Control-Allow-Credentials: true`. Браузер отправляет авторизационную информацию и куки на удалённый сервер.

❶ На заметку:

Почитать больше о кросс-доменных доступах вы можете в главе [Fetch: запросы на другие сайты](#). Там описан метод `fetch` для сетевых запросов, но политика там точно такая же.

Такое понятие как «куки» (cookies) не рассматривается в текущей главе, но вы можете почитать о них в главе [Куки, `document.cookie`](#).

В нашем случае атрибут `crossorigin` отсутствовал. Поэтому кросс-доменный доступ был запрещён. Давайте добавим его.

Мы можем выбрать `"anonymous"` (куки не отправляются, требуется один серверный заголовок) или `"use-credentials"` (куки отправляются, требуются два серверных заголовка) в качестве значения атрибута.

Если куки нас не волнуют, тогда смело выбираем "anonymous" :

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
  alert(` ${message}\n${url}, ${line}:${col}`);
};
</script>
<script crossorigin="anonymous" src="https://cors.javascript.info/article/onload-oner
```

Теперь при условии, что сервер предоставил заголовок `Access-Control-Allow-Origin`, всё хорошо. У нас есть полный отчёт по ошибкам.

Итого

Изображения ``, внешние стили, скрипты и другие ресурсы предоставляют события `load` и `error` для отслеживания загрузки:

- `load` срабатывает при успешной загрузке,
- `error` срабатывает при ошибке загрузки.

Единственное исключение – это `<iframe>`: по историческим причинам срабатывает всегда `load` вне зависимости от того, как завершилась загрузка, даже если страница не была найдена.

Событие `readystatechange` также работает для ресурсов, но используется редко, потому что события `load/error` проще в использовании.

✓ Задачи

Загрузите изображения с колбэком

важность: 4

Обычно изображения начинают загружаться в момент их создания. Когда мы добавляем `` на страницу, пользователь не увидит его тут же. Браузер сначала должен его загрузить.

Чтобы показать изображение сразу, мы можем создать его «заранее»:

```
let img = document.createElement('img');
img.src = 'my.jpg';
```

Браузер начнёт загружать изображение и положит его в кеш. Позже, когда такое же изображение появится в документе (не важно как), оно будет показано мгновенно.

Создайте функцию `preloadImages(sources, callback)`, которая загружает все изображения из массива `sources` и, когда все они будут загружены, вызывает `callback`.

В данном примере будет показан `alert` после загрузки всех изображений.

```
function loaded() {
  alert("Изображения загружены")
}

preloadImages(["1.jpg", "2.jpg", "3.jpg"], loaded);
```

В случае ошибки функция должна считать изображение «загруженным».

Другими словами, `callback` выполняется в том случае, когда все изображения либо загружены, либо в процессе их загрузки возникла ошибка.

Такая функция полезна, например, когда нам нужно показать галерею с маленькими скролящимися изображениями, и мы хотим быть уверены, что все из них загружены.

В песочнице подготовлены ссылки к тестовым изображениям, а также код для проверки их загрузки. Код должен выводить `300`.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Разное

MutationObserver: наблюдатель за изменениями

`MutationObserver` – это встроенный объект, наблюдающий за DOM-элементом и запускающий колбэк в случае изменений.

Сначала мы познакомимся с синтаксисом, а затем разберём примеры использования.

Синтаксис

`MutationObserver` очень прост в использовании.

Сначала мы создаём наблюдатель за изменениями с помощью колбэк-функции:

```
let observer = new MutationObserver(callback);
```

Потом прикрепляем его к DOM-узлу:

```
observer.observe(node, config);
```

`config` – это объект с булевыми параметрами «на какие изменения реагировать»:

- `childList` – изменения в непосредственных детях `node`,
- `subtree` – во всех потомках `node`,
- `attributes` – в атрибутах `node`,
- `attributeFilter` – массив имён атрибутов, чтобы наблюдать только за выбранными.
- `characterData` – наблюдать ли за `node.data` (текстовое содержимое),

И ещё пара опций:

- `characterData oldValue` – если `true`, будет передавать и старое, и новое значение `node.data` в колбэк (см далее), иначе только новое (также требуется опция `characterData`),
- `attribute oldValue` – если `true`, будет передавать и старое, и новое значение атрибута в колбэк (см далее), иначе только новое (также требуется опция `attributes`).

Затем, после изменений, выполняется `callback`, в который изменения передаются первым аргументом как список объектов [MutationRecord](#) ↗, а сам наблюдатель идёт вторым аргументом.

Объекты [MutationRecord](#) ↗ имеют следующие свойства:

- `type` – тип изменения, один из:
 - `"attributes"` изменён атрибут,
 - `"characterData"` изменены данные `elem.data`, это для текстовых узлов
 - `"childList"` добавлены/удалены дочерние элементы,
- `target` – где произошло изменение: элемент для `"attributes"`, текстовый узел для `"characterData"` или элемент для `"childList"`,
- `addedNodes/removedNodes` – добавленные/удалённые узлы,
- `previousSibling/nextSibling` – предыдущий или следующий одноуровневый элемент для добавленных/удалённых элементов,
- `attributeName/attributeNamespace` – имя/пространство имён (для XML) изменённого атрибута,

- `oldValue` – предыдущее значение, только для изменений атрибута или текста, если включена соответствующая опция `attribute oldValue / characterData oldValue`.

Для примера возьмём `<div>` с атрибутом `contentEditable`. Этот атрибут позволяет нам сфокусироваться на элементе, например, кликнув, и отредактировать содержимое.

```
<div contentEditable id="elem">Отредактируй <b>меня</b>, пожалуйста</div>

<script>
let observer = new MutationObserver(mutationRecords => {
  console.log(mutationRecords); // console.log(изменения)
});

// наблюдать за всем, кроме атрибутов
observer.observe(elem, {
  childList: true, // наблюдать за непосредственными детьми
  subtree: true, // и более глубокими потомками
  characterDataOldValue: true // передавать старое значение в колбэк
});
</script>
```

Теперь, если мы изменим текст внутри `меня`, мы получим единичное изменение:

```
mutationRecords = [
  {
    type: "characterData",
    oldValue: "меня",
    target: <text node>,
    // другие свойства пусты
  }
];
```

Если мы выберем или удалим `меня` полностью, мы получим сразу несколько изменений:

```
mutationRecords = [
  {
    type: "childList",
    target: <div#elem>,
    removedNodes: [<b>],
    nextSibling: <text node>,
    previousSibling: <text node>
    // другие свойства пусты
  },
  {
    type: "characterData"
    target: <text node>
    // ...детали изменений зависят от того, как браузер обрабатывает такое удаление
  }
];
```

```
// он может соединить два соседних текстовых узла "Отредактируй " и ", пожалуйста"  
// или может оставить их разными текстовыми узлами  
}];
```

Так что, `MutationObserver` позволяет реагировать на любые изменения в DOM-дереве.

Использование для интеграции

Когда это может быть нужно?

Представим ситуацию, когда вы подключаете сторонний скрипт, который добавляет какую-то полезную функциональность на страницу, но при этом делает что-то лишнее, например, показывает рекламу `<div class="ads">Ненужная реклама</div>`.

Разумеется, сторонний скрипт не даёт каких-то механизмов её убрать.

Используя `MutationObserver`, мы можем отследить, когда в нашем DOM появится такой элемент и удалить его. А полезную функциональность оставить. Хотя, конечно, создатели стороннего скрипта вряд ли обрадуются, что вы их полезный скрипт взяли, а рекламу удалили.

Есть и другие ситуации, когда сторонний скрипт добавляет что-то в наш документ, и мы хотели бы отследить, когда это происходит, чтобы адаптировать нашу страницу, динамически поменять какие-то размеры и т.п.

`MutationObserver` для этого как раз отлично подходит.

Использование для архитектуры

Есть и ситуации, когда `MutationObserver` хорошо подходит с архитектурной точки зрения.

Представим, что мы создаём сайт о программировании. Естественно, статьи на нём и другие материалы могут содержать фрагменты с исходным кодом.

Такой фрагмент в HTML-разметке выглядит так:

```
...  
<pre class="language-javascript"><code>  
 // вот код  
 let hello = "world";  
</code></pre>  
...
```

Также на нашем сайте мы будем использовать JavaScript-библиотеку для подсветки синтаксиса, например [Prism.js](#). Вызов метода

`Prism.highlightElem(pre)` ищет такие элементы `pre` и добавляет в них стили и теги, которые в итоге дают цветную подсветку синтаксиса, подобно той, которую вы видите в примерах здесь, на этой странице.

Когда конкретно нам вызвать этот метод подсветки? Можно по событию `DOMContentLoaded` или просто внизу страницы написать код, который будет искать все `pre[class*="language"]` и вызывать `Prism.highlightElem` для них:

```
// выделить все примеры кода на странице
document.querySelectorAll('pre[class*="language"]').forEach(Prism.highlightElem);
```

Пока всё просто, правда? В HTML есть фрагменты кода в `<pre>`, и для них мы включаем подсветку синтаксиса.

Идём дальше. Представим, что мы собираемся динамически подгружать материалы с сервера. Позже в учебнике мы изучим [способы для этого](#). На данный момент имеет значение только то, что мы получаем HTML-статью с веб-сервера и показываем её по запросу:

```
let article = /* получить новую статью с сервера */
articleElem.innerHTML = article;
```

HTML подгруженной статьи `article` может содержать примеры кода. Нам нужно вызвать `Prism.highlightElem` для них, чтобы подсветить синтаксис.

Кто и когда должен вызывать `Prism.highlightElem` для динамически загруженной статьи?

Мы можем добавить этот вызов к коду, который загружает статью, например, так:

```
let article = /* получить новую статью с сервера */
articleElem.innerHTML = article;

let snippets = articleElem.querySelectorAll('pre[class*="language-"]');
snippets.forEach(Prism.highlightElem);
```

...Но представьте, что у нас есть много мест в коде, где мы загружаем что-либо: статьи, опросы, посты форума. Нужно ли нам в каждый такой вызов добавлять `Prism.highlightElem`? Получится не очень удобно, да и можно легко забыть сделать это.

А что, если содержимое загружается вообще сторонним кодом? Например, у нас есть форум, написанный другим человеком, загружающий содержимое

динамически, и нам захотелось добавить к нему выделение синтаксиса. Никто не любит править чужие скрипты.

К счастью, есть другой вариант.

Мы можем использовать `MutationObserver`, чтобы автоматически определять момент, когда примеры кода появляются на странице, и подсвечивать их.

Тогда вся функциональность для подсветки синтаксиса будет в одном месте, а мы будем избавлены от необходимости интегрировать её.

Пример динамической подсветки синтаксиса

Вот работающий пример.

Если вы запустите этот код, он начнёт наблюдать за элементом ниже, подсвечивая код любого примера, который появляется там:

```
let observer = new MutationObserver(mutations => {

    for(let mutation of mutations) {
        // проверим новые узлы, есть ли что-то, что надо подсветить?

        for(let node of mutation.addedNodes) {
            // отслеживаем только узлы-элементы, другие (текстовые) пропускаем
            if (!(node instanceof HTMLElement)) continue;

            // проверить, не является ли вставленный элемент примером кода
            if (node.matches('pre[class*="language-"]')) {
                Prism.highlightElement(node);
            }

            // или, может быть, пример кода есть в его поддереве?
            for(let elem of node.querySelectorAll('pre[class*="language-"]')) {
                Prism.highlightElement(elem);
            }
        }
    }
});

let demoElem = document.getElementById('highlight-demo');

observer.observe(demoElem, {childList: true, subtree: true});
```

Ниже находится HTML-элемент и JavaScript, который его динамически заполнит примером кода через `innerHTML`.

Пожалуйста, запустите предыдущий код (он наблюдает за этим элементом), а затем код, расположенный ниже. Вы увидите как `MutationObserver` обнаружит и подсветит фрагменты кода.

Демо-элемент с `id="highlight-demo"`, за которым следует код примера выше.

```
let demoElem = document.getElementById('highlight-demo');

// динамически вставить содержимое как фрагменты кода
demoElem.innerHTML = `Фрагмент кода ниже:
<pre class="language-javascript"><code> let hello = "world!"; </code></pre>
<div>Ещё один:</div>
<div>
  <pre class="language-css"><code>.class { margin: 5px; } </code></pre>
</div>
`;
```

Теперь у нас есть `MutationObserver`, который может отслеживать вставку кода в наблюдаемых элементах или во всём документе. Мы можем добавлять/удалять фрагменты кода в HTML, не задумываясь об их подсветке.

Дополнительные методы

Метод, останавливающий наблюдение за узлом:

- `observer.disconnect()` – останавливает наблюдение.

Вместе с ним используют метод:

- `mutationRecords = observer.takeRecords()` – получает список необработанных записей изменений, которые произошли, но колбэк для них ещё не выполнился.

```
// мы отключаем наблюдатель
observer.disconnect();

// он, возможно, не успел обработать некоторые изменения
let mutationRecords = observer.takeRecords();
// обработать mutationRecords
```

Сборка мусора

Объекты `MutationObserver` используют внутри себя так называемые [«слабые ссылки»](#) на узлы, за которыми смотрят. Так что если узел удалён из DOM и больше не достижим, то он будет удалён из памяти вне зависимости от наличия наблюдателя.

Итого

`MutationObserver` может реагировать на изменения в DOM: атрибуты, добавленные/удалённые элементы, текстовое содержимое.

Мы можем использовать его, чтобы отслеживать изменения, производимые другими частями нашего собственного кода, а также интегрироваться со сторонними библиотеками.

`MutationObserver` может отслеживать любые изменения. Разные опции конфигурации «что наблюдать» предназначены для оптимизации, чтобы не тратить ресурсы на лишние вызовы колбэка.

Selection и Range

В этой главе мы рассмотрим выделение как в документе, так и в полях формы, таких как `<input>`.

JavaScript позволяет получать существующее выделение, выделять и снимать выделение как целиком, так и по частям, убирать выделенную часть из документа, оборачивать её в тег и так далее.

Вы можете получить готовые решения в секции «Итого» в конце статьи, но узнаете гораздо больше, если прочитаете главу целиком. Используемые для выделения встроенные классы `Range` и `Selection` просты для понимания, и после их изучения вам уже не понадобятся «готовые рецепты», чтобы сделать всё, что захотите.

Range

В основе выделения лежит `Range ↗` – диапазон. Он представляет собой пару «границных точек»: начало и конец диапазона.

Каждая точка представлена как родительский DOM-узел с относительным смещением от начала. Если этот узел – DOM-элемент, то смещение – это номер дочернего элемента, а для текстового узла смещение – позиция в тексте. Скоро будут примеры.

Давайте что-нибудь выделим.

Для начала мы создадим диапазон (конструктор не имеет параметров):

```
let range = new Range();
```

Затем мы установим границы выделения, используя `range.setStart(node, offset)` и `range.setEnd(node, offset)`.

Например, рассмотрим этот фрагмент HTML-кода:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

Взглянем на его DOM-структуру, обратите внимание на текстовые узлы, они важны для нас:



Выделим "Example: <i>italic</i>" . Это первые два дочерних узла тега `<p>` (учитывая текстовые узлы):

```
<p>Example: <i>italic</i> and <b>bold</b></p>
```

A horizontal timeline from 0 to 3. The first two segments (0 to 1 and 1 to 2) are highlighted in orange, representing the selected range. The text 'Example:' is at index 0, '*italic*' is at index 1, and 'and' is at index 2.

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

```
<script>
```

```
let range = new Range();
```

```
range.setStart(p, 0);  
range.setEnd(p, 2);
```

```
// toString, вызванный у экземпляра Range, возвращает его содержимое в виде текста  
alert(range); // Example: italic
```

```
// применим этот диапазон к выделению документа (объясняется далее)  
document.getSelection().addRange(range);
```

```
</script>
```

- `range.setStart(p, 0)` – устанавливает начало диапазона на нулевом дочернем элементе тега `<p>` (Это текстовый узел "Example: ").
- `range.setEnd(p, 2)` – расширяет диапазон до 2го (но не включая его) дочернего элемента тега `<p>` (это текстовый узел " and " , но так как конец не включён, последний включённый узел – это тег `<i>`).

Ниже представлен расширенный пример, в котором вы можете попробовать другие варианты:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

From <input id="start" type="number" value=1> – To <input id="end" type="number" value=4>
<button id="button">Click to select</button>
<script>
  button.onclick = () => {
    let range = new Range();

    range.setStart(p, start.value);
    range.setEnd(p, end.value);

    // применим выделение, объясняется далее
    document.getSelection().removeAllRanges();
    document.getSelection().addRange(range);
  };
</script>
```

Example: **italic** and **bold**

From – To Click to select

К примеру, выделение с **1** до **4** возвращает следующий диапазон

italic and **bold**.

<p>Example: <i>italic</i> and bold</p>



Не обязательно использовать один и тот же элемент в `setStart` и `setEnd`.

Диапазон может охватывать множество не связанных между собой элементов.
Важно лишь чтобы конец шёл после начала.

Выделение частей текстовых узлов

Давайте выделим текст частично, как показано ниже:

<p>Example: <i>italic</i> and bold</p>



Это также возможно, нужно просто установить начало и конец как относительное смещение в текстовых узлах.

Нам нужно создать диапазон, который:

- начинается со второй позиции первого дочернего узла тега `<p>` (захватываем всё, кроме первых двух букв "Example: ")
- заканчивается на 3 позиции первого дочернего узла тега `` (захватываем первые три буквы «bold», но не более):

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

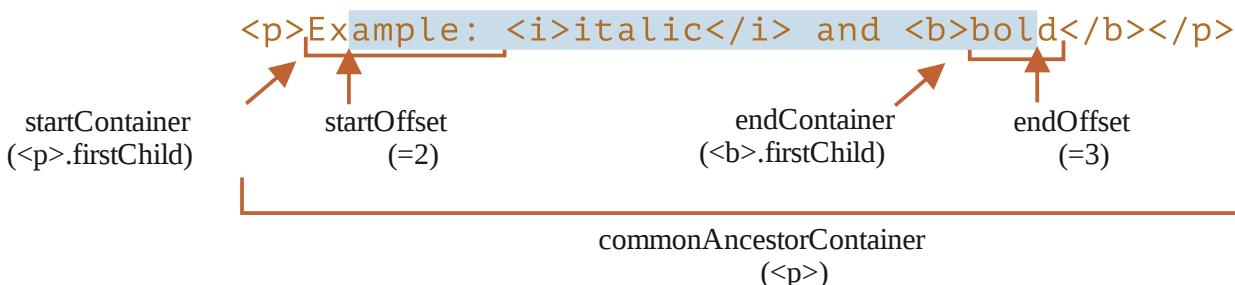
<script>
  let range = new Range();

  range.setStart(p.firstChild, 2);
  range.setEnd(p.querySelector('b').firstChild, 3);

  alert(range); // ample: italic and bol

  // применим выделение к документу (объясняется далее)
  window.getSelection().addRange(range);
</script>
```

Объект диапазона Range имеет следующие свойства:



- `startContainer`, `startOffset` – узел и начальное смещение,
 - в примере выше: первый текстовый узел внутри тега `<p>` и `2`.
- `endContainer`, `endOffset` – узел и конечное смещение,
 - в примере выше: первый текстовый узел внутри тега `` и `3`.
- `collapsed` – boolean, `true`, если диапазон начинается и заканчивается на одном и том же месте (следовательно, в диапазон ничего не входит),
 - в примере выше: `false`
- `commonAncestorContainer` – ближайший общий предок всех узлов в пределах диапазона,
 - в примере выше: `<p>`

Методы Range

Существует множество удобных методов для манипулирования диапазонами.

Установить начало диапазона:

- `setStart(node, offset)` установить начальную границу в позицию `offset` в `node`
- `setStartBefore(node)` установить начальную границу прямо перед `node`
- `setStartAfter(node)` установить начальную границу прямо после `node`

Установить конец диапазона (похожи на предыдущие методы):

- `setEnd(node, offset)` установить конечную границу в позицию `offset` в `node`
- `setEndBefore(node)` установить конечную границу прямо перед `node`
- `setEndAfter(node)` установить конечную границу прямо после `node`

Как было показано, `node` может быть как текстовым узлом, так и элементом: для текстовых узлов `offset` пропускает указанное количество символов, в то время как для элементов – указанное количество дочерних узлов.

Другие:

- `selectNode(node)` выделить `node` целиком
- `selectNodeContents(node)` выделить всё содержимое `node`
- `collapse(toStart)` если указано `toStart=true`, установить конечную границу в начало, иначе установить начальную границу в конец, схлопывая таким образом диапазон
- `cloneRange()` создать новый диапазон с идентичными границами

Чтобы манипулировать содержимым в пределах диапазона:

- `deleteContents()` – удалить содержимое диапазона из документа
- `extractContents()` – удалить содержимое диапазона из документа и вернуть как [DocumentFragment](#)
- `cloneContents()` – склонировать содержимое диапазона и вернуть как [DocumentFragment](#)
- `insertNode(node)` – вставить `node` в документ в начале диапазона
- `surroundContents(node)` – обернуть `node` вокруг содержимого диапазона. Чтобы этот метод сработал, диапазон должен содержать как открывающие, так и закрывающие теги для всех элементов внутри себя: не допускаются частичные диапазоны по типу `<i>abc`.

Используя эти методы, мы можем делать с выделенными узлами что угодно.

Проверим описанные методы в действии:

Нажмите на кнопку, чтобы соответствующий метод отработал на выделении, или на "resetExample" для сброса.

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<p id="result"></p>
<script>
    let range = new Range();

    // Каждый описанный метод представлен здесь:
    let methods = {
        deleteContents() {
            range.deleteContents()
        },
        extractContents() {
            let content = range.extractContents();
            result.innerHTML = "";
            result.append("Извлечено: ", content);
        },
        cloneContents() {
            let content = range.cloneContents();
            result.innerHTML = "";
            result.append("Клонировано: ", content);
        },
        insertNode() {
            let newNode = document.createElement('u');
            newNode.innerHTML = "НОВЫЙ УЗЕЛ";
            range.insertNode(newNode);
        },
        surroundContents() {
            let newNode = document.createElement('u');
            try {
                range.surroundContents(newNode);
            } catch(e) { alert(e) }
        },
        resetExample() {
            p.innerHTML = `Example: <i>italic</i> and <b>bold</b>`;
            result.innerHTML = "";

            range.setStart(p.firstChild, 2);
            range.setEnd(p.querySelector('b').firstChild, 3);

            window.getSelection().removeAllRanges();
            window.getSelection().addRange(range);
        }
    };

    for(let method in methods) {
        document.write(`<div><button onclick="methods.${method}()">${method}</button></div>`);
    }

    methods.resetExample();
</script>
```

Нажмите на кнопку, чтобы соответствующий метод отработал на выделении, или на "resetExample", чтобы восстановить выделение как было.

Example: *italic* and **bold**

deleteContents
extractContents
cloneContents
insertNode
surroundContents
resetExample

Также существуют методы сравнения диапазонов, но они редко используются. Когда они вам понадобятся, вы можете прочитать о них в [спецификации](#) или [справочнике MDN](#).

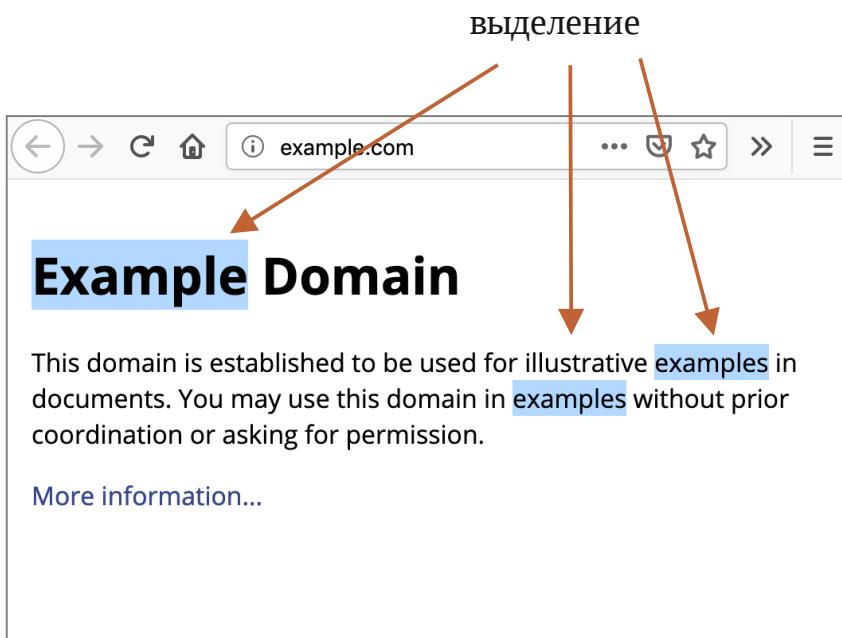
Selection

`Range` это общий объект для управления диапазонами выделения. Мы можем создавать и передавать подобные объекты. Сами по себе они ничего визуально не выделяют.

Выделение в документе представлено объектом `Selection`, который может быть получен как `window.getSelection()` или `document.getSelection()`.

Выделение может включать ноль или более диапазонов. По крайней мере, так утверждается в [Спецификации Selection API](#). На практике же выделить несколько диапазонов в документе можно только в Firefox, используя `Ctrl+click` (`Cmd+click` для Mac).

Ниже представлен скриншот выделения с 3 диапазонами, сделанный в Firefox:



Остальные браузеры поддерживают максимум 1 диапазон. Как мы увидим далее, некоторые методы `Selection` подразумевают, что может быть несколько диапазонов, но, как было сказано ранее, во всех браузерах, кроме Firefox, может быть не более одного диапазона.

Свойства Selection

Аналогично диапазону, выделение имеет начальную границу, именуемую «якорем», и конечную, называемую «фокусом».

Основные свойства выделения:

- `anchorNode` – узел, с которого начинается выделение,
- `anchorOffset` – смещение в `anchorNode`, где начинается выделение,
- `focusNode` – узел, на котором выделение заканчивается,
- `focusOffset` – смещение в `focusNode`, где выделение заканчивается,
- `isCollapsed` – `true`, если диапазон выделения пуст или не существует.
- `rangeCount` – количество диапазонов в выделении, максимум `1` во всех браузерах, кроме Firefox.

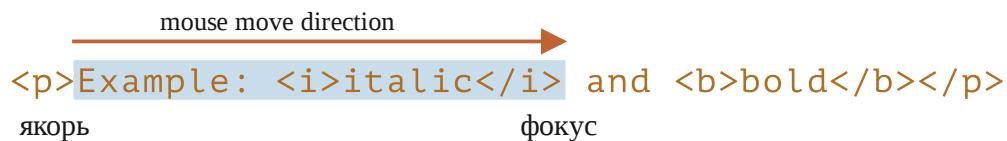
 Конец выделения может быть в документе до его начала

Существует несколько методов выделить содержимое, в зависимости от устройства пользователя: мышь, горячие клавиши, нажатия пальцем и другие.

Некоторые из них, такие как мышь, позволяют создавать выделение в обоих направлениях: слева направо и справа налево.

Если начало (якорь) выделения идёт в документе перед концом (фокус), говорят, что такое выделение «направлено вперёд».

К примеру, если пользователь начинает выделение с помощью мыши в направлении от «Example» до «italic»:



Иначе, если выделение идёт от «italic» до «Example», выделение идёт в «обратном» направлении, его фокус будет перед якорем:



Это отличается от объектов `Range`, которые всегда направлены вперёд: начало диапазона не может стоять после его конца.

События при выделении

Существуют события, позволяющие отслеживать выделение:

- `elem.onselectstart` – когда с elem начинается выделение, например пользователь начинает двигать мышкой с зажатой кнопкой.
 - `preventDefault()` отменяет начало выделения.
 - `document.onselectionchange` – когда выделение изменено.
 - Заметьте: этот обработчик можно поставить только на `document`.

Демо отслеживания выделения

Ниже представлено небольшое демо. В нём границы выделения выводятся динамически по мере того, как оно меняется:

<u>Выдели меня: **<i>**курсив**</i>** и ****жирный****

```

От <input id="from" disabled> - До <input id="to" disabled>
<script>
  document.onselectionchange = function() {
    let {anchorNode, anchorOffset, focusNode, focusOffset} = document.getSelection();

    from.value = `${anchorNode && anchorNode.data}:${anchorOffset}`;
    to.value = `${focusNode && focusNode.data}:${focusOffset}`;
  };
</script>

```

Демо получения выделения

Чтобы получить всё выделение:

- Как текст: просто вызовите `document.getSelection().toString()`.
- Как DOM-элементы: получите выделенные диапазоны и вызовите их метод `cloneContents()` (только первый диапазон, если мы не поддерживаем мультивыделение в Firefox).

Ниже представлено демо получения выделения как в виде текста, так и в виде DOM-узлов:

```

<p id="p">Выдели меня: <i>курсив</i> и <b>жирный</b></p>

Склонировано: <span id="cloned"></span>
<br>
Как текст: <span id="astext"></span>

<script>
  document.onselectionchange = function() {
    let selection = document.getSelection();

    cloned.innerHTML = astext.innerHTML = "";

    // Клонируем DOM-элементы из диапазонов (здесь мы поддерживаем множественное выде.
    for (let i = 0; i < selection.rangeCount; i++) {
      cloned.append(selection.getRangeAt(i).cloneContents());
    }

    // Получаем как текст
    astext.innerHTML += selection;
  };
</script>

```

Методы Selection

Методы Selection для добавления и удаления диапазонов:

- `getRangeAt(i)` – взять i-ый диапазон, начиная с 0. Во всех браузерах, кроме Firefox, используется только 0.
- `addRange(range)` – добавить range в выделение. Все браузеры, кроме Firefox, проигнорируют этот вызов, если в выделении уже есть диапазон.
- `removeRange(range)` – удалить range из выделения.
- `removeAllRanges()` – удалить все диапазоны.
- `empty()` – сокращение для `removeAllRanges`.

Также существуют методы управления диапазонами выделения напрямую, без обращения к Range:

- `collapse(node, offset)` – заменить выделенный диапазон новым, который начинается и заканчивается на node, на позиции offset.
- `setPosition(node, offset)` – то же самое, что `collapse` (дублирующий метод-псевдоним).
- `collapseToStart()` – схлопнуть (заменить на пустой диапазон) к началу выделения,
- `collapseToEnd()` – схлопнуть диапазон к концу выделения,
- `extend(node, offset)` – переместить фокус выделения к данному node, с позиции offset,
- `setBaseAndExtent(anchorNode, anchorOffset, focusNode, focusOffset)` – заменить диапазон выделения на заданные начало anchorNode/anchorOffset и конец focusNode/focusOffset. Будет выделено всё содержимое между этими границами
- `selectAllChildren(node)` – выделить все дочерние узлы данного узла node.
- `deleteFromDocument()` – удалить содержимое выделения из документа.
- `containsNode(node, allowPartialContainment = false)` – проверяет, содержит ли выделение node (частично, если второй аргумент равен true)

Так что для многих задач мы можем вызывать методы Selection, не обращаясь к связанному объекту Range.

К примеру, выделение всего параграфа `<p>`:

```
<p id="p">Выдели меня: <i>курсив</i> и <b>жирный</b></p>

<script>
  // выделить всё содержимое от нулевого потомка тега <p> до последнего
  document.getSelection().setBaseAndExtent(p, 0, p, p.childNodes.length);
</script>
```

То же самое с помощью `Range`:

```
<p id="p">Выдели меня: <i>курсив</i> и <b>жирный</b></p>

<script>
  let range = new Range();
  range.selectNodeContents(p); // или selectNode(p), чтобы выделить и тег <p>

  document.getSelection().removeAllRanges(); // очистить текущее выделение, если оно есть
  document.getSelection().addRange(range);
</script>
```

❶ Чтобы что-то выделить, сначала снимите текущее выделение

Если выделение уже существует, сначала снимите его, используя `removeAllRanges()`, и только затем добавляйте новые диапазоны. В противном случае все браузеры, кроме Firefox, проигнорируют добавление.

Исключением являются некоторые методы выделения, которые заменяют существующее выделение, например, `setBaseAndExtent`.

Выделение в элементах форм

Элементы форм, такие как `input` и `textarea`, предоставляют [отдельное API для выделения](#). Так как значения полей представляют собой простой текст, а не HTML, и нам не нужны такие сложные объекты, как `Range` и `Selection`.

Свойства:

- `input.selectionStart` – позиция начала выделения (это свойство можно изменять),
- `input.selectionEnd` – позиция конца выделения (это свойство можно изменять),
- `input.selectionDirection` – направление выделения, одно из: «forward» (вперёд), «backward» (назад) или «none» (без направления, если, к примеру, выделено с помощью двойного клика мыши).

События:

- `input.onselect` – срабатывает, когда выделение завершено.

Методы:

- `input.select()` – выделяет всё содержимое `input` (может быть `textarea` вместо `input`),

- `input.setSelectionRange(start, end, [direction])` – изменить выделение, чтобы начиналось с позиции `start`, и заканчивалось `end`, в данном направлении `direction` (необязательный параметр).
- `input.setRangeText(replacement, [start], [end], [selectionMode])` – заменяет выделенный текст в диапазоне новым.

Если аргументы `start` и `end` указаны, то они задают начало и конец диапазона, иначе используется текущее выделение.

Последний аргумент, `selectionMode`, определяет, как будет вести себя выделение после замены текста. Возможные значения:

- `"select"` – только что вставленный текст будет выделен.
- `"start"` – диапазон выделения схлопнется прямо перед вставленным текстом (так что курсор окажется непосредственно перед ним).
- `"end"` – диапазон выделения схлопнется прямо после вставленного текста (курсор окажется сразу после него).
- `"preserve"` – пытается сохранить выделение. Значение по умолчанию.

Давайте посмотрим на эти методы в действии.

Пример: отслеживание выделения

К примеру, этот код использует событие `onselect`, чтобы отслеживать выделение:

```
<textarea id="area" style="width:80%;height:60px">
Выделите что-нибудь в этом тексте, чтобы обновить значения ниже.
</textarea>
<br>
От <input id="from" disabled> – До <input id="to" disabled>
```

```
<script>
area.onselect = function() {
    from.value = area.selectionStart;
    to.value = area.selectionEnd;
};
</script>
```

Выделите что-нибудь в этом тексте, чтобы обновить значения ниже.

От – До

Заметьте:

- `onselect` срабатывает при выделении чего-либо, но не при снятии выделения.

- событие `document.onselectionchange` не должно срабатывать при выделении внутри элемента формы в соответствии со спецификацией ↗, так как оно не является выделением элементов в `document`. Хотя некоторые браузеры генерируют это событие, полагаться на это не стоит.

Пример: изменение позиции курсора

Мы можем изменять `selectionStart` и `selectionEnd`, устанавливая выделение.

Важный граничный случай – когда `selectionStart` и `selectionEnd` равны друг другу. В этом случае они указывают на позицию курсора. Иными словами, когда ничего не выбрано, выделение склоняется на позиции курсора.

Таким образом, задавая `selectionStart` и `selectionEnd` одно и то же значение, мы можем передвигать курсор.

Например:

```
<textarea id="area" style="width:80%;height:60px">
Переведите фокус на меня, курсор окажется на 10-й позиции
</textarea>

<script>
area.onfocus = () => {
  // нулевая задержка setTimeout нужна, чтобы это сработало после получения фокуса
  setTimeout(() => {
    // мы можем задать любое выделение
    // если начало и конец совпадают, курсор устанавливается на этом месте
    area.selectionStart = area.selectionEnd = 10;
  });
};
</script>
```

Переведите фокус на меня, курсор окажется на 10-й позиции

Пример: изменение выделения

Чтобы изменять содержимое выделения, мы можем использовать метод `input.setRangeText`. Конечно, мы можем читать `selectionStart/End` и, зная позиции выделения, изменять соответствующую подстроку в `value`, но `setRangeText` намного мощнее и, зачастую, удобнее.

Это довольно сложный метод. В простейшем случае он принимает один аргумент, заменяет содержание выделенной области и снимает выделение.

В этом примере выделенный текст будет обёрнут в `* . . . *`:

```

<input id="input" style="width:200px" value="Select here and click the button">
<button id="button">Обернуть выделение звёздочками *...*</button>

<script>
button.onclick = () => {
  if (input.selectionStart == input.selectionEnd) {
    return; // ничего не выделено
  }

  let selected = input.value.slice(input.selectionStart, input.selectionEnd);
  input.setRangeText(`*${selected}*`);
};
</script>

```

Select here and click the button Обернуть выделение звёздочками *...*

Передавая больше параметров, мы можем устанавливать `start` и `end`.

В этом примере мы найдём "ЭТО" в поле ввода, заменим его и оставим заменённый текст выделенным:

```

<input id="input" style="width:200px" value="Замените ЭТО в тексте">
<button id="button">Заменить ЭТО</button>

<script>
button.onclick = () => {
  let pos = input.value.indexOf("ЭТО");
  if (pos >= 0) {
    input.setRangeText("*ЭТО*", pos, pos + 3, "select");
    input.focus(); // ставим фокус, чтобы выделение было видно
  }
};
</script>

```

Замените ЭТО в тексте Заменить ЭТО

Пример: вставка на месте курсора

Если ничего не выделено, или мы указали одинаковые `start` и `end` в методе `setRangeText`, то текст просто вставляется, и ничего не удаляется.

Мы также можем вставить что-нибудь на текущей позиции курсора, используя `setRangeText`.

Кнопка в примере вставляет "ПРИВЕТ" на месте курсора и устанавливает его после вставленного текста. Если какой-то текст был выделен, он будет заменён (мы можем узнать о наличии выделения, проверив `selectionStart!=selectionEnd` и, если выделение есть, сделать что-то ещё):

```
<input id="input" style="width:200px" value="Текст Текст Текст Текст Текст">
<button id="button">Вставить "ПРИВЕТ" на месте курсора</button>

<script>
  button.onclick = () => {
    input.setRangeText("ПРИВЕТ", input.selectionStart, input.selectionEnd, "end");
    input.focus();
  };
</script>
```

Текст Текст Текст Текст Текст Вставить "ПРИВЕТ" на месте курсора

Сделать что-то невыделяемым

Существуют три способа сделать что-то невыделяемым:

1. Используйте CSS-свойство `user-select: none`.

```
<style>
#elem {
  user-select: none;
}
</style>
<div>Можно выделить <div id="elem">Нельзя выделить</div> Можно выделить</div>
```

Это свойство не позволяет начать выделение с `elem`, но пользователь может начать выделять с другого места и включить `elem`.

После этого `elem` станет частью `document.getSelection()`, так что на самом деле выделение произойдёт, но его содержимое обычно игнорируется при копировании и вставке.

2. Предотвратить действие по умолчанию в событии `onselectstart` или `mousedown`.

```
<div>Можно выделить <div id="elem">Нельзя выделить</div> Можно выделить</div>

<script>
  elem.onselectstart = () => false;
</script>
```

Этот способ также не даёт начать выделение с `elem`, но пользователь может начать с другого элемента, а затем расширить выделение до `elem`.

Это удобно, когда есть другой обработчик события на том действии, которое запускает выделение (скажем, `mousedown`). Так что мы отключаем выделение, чтобы избежать конфликта.

А содержимое `elem` при этом может быть скопировано.

3. Мы также можем очистить выделение после срабатывания с помощью `document.getSelection().empty()`. Этот способ используется редко, так как он вызывает нежелаемое мерцание при появлении и исчезновении выделения.

Ссылки

- Спецификация DOM: Range [↗](#)
- Selection API [↗](#)
- Спецификация HTML: API для выделения в элементах управления текстом [↗](#)

Итого

Мы подробно рассмотрели два разных API для выделения:

1. Для документа: объекты `Selection` и `Range`.
2. Для `input`, `textarea`: дополнительные методы и свойства.

Второе API очень простое, так как работает с текстом.

Самые используемые готовые решения:

1. Получить выделение:

```
let selection = document.getSelection();

let cloned = /* элемент, в который мы хотим скопировать выделенные узлы */;

// затем применяем методы Range к selection.getRangeAt(0)
// или, как здесь, ко всем диапазонам, чтобы поддерживать множественное выделение
for (let i = 0; i < selection.rangeCount; i++) {
  cloned.append(selection.getRangeAt(i).cloneContents());
}
```

2. Установить выделение:

```
let selection = document.getSelection();

// напрямую:
selection.setBaseAndExtent(...from...to...);

// или можно создать диапазон range и:
selection.removeAllRanges();
selection.addRange(range);
```

И пару слов о курсоре. Позиция курсора в редактируемых элементах, таких как `<textarea>`, всегда находится в начале или конце выделения.

Мы можем использовать это, как для того, чтобы получить позицию курсора, так и чтобы переместить его, установив `elem.selectionStart` и `elem.selectionEnd`.

P.S. Если вам нужна поддержка старого IE8-, посмотрите в [архивную статью](#).

Событийный цикл: микрозадачи и макрозадачи

Поток выполнения в браузере, равно как и в Node.js, основан на *событийном цикле*.

Понимание работы событийного цикла важно для оптимизаций, иногда для правильной архитектуры.

В этой главе мы сначала разберём теорию, а затем рассмотрим её практическое применение.

Событийный цикл

Идея *событийного цикла* очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Общий алгоритм движения:

1. Пока есть задачи:
 - выполнить их, начиная с самой старой
2. Бездействовать до появления новой задачи, а затем перейти к пункту 1

Это формализация того, что мы наблюдаем, просматривая веб-страницу. Движок JavaScript большую часть времени ничего не делает и работает, только если требуется исполнить скрипт/обработчик или обработать событие.

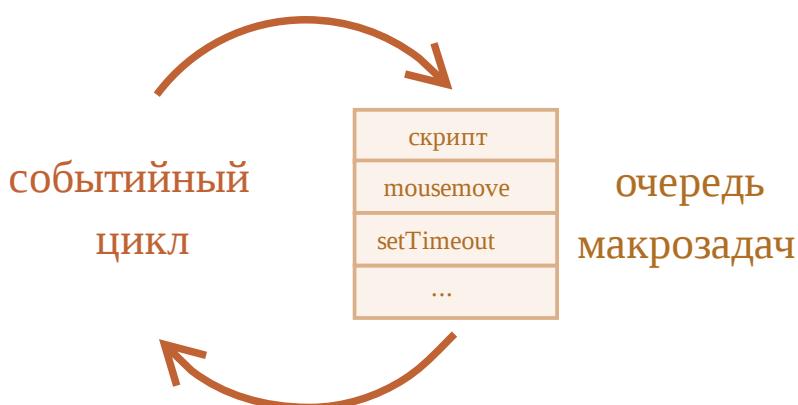
Примеры задач:

- Когда загружается внешний скрипт `<script src=". . .">`, то задача – это выполнение этого скрипта.
- Когда пользователь двигает мышь, задача – сгенерировать событие `mousemove` и выполнить его обработчики.
- Когда истечёт таймер, установленный с помощью `setTimeout(func, . . .)`, задача – это выполнение функции `func`
- И так далее.

Задачи поступают на выполнение – движок выполняет их – затем ожидает новые задачи (во время ожидания практически не нагружая процессор компьютера)

Может так случиться, что задача поступает, когда движок занят чем-то другим, тогда она ставится в очередь.

Очередь, которую формируют такие задачи, называют «очередью макрозадач» (macrotask queue, термин v8).



Например, когда движок занят выполнением скрипта, пользователь может передвинуть мышь, тем самым вызвав появление события `mousemove`, или может истечь таймер, установленный `setTimeout`, и т.п. Эти задачи формируют очередь, как показано на иллюстрации выше.

Задачи из очереди исполняются по правилу «первым пришёл – первым ушёл». Когда браузер заканчивает выполнение скрипта, он обрабатывает событие `mousemove`, затем выполняет обработчик, заданный `setTimeout`, и так далее.

Пока что всё просто, не правда ли?

Отметим две детали:

1. Рендеринг (отрисовка страницы) никогда не происходит во время выполнения задачи движком. Не имеет значения, сколь долго выполняется задача. Изменения в DOM отрисовываются только после того, как задача выполнена.
2. Если задача выполняется очень долго, то браузер не может выполнять другие задачи, обрабатывать пользовательские события, поэтому спустя некоторое время браузер предлагает «убить» долго выполняющуюся задачу. Такое возможно, когда в скрипте много сложных вычислений или ошибка, ведущая к бесконечному циклу.

Это была теория. Теперь давайте взглянем, как можно применить эти знания.

Пример 1: разбиение «тяжёлой» задачи.

Допустим, у нас есть задача, требующая значительных ресурсов процессора.

Например, подсветка синтаксиса (используется для выделения цветом участков кода на этой странице) – довольно процессороёмкая задача. Для подсветки кода надо выполнить синтаксический анализ, создать много элементов для цветового выделения, добавить их в документ – для большого текста это требует значительных ресурсов.

Пока движок занят подсветкой синтаксиса, он не может делать ничего, связанного с DOM, не может обрабатывать пользовательские события и т.д. Возможно даже «подвисание» браузера, что совершенно неприемлемо.

Мы можем избежать этого, разбив задачу на части. Сделать подсветку для первых 100 строк, затем запланировать `setTimeout` (с нулевой задержкой) для разметки следующих 100 строк и т.д.

Чтобы продемонстрировать такой подход, давайте будем использовать для простоты функцию, которая считает от 1 до 10000000000 .

Если вы запустите код ниже, движок «зависнет» на некоторое время. Для серверного JS это будет явно заметно, а если вы будете выполнять этот код в браузере, то попробуйте понажимать другие кнопки на странице – вы заметите, что никакие другие события не обрабатываются до завершения функции счёта.

```
let i = 0;

let start = Date.now();

function count() {

    // делаем тяжёлую работу
    for (let j = 0; j < 1e9; j++) {
        i++;
    }

    alert("Done in " + (Date.now() - start) + 'ms');
}

count();
```

Браузер может даже показать сообщение «скрипт выполняется слишком долго».

Давайте разобьём задачу на части, воспользовавшись вложенным `setTimeout`:

```
let i = 0;
```

```

let start = Date.now();

function count() {
    // делаем часть тяжёлой работы (*)
    do {
        i++;
    } while (i % 1e6 != 0);

    if (i == 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count); // планируем новый вызов (**)
    }
}

count();

```

Теперь интерфейс браузера полностью работоспособен во время выполнения «счёта».

Один вызов `count` делает часть работы (*), а затем, если необходимо, планирует свой очередной запуск (**):

1. Первое выполнение производит счёт: $i=1\dots 1000000$.
2. Второе выполнение производит счёт: $i=1000001\dots 2000000$.
3. ...и так далее.

Теперь если новая сторонняя задача (например, событие `onclick`) появляется, пока движок занят выполнением 1-й части, то она становится в очередь, и затем выполняется, когда 1-я часть завершена, перед следующей частью. Периодические возвраты в событийный цикл между запусками `count` дают движку достаточно «воздуха», чтобы сделать что-то ещё, отреагировать на действия пользователя.

Отметим, что оба варианта – с разбиением задачи с помощью `setTimeout` и без – сопоставимы по скорости выполнения. Нет большой разницы в общем времени счёта.

Чтобы сократить разницу ещё сильнее, давайте немного улучшим наш код.

Мы перенесём планирование очередного вызова в начало `count()`:

```

let i = 0;

let start = Date.now();

function count() {

```

```
// перенесём планирование очередного вызова в начало
if (i < 1e9 - 1e6) {
    setTimeout(count); // запланировать новый вызов
}

do {
    i++;
} while (i % 1e6 != 0);

if (i == 1e9) {
    alert("Done in " + (Date.now() - start) + 'ms');
}

}

count();
```

Теперь, когда мы начинаем выполнять `count()` и видим, что потребуется выполнить `count()` ещё раз, мы планируем этот вызов немедленно, перед выполнением работы.

Если вы запустите этот код, то легко заметите, что он требует значительно меньше времени.

Почему?

Всё просто: как вы помните, в браузере есть минимальная задержка в 4 миллисекунды при множестве вложенных вызовов `setTimeout`. Даже если мы указываем задержку `0`, на самом деле она будет равна `4 ms` (или чуть больше). Поэтому чем раньше мы запланируем выполнение – тем быстрее выполнится код.

Итак, мы разбили ресурсоёмкую задачу на части – теперь она не блокирует пользовательский интерфейс, причём почти без потерь в общем времени выполнения.

Пример 2: индикация прогресса

Ещё одно преимущество разделения на части крупной задачи в браузерных скриптах – это возможность показывать индикатор выполнения.

Обычно браузер отрисовывает содержимое страницы после того, как заканчивается выполнение текущего кода. Не имеет значения, насколько долго выполняется задача. Изменения в DOM отображаются только после её завершения.

С одной стороны, это хорошо, потому что наша функция может создавать много элементов, добавлять их по одному в документ и изменять их стили – пользователь не увидит «промежуточного», незаконченного состояния. Это важно, верно?

В примере ниже изменения `i` не будут заметны, пока функция не завершится, поэтому мы увидим только последнее значение `i`:

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

<script>

function count() {
  for (let i = 0; i < 1e6; i++) {
    i++;
    progress.innerHTML = i;
  }
}

count();
</script>
```

...Но, возможно, мы хотим что-нибудь показать во время выполнения задачи, например, индикатор выполнения.

Если мы разобьём тяжёлую задачу на части, используя `setTimeout`, то изменения индикатора будут отрисованы в промежутках между частями.

Так будет красивее:

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

<script>
let i = 0;

function count() {

  // сделать часть крупной задачи (*)
  do {
    i++;
    progress.innerHTML = i;
  } while (i % 1e3 != 0);

  if (i < 1e7) {
    setTimeout(count);
  }
}

count();
</script>
```

Теперь `<div>` показывает растущее значение `i` – это своего рода индикатор выполнения.

Пример 3: делаем что-нибудь после события

В обработчике события мы можем решить отложить некоторые действия, пока событие не «всплылёт» и не будет обработано на всех уровнях. Мы можем добиться этого, обернув код в `setTimeout` с нулевой задержкой.

В главе [Генерация пользовательских событий](#) мы видели пример: наше событие `menu-open` генерируется через `setTimeout`, чтобы оно возникло после того, как полностью обработано событие «click».

```
menu.onclick = function() {
  // ...

  // создадим наше собственное событие с данными пункта меню, по которому щёлкнули мышь
  let customEvent = new CustomEvent("menu-open", {
    bubbles: true
  });

  // сгенерировать наше событие асинхронно
  setTimeout(() => menu.dispatchEvent(customEvent));
};
```

Макрозадачи и Микрозадачи

Помимо макрозадач, описанных в этой части, существуют *микрозадачи*, упомянутые в главе [Микрозадачи](#).

Микрозадачи приходят только из кода. Обычно они создаются промисами: выполнение обработчика `.then/catch/finally` становится микрозадачей. Микрозадачи также используются «под капотом» `await`, т.к. это форма обработки промиса.

Также есть специальная функция `queueMicrotask(func)`, которая помещает `func` в очередь микрозадач.

Сразу после каждой макрозадачи движок исполняет все задачи из очереди микрозадач перед тем, как выполнить следующую макрозадачу или отобразить изменения на странице, или сделать что-то ещё.

Например:

```
setTimeout(() => alert("timeout"));

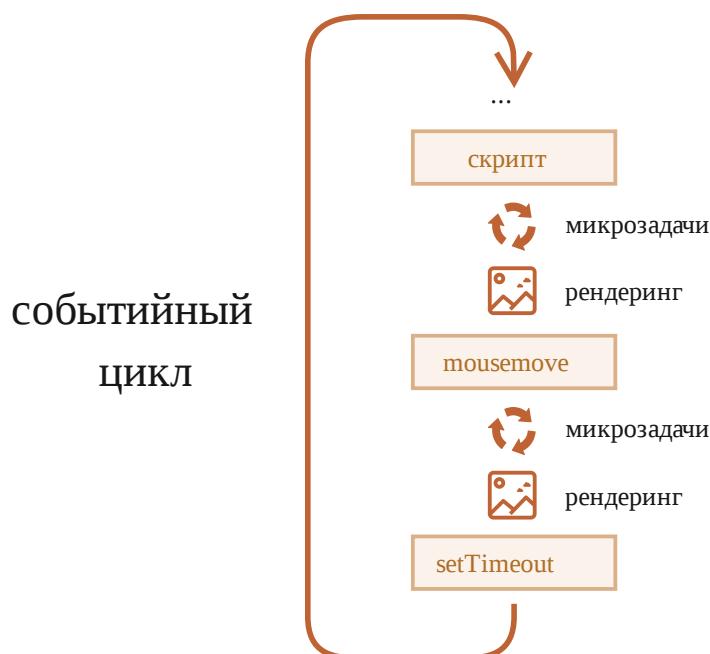
Promise.resolve()
  .then(() => alert("promise"));

alert("code");
```

Какой здесь будет порядок?

1. `code` появляется первым, т.к. это обычный синхронный вызов.
2. `promise` появляется вторым, потому что `.then` проходит через очередь микрозадач и выполняется после текущего синхронного кода.
3. `timeout` появляется последним, потому что это макрозадача.

Более подробное изображение событийного цикла выглядит так:



Все микrozадачи завершаются до обработки каких-либо событий или рендеринга, или перехода к другой макрозадаче.

Это важно, так как гарантирует, что общее окружение остаётся одним и тем же между микrozадачами – не изменены координаты мыши, не получены новые данные по сети и т.п.

Если мы хотим запустить функцию асинхронно (после текущего кода), но до отображения изменений и до новых событий, то можем запланировать это через `queueMicrotask`.

Вот пример с индикатором выполнения, похожий на предыдущий, но в этот раз использована функция `queueMicrotask` вместо `setTimeout`. Обратите внимание – отрисовка страницы происходит только в самом конце. Как и в случае обычного синхронного кода.

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

<script>
  let i = 0;
```

```

function count() {

    // делаем часть крупной задачи (*)
    do {
        i++;
        progress.innerHTML = i;
    } while (i % 1e3 != 0);

    if (i < 1e6) {
        queueMicrotask(count);
    }

}

count();
</script>

```

Итого

Более подробный алгоритм событийного цикла (хоть и упрощённый в сравнении со [спецификацией](#)):

1. Выбрать и исполнить старейшую задачу из очереди макрозадач (например, «`script`»).
2. Исполнить все микрозадачи:
 - Пока очередь микрозадач не пуста: - Выбрать из очереди и исполнить старейшую микрозадачу
3. Отрисовать изменения страницы, если они есть.
4. Если очередь макрозадач пуста – подождать, пока появится макрозадача.
5. Перейти к шагу 1.

Чтобы добавить в очередь новую макрозадачу:

- Используйте `setTimeout(f)` с нулевой задержкой.

Этот способ можно использовать для разбиения больших вычислительных задач на части, чтобы браузер мог реагировать на пользовательские события и показывать прогресс выполнения этих частей.

Также это используется в обработчиках событий для отложенного выполнения действия после того, как событие полностью обработано (всплытие завершено).

Для добавления в очередь новой микрозадачи:

- Используйте `queueMicrotask(f)`.
- Также обработчики промисов выполняются в рамках очереди микрозадач.

События пользовательского интерфейса и сетевые события в промежутках между микрозадачами не обрабатываются: микрозадачи исполняются непрерывно одна за другой.

Поэтому `queueMicrotask` можно использовать для асинхронного выполнения функции в том же состоянии окружения.

Web Workers

Для длительных тяжёлых вычислений, которые не должны блокировать событийный цикл, мы можем использовать [Web Workers ↗](#).

Это способ исполнить код в другом, параллельном потоке.

Web Workers могут обмениваться сообщениями с основным процессом, но они имеют свои переменные и свой событийный цикл.

Web Workers не имеют доступа к DOM, поэтому основное их применение – вычисления. Они позволяют задействовать несколько ядер процессора одновременно.

Задачи

Что код выведет в консоли?

важность: 5

```
setTimeout(function timeout() {
    console.log('Таймаут');
}, 0);

let p = new Promise(function(resolve, reject) {
    console.log('Создание промиса');
    resolve();
});

p.then(function(){
    console.log('Обработка промиса');
});

console.log('Конец скрипта');
```

[К решению](#)

Что код выведет в консоли?

важность: 5

```
console.log(1);

setTimeout(() => console.log(2));

Promise.resolve().then(() => console.log(3));

Promise.resolve().then(() => setTimeout(() => console.log(4)));

Promise.resolve().then(() => console.log(5));

setTimeout(() => console.log(6));

console.log(7);
```

[К решению](#)

Решения

Навигация по DOM-элементам

Дочерние элементы в DOM

Есть несколько способов для получения элементов, например:

DOM-узел элемента `<div>`:

```
document.body.firstChild
// или
document.body.children[0]
// или (первый узел пробел, поэтому выбираем второй)
document.body.childNodes[1]
```

DOM-узел элемента ``:

```
document.body.lastElementChild
// или
document.body.children[1]
```

Второй `` (с именем Пит):

```
// получаем <ul>, и его последнего ребёнка
document.body.lastElementChild.lastElementChild
```

[К условию](#)

Вопрос о соседях

1. Да. Верно. Элемент `elem.lastChild` всегда последний, у него нет ссылки `nextSibling`.
2. Нет. Неверно. Потому что `elem.children[0]` – потомок-элемент. Но перед ним могут быть другие узлы. Например, `previousSibling` может быть текстовым узлом.

Обратите внимание, что в обоих случаях, если детей нет, то будет ошибка. При этом `elem.lastChild` равен `null`, а значит – ошибка при попытке доступа к `elem.lastChild.nextSibling`.

[К условию](#)

Выделите ячейки по диагонали

Для получения доступа к диагональным ячейкам таблицы используем свойства `rows` и `cells`.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Поиск: `getElement*`, `querySelector*`

Поиск элементов

Есть много путей как это сделать.

Вот некоторые:

```
// 1. Таблица с `id="age-table"`.
let table = document.getElementById('age-table')

// 2. Все label в этой таблице
table.getElementsByTagName('label')
// или
document.querySelectorAll('#age-table label')

// 3. Первый td в этой таблице
table.rows[0].cells[0]
// или
```

```
table.getElementsByTagName('td')[0]
// или
table.querySelector('td')

// 4. Форма с name="search"
// предполагаем, что есть только один элемент с таким name в документе
let form = document.getElementsByName('search')[0]
// или, именно форма:
document.querySelector('form[name="search"]')

// 5. Первый input в этой форме
form.getElementsByTagName('input')[0]
// или
form.querySelector('input')

// 6. Последний input в этой форме
let inputs = form.querySelectorAll('input') // найти все input
inputs[inputs.length-1] // взять последний
```

К условию

Свойства узлов: тип, тег и содержимое

Считаем потомков

Пройдём циклом по всем элементам ``:

```
for (let li of document.querySelectorAll('li')) {
  ...
}
```

В цикле нам нужно получить текст в каждом элементе `li`. Мы можем прочитать текстовое содержимое элемента списка из первого дочернего узла `li`, который будет текстовым узлом:

```
for (let li of document.querySelectorAll('li')) {
  let title = li.firstChild.data;

  // переменная title содержит текст элемента <li>
}
```

Так мы сможем получить количество потомков как `li.getElementsByTagName('li').length`.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Что содержит свойство `nodeType`?

Здесь есть подвох.

Во время выполнения `<script>` последним DOM-узлом является `<script>`, потому что браузер ещё не обработал остальную часть страницы.

Поэтому результатом будет `1` (узел-элемент).

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

[К условию](#)

Тег в комментарии

Ответ: `BODY`.

```
<script>
  let 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-->`, т.к. `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. Значение свойства `data` для элемента-комментария – это его содержимое (внутри `<! - - . . - - >`): "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.name); // HTMLDocument
```

Итак, `document` – объект класса `HTMLDocument`.

Какое место `HTMLDocument` занимает в иерархии?

Можно поискать в документации. Но попробуем выяснить это самостоятельно.

Пройдём по цепочке прототипов по ссылке `__proto__`.

Как мы знаем, методы класса находятся в `prototype` конструктора. Например, в `HTMLDocument.prototype` находятся методы для объектов типа `document`.

Также внутри `prototype` есть ссылка на функцию-конструктор:

```
alert(HTMLDocument.prototype.constructor === HTMLDocument); // true
```

Чтобы получить имя класса в строковой форме, используем `constructor.name`. Сделаем это для всей цепочки прототипов `document` вверх до класса `Node`:

```
alert(HTMLDocument.prototype.constructor.name); // HTMLDocument
alert(HTMLDocument.prototype.__proto__.constructor.name); // Document
alert(HTMLDocument.prototype.__proto__.__proto__.constructor.name); // Node
```

Вот и иерархия.

Мы также можем исследовать объект с помощью `console.dir(document)` и увидеть имена функций-конструкторов, открыв `__proto__`. Браузерная консоль берёт их как раз из свойства `constructor`.

[К условию](#)

Атрибуты и свойства

Получите атрибут

```
<!DOCTYPE html>
<html>
<body>

<div data-widget-name="menu">Choose the genre</div>

<script>
    // получаем элемент
    let elem = document.querySelector('[data-widget-name]');

    // читаем значение
    alert(elem.dataset.widgetName);
    // или так
    alert(elem.getAttribute('data-widget-name'));
</script>
</body>
</html>
```

[К условию](#)

Сделайте внешние ссылки оранжевыми

Во-первых, мы должны найти все внешние ссылки.

Это можно сделать двумя способами.

Первый – это найти все ссылки, используя `document.querySelectorAll('a')`, а затем отфильтровать ненужное:

```
let links = document.querySelectorAll('a');

for (let link of links) {
```

```
let href = link.getAttribute('href');
if (!href) continue; // нет атрибута

if (!href.includes('://')) continue; // нет протокола

if (href.startsWith('http://internal.com')) continue; // внутренняя

link.style.color = 'orange';
}
```

Пожалуйста, обратите внимание: мы используем `link.getAttribute('href')`. Не `link.href`, потому что нам нужно значение из HTML.

...Другой, более простой путь – добавить проверку в CSS-селектор:

```
// найти все ссылки, атрибут href у которых содержит ://
// и при этом href не начинается с http://internal.com
let selector = 'a[href*="://"]:not([href^="http://internal.com"]);';
let links = document.querySelectorAll(selector);

links.forEach(link => link.style.color = 'orange');
```

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Изменение документа

createTextNode vs innerHTML vs textContent

Ответ: **1 и 3.**

Результатом обеих команд будет добавление `text` «как текст» в `elem`.

Пример:

```
<div id="elem1"></div>
<div id="elem2"></div>
<div id="elem3"></div>
<script>
  let text = '<b>текст</b>';
  elem1.append(document.createTextNode(text));
  elem2.innerHTML = text;
```

```
elem3.textContent = text;  
</script>
```

[К условию](#)

Очистите элемент

Сначала давайте посмотрим, как *не* надо это делать:

```
function clear(elem) {  
    for (let i=0; i < elem.childNodes.length; i++) {  
        elem.childNodes[i].remove();  
    }  
}
```

Это не будет работать, потому что вызов `remove()` сдвигает коллекцию `elem.childNodes`, поэтому элементы начинаются каждый раз с индекса `0`. А `i` увеличивается, и некоторые элементы будут пропущены.

Цикл `for..of` делает то же самое.

Правильным вариантом может быть:

```
function clear(elem) {  
    while (elem.firstChild) {  
        elem.firstChild.remove();  
    }  
}
```

А также есть более простой способ сделать то же самое:

```
function clear(elem) {  
    elem.innerHTML = '';  
}
```

[К условию](#)

Почему остаётся "aaa"?

HTML в задаче некорректен. В этом всё дело.

Браузер исправил ошибку автоматически. Но внутри `<table>` не может быть текста: в соответствии со спецификацией допускаются только

табличные теги. Поэтому браузер показывает "aaa" до `<table>`.

Теперь очевидно, что когда мы удаляем таблицу, текст остаётся.

На этот вопрос можно легко ответить, изучив DOM, используя инструменты браузера. Вы увидите "aaa" до элемента `<table>`.

Вообще, в стандарте HTML описано, как браузеру обрабатывать некорректный HTML, так что такое действие браузера является правильным.

[К условию](#)

Создайте список

Обратите внимание на использование `textContent` для добавления содержимого в ``.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Создайте дерево из объекта

Самый лёгкий способ – это использовать рекурсию.

1. [Решение с innerHTML](#) ↗ .
2. [Решение через DOM](#) ↗ .

[К условию](#)

Выведите список потомков в дереве

Чтобы добавить текст к каждому ``, мы можем изменить текстовый узел `data`.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Создайте календарь в виде таблицы

Для решения задачи сгенерируем таблицу в виде строки: "<table>...</table>" , а затем присвоим в `innerHTML`.

Алгоритм:

1. Создать заголовок таблицы с `<th>` и именами дней недели.
2. Создать объект даты `d = new Date(year, month-1)`. Это первый день месяца `month` (с учётом того, что месяцы в JS начинаются от 0, а не от 1).
3. Ячейки первого ряда пустые от начала и до дня недели `d.getDay()` , с которого начинается месяц. Заполним `<td></td>`.
4. Увеличить день в `d : d.setDate(d.getDate() + 1)`. Если `d.getMonth()` ещё не в следующем месяце, то добавим новую ячейку `<td>` в календарь. Если это воскресенье, то добавим новую строку `</tr><tr>`.
5. Если месяц закончился, но строка таблицы ещё не заполнена, добавим в неё пустые `<td>`, чтобы сделать в календаре красивые пустые квадратики.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Цветные часы с использованием setInterval

Для начала придумаем подходящую HTML/CSS-строктуру.

Здесь каждый компонент времени удобно поместить в соответствующий ``:

```
<div id="clock">
  <span class="hour">hh</span>:<span class="min">mm</span>:<span class="sec">s
</div>
```

Каждый `span` раскрашивается при помощи CSS.

Функция `update` будет обновлять часы, `setInterval` вызывает её каждую секунду:

```
function update() {
  let clock = document.getElementById('clock');
  let date = new Date(); // (*)
  let hours = date.getHours();
```

```
if (hours < 10) hours = '0' + hours;
clock.children[0].innerHTML = hours;

let minutes = date.getMinutes();
if (minutes < 10) minutes = '0' + minutes;
clock.children[1].innerHTML = minutes;

let seconds = date.getSeconds();
if (seconds < 10) seconds = '0' + seconds;
clock.children[2].innerHTML = seconds;
}
```

В строке `(*)` каждый раз мы получаем текущую дату. Вызовы `setInterval` не надёжны: они могут происходить с задержками.

Функция `clockStart` для запуска часов:

```
let timerId;

function clockStart() { // запустить часы
    timerId = setInterval(update, 1000);
    update(); // (*)
}

function clockStop() {
    clearInterval(timerId);
    timerId = null;
}
```

Обратите внимание, что вызов `update()` не только запланирован, но и тут же производится в строке `(*)`. Иначе посетителю пришлось бы ждать до первого выполнения `setInterval`, то есть целую секунду.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Вставьте HTML в список

Когда нам необходимо вставить фрагмент HTML-кода, можно использовать `insertAdjacentHTML`, он лучше всего подходит для таких задач.

Решение:

```
one.insertAdjacentHTML('afterend', '<li>2</li><li>3</li>');
```

[К условию](#)

Сортировка таблицы

Решение короткое, но может показаться немного сложным, поэтому здесь я предоставлю подробные комментарии:

```
let sortedRows = Array.from(table.rows)
  .slice(1)
  .sort((rowA, rowB) => rowA.cells[0].innerHTML > rowB.cells[0].innerHTML ? 1
        : -1)

table.tBodies[0].append(...sortedRows);
```

1.

Получим все `<tr>`, как `table.querySelectorAll('tr')`, затем сделаем массив из них, потому что нам понадобятся методы массива.

2.

Первый TR (`table.rows[0]`) – это заголовок таблицы, поэтому мы берём `.slice(1)`.

3.

Затем отсортируем их по содержимому в первом `<td>` (по имени).

4.

Теперь вставим узлы в правильном порядке

```
.append(...sortedRows).
```

Таблицы всегда имеют неявный элемент `<tbody>`, поэтому нам нужно получить его и вставить в него: простой `table.append(...)` потерпит неудачу.

Обратите внимание: нам не нужно их удалять, просто «вставляем их заново», они автоматически покинут старое место.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Стили и классы

Создать уведомление

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Размеры и прокрутка элементов

Найти размер прокрутки снизу

Решение:

```
let scrollBottom = elem.scrollHeight - elem.scrollTop - elem.clientHeight;
```

Другими словами: (вся высота) минус (часть, прокрученная сверху) минус (видимая часть) – результат в точности соответствует размеру прокрутки снизу.

[К условию](#)

Узнать ширину полосы прокрутки

Чтобы получить ширину полосы прокрутки, создадим элемент с прокруткой, но без рамок и внутренних отступов.

Тогда разница между его полной шириной `offsetWidth` и шириной внутреннего содержимого `clientWidth` будет равна как раз прокрутке:

```
// создадим элемент с прокруткой
let div = document.createElement('div');

div.style.overflowY = 'scroll';
div.style.width = '50px';
div.style.height = '50px';

// мы должны вставить элемент в документ, иначе размеры будут равны 0
document.body.append(div);
let scrollWidth = div.offsetWidth - div.clientWidth;

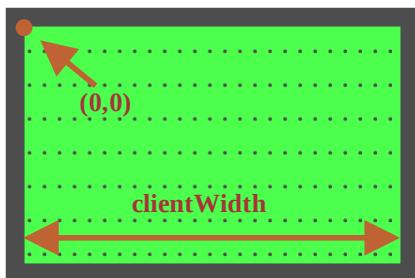
div.remove();

alert(scrollWidth);
```

Поместите мяч в центр поля

Мяч имеет CSS-свойство `position:absolute`. Это означает, что координаты `left/top` измеряются относительно ближайшего спозиционированного элемента, которым является `#field` (т.к. у него есть CSS-свойство `position:relative`).

Координаты отсчитываются от внутреннего верхнего левого угла поля:

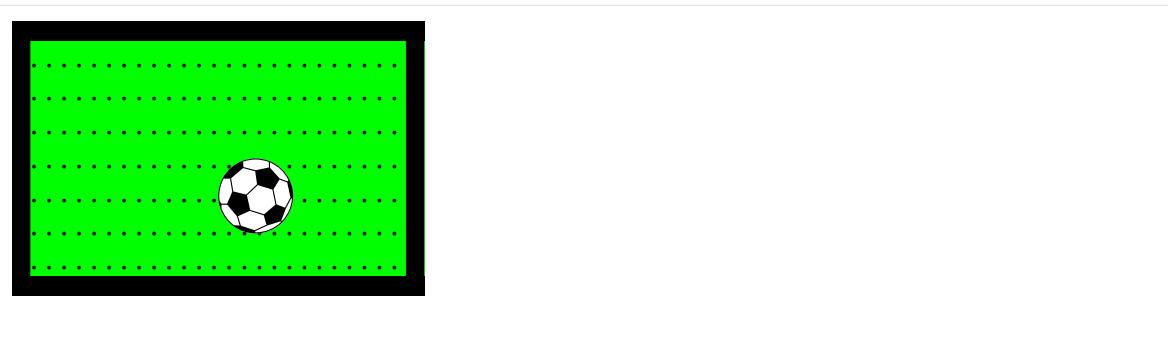


Ширина и высота внутреннего поля – это `clientWidth/clientHeight`. Таким образом, его центр имеет координаты `(clientWidth/2, clientHeight/2)`.

...Но если мы установим мячу такие значения `ball.style.left/top`, то в центре будет не сам мяч, а его левый верхний угол:

```
ball.style.left = Math.round(field.clientWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2) + 'px';
```

Вот как это выглядит:



Для того, чтобы центр мяча находился в центре поля, нам нужно сместить мяч на половину его ширины влево и на половину его высоты вверх:

```
ball.style.left = Math.round(field.clientWidth / 2 - ball.offsetWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight / 2) + 'px';
```

```
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight / 2) +
```

Внимание, подводный камень!

Код выше стабильно работать не будет, потому что `` идёт без ширины/высоты:

```

```

Если браузеру неизвестны ширина и высота изображения (из атрибута HTML-тега или CSS-свойств), он считает их равными `0` до тех пор, пока изображение не загрузится.

При первой загрузке браузер обычно кеширует изображения, так что при последующей загрузке оно будет доступно тут же, вместе с размерами. Но при первой загрузке значение ширины мяча `ball.offsetWidth` равно `0`. Это приводит к вычислению неверных координат.

Мы можем исправить это, добавив атрибуты `width/height` тегу

```
<img> :
```

```

```

...Или задав размеры в CSS:

```
#ball {  
    width: 40px;  
    height: 40px;  
}
```

[Открыть решение в песочнице.](#) ↗

[К условию](#)

В чём отличие CSS-свойств `width` и `clientWidth`

Отличия:

1. `clientWidth` возвращает число, а `getComputedStyle(elem).width` – строку с `px` на конце.
2. `getComputedStyle` не всегда даст ширину, он может вернуть, к примеру, `"auto"` для строчного элемента.

3. `clientWidth` соответствует внутренней области элемента, включая внутренние отступы `padding`, а CSS-ширина (при стандартном значении `box-sizing`) соответствует внутренней области без внутренних отступов `padding`.
4. Если есть полоса прокрутки, и для неё зарезервировано место, то некоторые браузеры вычитают его из CSS-ширины (т.к. оно больше недоступно для содержимого), а некоторые – нет. Свойство `clientWidth` всегда ведёт себя одинаково: оно всегда обозначает размер за вычетом прокрутки, т.е. реально доступный для содержимого.

[К условию](#)

Координаты

Найдите координаты точек относительно окна браузера

Внешние углы

Координаты внешних углов – это как раз то, что возвращает функция `elem.getBoundingClientRect()`.

Координаты верхнего левого внешнего угла будут в переменной `answer1` и нижнего правого – в `answer2`:

```
let coords = elem.getBoundingClientRect();

let answer1 = [coords.left, coords.top];
let answer2 = [coords.right, coords.bottom];
```

Верхний левый внутренний угол

Тут значения отличаются на ширину рамки. Надёжный способ получить интересующее значение – это использовать `clientLeft/clientTop`:

```
let answer3 = [coords.left + field.clientLeft, coords.top + field.clientTop];
```

Нижний правый внутренний угол

В нашем случае нужно вычесть размеры рамки из внешних координат.

Это может быть сделано с помощью CSS:

```
let answer4 = [
    coords.right - parseInt(getComputedStyle(field).borderRightWidth),
    coords.bottom - parseInt(getComputedStyle(field).borderBottomWidth)
];
```

Другим вариантом решения было бы добавление `clientWidth/clientHeight` к координатам верхнего левого угла. Так даже было бы лучше.

```
let answer4 = [
    coords.left + elem.clientLeft + elem.clientWidth,
    coords.top + elem.clientTop + elem.clientHeight
];
```

[Открыть решение в песочнице.](#)

[К условию](#)

Покажите заметку рядом с элементом

В этой задаче нам нужно только аккуратно вычислить координаты. Смотрите код для изучения деталей реализации.

Обратите внимание, что элементы должны уже быть в документе перед чтением `offsetHeight` и других свойств. Спрятанный (`display:none`) элемент или элемент вне документа не имеют размеров.

[Открыть решение в песочнице.](#)

[К условию](#)

Покажите заметку около элемента (абсолютное позиционирование)

Решение достаточно простое:

- Используйте `position:absolute` в CSS вместо `position:fixed` для элемента с классом `.note`.
- Используйте функцию `getCoords()` из главы [Координаты](#), чтобы получить координаты относительно документа.

[Открыть решение в песочнице.](#)

[К условию](#)

Расположите заметку внутри элемента (абсолютное позиционирование)

[Открыть решение в песочнице.](#)

[К условию](#)

Введение в браузерные события

Скрыть элемент по нажатию кнопки

[Открыть решение в песочнице.](#)

[К условию](#)

Спрятать себя

Можем использовать `this` в обработчике для доступа к самому элементу:

```
<input type="button" onclick="this.hidden=true" value="Нажми, чтобы спрятать">
```

[К условию](#)

Какой обработчик запустится?

Ответ: `1` и `2`.

Первый обработчик сработает, потому что он не был удалён методом `removeEventListener`. Чтобы удалить обработчик, необходимо передать именно ту функцию, которая была назначена в качестве обработчика. Несмотря на то, что код идентичен, в `removeEventListener` передаётся новая, другая функция.

Для того чтобы удалить функцию-обработчик, нужно где-то сохранить ссылку на неё, например:

```
function handler() {
  alert(1);
}

button.addEventListener("click", handler);
button.removeEventListener("click", handler);
```

Обработчик `button.onclick` сработает независимо от `addEventListener`.

К условию

Передвигните мяч по полю

Сначала мы должны выбрать метод позиционирования мяча.

Мы не можем использовать `position:fixed`, поскольку прокрутка страницы будет перемещать мяч с поля.

Правильнее использовать `position:absolute`, и, чтобы сделать позиционирование действительно надёжным, сделаем само поле (`field`) позиционированным.

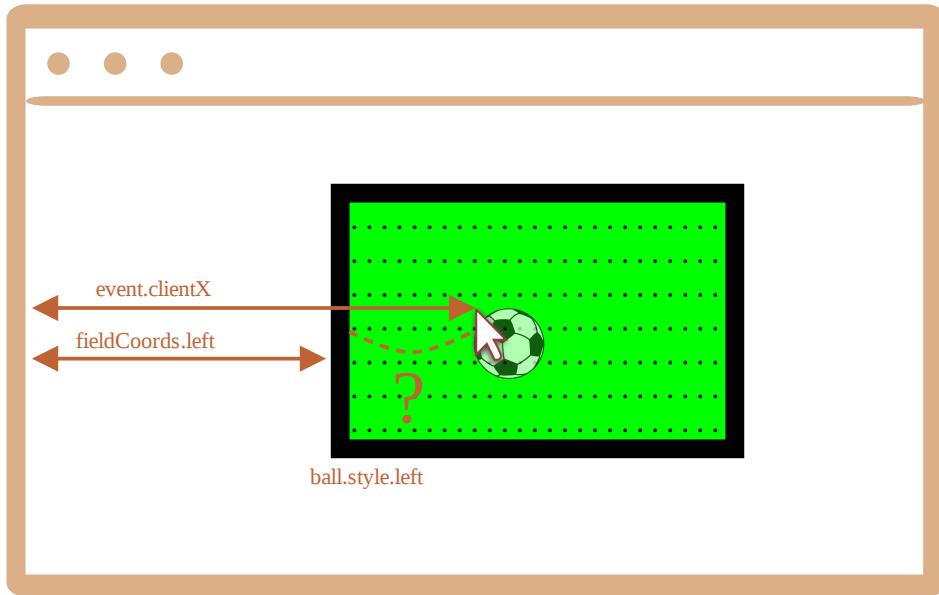
Тогда мяч будет позиционирован относительно поля:

```
#field {
  width: 200px;
  height: 150px;
  position: relative;
}

#ball {
  position: absolute;
  left: 0; /* по отношению к ближайшему расположенному предку (поле) */
  top: 0;
  transition: 1s all; /* CSS-анимация для значений left/top делает передвижение */
}
```

Далее мы должны назначить корректные значения `ball.style.left/top`. Сейчас они содержат координаты относительно поля.

Картинка:



У нас есть значения `event.clientX/clientY` – координаты нажатия мышки относительно окна браузера;

Чтобы получить значение `left` для мяча после нажатия мышки относительно поля, мы должны из координаты нажатия мышки вычесть координату левого края поля и ширину границы:

```
let left = event.clientX - fieldCoords.left - field.clientLeft;
```

Значение `ball.style.left` означает «левый край элемента» (мяча). И если мы назначим такой `left` для мяча, тогда его левая граница, а не центр, будет под курсором мыши.

Нам нужно сдвинуть мяч на половину его высоты вверх и половину его ширины влево, чтобы центр мяча точно совпадал с точкой нажатия мышки.

В итоге значение для `left` будет таким:

```
let left = event.clientX - fieldCoords.left - field.clientLeft - ball.offsetWidth / 2;
```

Вертикальная координата будет вычисляться по той же логике.

Следует помнить, что ширина и высота мяча должна быть известна в тот момент, когда мы получаем значение `ball.offsetWidth`. Это значение может задаваться в HTML или CSS.

[Открыть решение в песочнице.](#)

Создать раскрывающееся меню

HTML/CSS

Для начала создадим разметку HTML/CSS нашего меню.

Меню – это отдельный графический компонент на странице, так что его лучше вынести в отдельный DOM-элемент.

Список пунктов меню может быть представлен в виде списка `ul/li`.

Пример HTML-структуры:

```
<div class="menu">
  <span class="title">Сладости (нажми меня)!</span>
  <ul>
    <li>Пирожное</li>
    <li>Пончик</li>
    <li>Мёд</li>
  </ul>
</div>
```

Для заголовка мы используем тег ``, потому что `<div>`, как и любой блочный элемент, имеет скрытое свойство `display: block`, а значит, занимает ширину 100%.

Например:

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

Сладости (нажми меня)!

Таким образом, если мы зададим обработчик события `onclick`, то он будет срабатывать по клику на всей ширине поля.

...тег `` – строчный элемент, по умолчанию имеет свойство `display: inline`, который занимает ровно столько места, сколько занимает сам текст:

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

Сладости (нажми меня)!

Переключение меню

Переключение меню должно менять стрелку и скрывать или показывать список элементов меню.

Все эти изменения прекрасно обрабатываются средствами CSS.

Посредством JavaScript мы будем отмечать текущее состояние меню, добавляя или удаляя класс `.open`.

Без класса `.open` меню будет закрыто:

```
.menu ul {  
    margin: 0;  
    list-style: none;  
    padding-left: 20px;  
    display: none;  
}  
  
.menu .title::before {  
    content: '▶';  
    font-size: 80%;  
    color: green;  
}
```

...А с ним (с классом `.open`) стрелка будет меняться, и список будет показываться:

```
.menu.open .title::before {  
    content: '▼';  
}  
  
.menu.open ul {  
    display: block;  
}
```

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Добавить кнопку закрытия

Чтобы добавить кнопку закрытия, мы можем использовать либо `position: absolute` (и сделать плитку (`pane`) `position: relative`)

либо `float:right`. Преимущество варианта с `float:right` в том, что кнопка закрытия никогда не перекроет текст, но вариант `position:absolute` даёт больше свободы для действий. В общем, выбор за вами.

Тогда для каждой плитки код может выглядеть следующим образом:

```
pane.insertAdjacentHTML("afterbegin", '<button class="remove-button">[x]</button>');
```

Элемент `<button>` становится `pane.firstChild`, таким образом мы можем добавить на него обработчик события:

```
pane.firstChild.onclick = () => pane.remove();
```

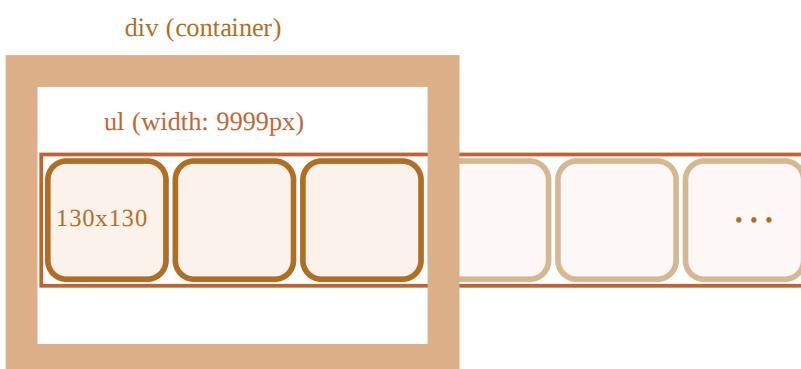
[Открыть решение в песочнице.](#) ↗

[К условию](#)

Карусель

Лента изображений в разметке должна быть представлена как список `ul/li` с картинками ``.

Нужно расположить ленту внутри `<div>` фиксированного размера, так чтобы в один момент была видна только нужная часть списка:

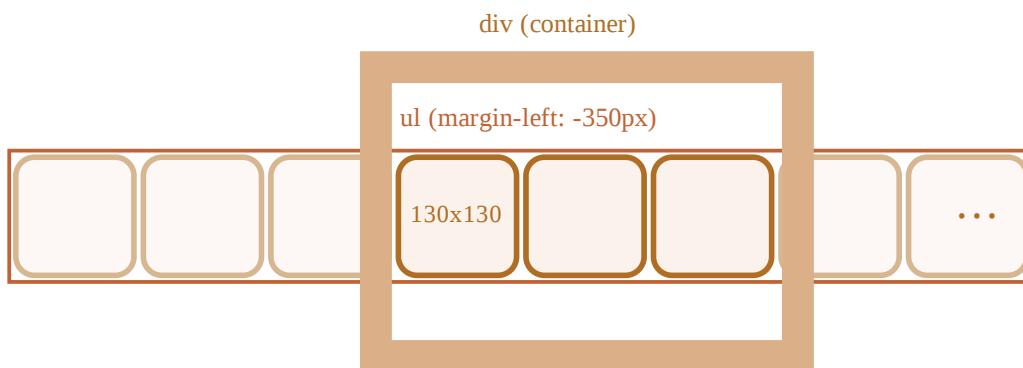


Чтобы список сделать горизонтальным, нам нужно применить CSS-свойство `display: inline-block` для ``.

Для тега `` мы также должны настроить `display`, поскольку по умолчанию он `inline`. Во всех элементах типа `inline` резервируется

дополнительное место под «хвосты» символов. И чтобы его убрать, нам нужно прописать `display: block`.

Для «прокрутки» будем сдвигать ``. Это можно делать по-разному, например, назначением CSS-свойства `transform: translateX()` (лучше для производительности) или `margin-left`:



У внешнего `<div>` фиксированная ширина, поэтому «лишние» изображения обрезаются.

Вся карусель – это самостоятельный «графический компонент» на странице, таким образом нам лучше его «обернуть» в отдельный `<div class="carousel">` и уже модифицировать стили внутри него.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Делегирование событий

Спрячьте сообщения с помощью делегирования

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Раскрывающееся дерево

Решение состоит из двух шагов:

1. Оборачиваем текст каждого заголовка дерева в элемент ``.

Затем мы можем добавить стили CSS на `:hover` и обрабатывать клики только на тексте, т.к. ширина элемента `` в точности совпадает с шириной текста.

2. Устанавливаем обработчик на корневой узел дерева `tree` и ловим клики на элементах ``, содержащих заголовки.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Сортируемая таблица

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Поведение "подсказка"

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Действия браузера по умолчанию

Почему не работает `return false?`

Когда браузер считывает атрибут `on*`, например `onclick`, он создаёт функцию-обработчик с содержимым этого атрибута в качестве тела функции.

Функция для `onclick="handler()"` будет:

```
function(event) {  
    handler() // содержимое onclick  
}
```

Сейчас нам видно, что возвращаемое значение `handler()` не используется и не влияет на результат.

Исправить очень просто:

```
<script>
  function handler() {
    alert("...");
    return false;
  }
</script>

<a href="https://w3.org" onclick="return handler()">w3.org</a>
```

Также мы можем использовать `event.preventDefault()`, например:

```
<script>
  function handler(event) {
    alert("...");
    event.preventDefault();
  }
</script>

<a href="https://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 ушёл посетитель. Или мы можем загрузить содержимое и отобразить его прямо на странице (если допустимо).

Всё, что нам необходимо, это поймать событие `contents.onclick` и использовать функцию `confirm`, чтобы задать вопрос пользователю. Хорошей идеей было бы использовать `link.getAttribute('href')` вместо `link.href` для ссылок. Смотрите решение в песочнице.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Галерея изображений

Решение состоит в том, чтобы добавить обработчик на контейнер `#thumbs` и отслеживать клики на ссылках. Если клик происходит по ссылке `<a>`, тогда меняем атрибут `src` элемента `#largeImg` на `href` уменьшенного изображения.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Основы событий мыши

Выделяемый список

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Движение мыши: mouseover/out,mouseenter/leave

Улучшенная подсказка

[Открыть решение в песочнице.](#) ↗

[К условию](#)

"Умная" подсказка

Алгоритм выглядит просто:

1. Назначаем обработчики `onmouseover/out` на элементе. Также можно было бы использовать `onmouseenter/leave`, но они менее универсальны и не сработают с делегированием.
2. Когда курсор переходит на элемент, начинаем измерять скорость его движения, используя `mousemove`.
3. Если скорость низкая, то вызываем `over`.
4. Когда мы выходим из элемента, если запускали `over`, вызываем `out`.

Но как измерить скорость?

Первая идея может быть такой: запускать нашу функцию каждые `100ms` и находить разницу между прежними и текущими координатами курсора. Если она мала, то значит и скорость низкая.

К сожалению, в JavaScript нет возможности получать текущие координаты мыши. Не существует функции типа `получитьТекущиеКоординатыМышь()`.

Единственный путь – это слушать события мыши, например `mousemove`, и координаты брать из объекта события.

Так что поставим обработчик на `mousemove`, чтобы отслеживать координаты и запоминать их. И будем сравнивать результаты каждые `100ms`.

P.S. Обратите внимание: тесты для решения этой задачи используют `dispatchEvent`, чтобы проверить, что подсказка работает корректно.

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Drag'n'Drop с событиями мыши

Слайдер

Как можно видеть из HTML/CSS, слайдер – это `<div>`, подкрашенный фоном/градиентом, внутри которого находится другой `<div>`, оформленный как бегунок, с `position:relative`.

Используем для его координат `position:relative`, т.е. координаты ставятся не абсолютные, а относительно родителя, так как это удобнее.

И дальше реализуем Drag'n'Drop только по горизонтали, с ограничением по ширине.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Расставить супергероев по полю

Чтобы перетащить элемент, мы можем использовать `position:fixed`, это делает управление координатами проще. В конце следует переключиться обратно на `position:absolute`, чтобы положить элемент в документ.

Когда координаты находятся в верхней/нижней части окна, мы используем его `window.scrollTo` для прокрутки.

Детали решения расписаны в комментариях в исходном коде.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Клавиатура: keydown и keyup

Отследить одновременное нажатие

Необходимо использовать два обработчика событий: `document.onkeydown` и `document.onkeyup`.

Создадим множество `pressed = new Set()`, в которое будем записывать клавиши, нажатые в данный момент.

В первом обработчике будем добавлять в него значения, а во втором удалять. Каждый раз, как отрабатывает `keydown`, будем проверять – все ли нужные клавиши нажаты, и, если да – выводить сообщение.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Прокрутка

Бесконечная страница

Основа решения – функция, которая добавляет больше дат на страницу (или загружает больше материала в реальной жизни), пока мы находимся в конце этой страницы.

Мы можем вызвать её сразу же и добавить как обработчик для `window.onscroll`.

Самый важный вопрос: «Как обнаружить, что страница прокручена к самому низу?»

Давайте используем координаты относительно окна.

Документ представлен тегом `<html>` (и содержится в нём же), который доступен как `document.documentElement`.

Так что мы можем получить его координаты относительно окна как `document.documentElement.getBoundingClientRect()`, свойство `bottom` будет координатой нижней границы документа относительно окна.

Например, если высота всего HTML-документа `2000px`, тогда:

```
// когда мы находимся вверху страницы
// координата top относительно окна равна 0
document.documentElement.getBoundingClientRect().top = 0

// координата bottom относительно окна равна 2000
// документ длинный, вероятно, далеко за пределами нижней части окна
document.documentElement.getBoundingClientRect().bottom = 2000
```

Если прокрутить `500px` вниз, тогда:

```
// верх документа находится выше окна на 500px
document.documentElement.getBoundingClientRect().top = -500
// низ документа на 500px ближе
document.documentElement.getBoundingClientRect().bottom = 1500
```

Когда мы прокручиваем до конца, предполагая, что высота окна `600px`:

```
// верх документа находится выше окна на 1400px
document.documentElement.getBoundingClientRect().top = -1400
// низ документа находится ниже окна на 600px
document.documentElement.getBoundingClientRect().bottom = 600
```

Пожалуйста, обратите внимание, что `bottom` не может быть `0`, потому что низ документа никогда не достигнет верха окна. Нижним пределом координаты `bottom` является высота окна (выше мы предположили, что это `600`), больше прокручивать вверх нельзя.

Получить высоту окна можно как
`document.documentElement.clientHeight`.

Для нашей задачи мы хотим знать, когда нижняя граница документа находится не более чем в `100px` от неё (т.е. `600 - 700px`, если высота `600`).

Итак, вот функция:

```
function populate() {
  while(true) {
    // нижняя граница документа
    let windowRelativeBottom = document.documentElement.getBoundingClientRect()

    // если пользователь прокрутил достаточно далеко (< 100px до конца)
    if (windowRelativeBottom < document.documentElement.clientHeight + 100) {
      // добавим больше данных
      document.body.insertAdjacentHTML("beforeend", `<p>Дата: ${new Date()}</p>`)
    }
  }
}
```

[Открыть решение в песочнице.](#)

[К условию](#)

Кнопка вверх/вниз

[Открыть решение в песочнице.](#)

[К условию](#)

Загрузка видимых изображений

Обработчик `onscroll` должен проверить, какие изображения видимы, и показать их.

Мы также можем запустить его при загрузке страницы, чтобы сразу обнаружить видимые изображения и загрузить их.

Код должен выполниться, когда документ загружен, чтобы у него был доступ к его содержимому.

Можно разместить его перед закрывающим тегом `</body>`:

```
// ...содержимое страницы выше...

function isVisible(elem) {
  let coords = elem.getBoundingClientRect();
  let windowHeight = document.documentElement.clientHeight;
  // верхний край элемента виден?
  let topVisible = coords.top > 0 && coords.top < windowHeight;
  // нижний край элемента виден?
  let bottomVisible = coords.bottom < windowHeight && coords.bottom > 0;
  return topVisible || bottomVisible;
}
```

Функция `showVisible()` использует проверку на видимость, реализованную в `isVisible()` для загрузки видимых картинок:

```
function showVisible() {
  for (let img of document.querySelectorAll('img')) {
    let realSrc = img.dataset.src;
    if (!realSrc) continue;

    if (isVisible(img)) {
      img.src = realSrc;
      img.dataset.src = '';
    }
  }

  showVisible();
  window.onscroll = showVisible;
```

P.S. В решении этой задачи есть также вариант `isVisible`, который предварительно загружает изображения, находящиеся в пределах одной страницы выше/ниже от текущей прокрутки документа.

[Открыть решение в песочнице.](#)

[К условию](#)

Свойства и методы формы

Добавьте пункт к выпадающему списку

Решение шаг за шагом:

```
<select id="genres">
  <option value="rock">Рок</option>
  <option value="blues" selected>Блюз</option>
</select>

<script>
// 1)
let selectedOption = genres.options[genres.selectedIndex];
alert( selectedOption.value );
alert( selectedOption.text );

// 2)
let newOption = new Option("Классика", "classic");
genres.append(newOption);

// 3)
newOption.selected = true;
</script>
```

[К условию](#)

Фокусировка: focus/blur

Редактируемый div

[Открыть решение в песочнице.](#)

[К условию](#)

Редактирование TD по клику

1. По клику – заменить `innerHTML` ячейки на `<textarea>` с теми же размерами и без рамки. Можно использовать JavaScript или CSS, чтобы установить правильный размер.
2. Присвоить `textarea.value` значение `td.innerHTML`.
3. Установить фокус на текстовую область.
4. Показать кнопки ОК/ОТМЕНА под ячейкой, обрабатывать клики по ним.

[Открыть решение в песочнице.](#)

[К условию](#)

Мышь, управляемая клавиатурой

Мы можем использовать `mouse.onclick` для обработки клика и сделать мышь «перемещаемой» с помощью `position:fixed`, а затем использовать `mouse.onkeydown` для обработки клавиш со стрелками.

Единственная проблема в том, что `keydown` срабатывает только на элементах с фокусом. И нам нужно добавить `tabindex` к элементу. Так как изменять HTML запрещено, то для этого мы можем использовать свойство `mouse.tabIndex`.

P.S. Мы также можем заменить `mouse.onclick` на `mouse.onfocus`.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

События: change, input, cut, copy, paste

Депозитный калькулятор

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Отправка формы: событие и метод submit

Модальное диалоговое окно с формой

Модальное окно может быть реализовано с помощью полупрозрачного `<div id="cover-div">`, который полностью перекрывает всё окно:

```
#cover-div {  
    position: fixed;  
    top: 0;  
    left: 0;  
    z-index: 9000;  
    width: 100%;
```

```
height: 100%;  
background-color: gray;  
opacity: 0.3;  
}
```

Так как он перекрывает вообще всё, все клики будут именно по этому `<div>`.

Также возможно предотвратить прокрутку страницы, установив `body.style.overflowY='hidden'`.

Форма должна быть не внутри `<div>`, а после него, чтобы она не унаследовала полупрозрачность (`opacity`).

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Загрузка ресурсов: `.onload` и `onerror`

Загрузите изображения с колбэком

Алгоритм:

1. Создадим `img` для каждого ресурса.
2. Добавим обработчики `onload/onerror` для каждого изображения.
3. Увеличиваем счётчик при срабатывании `.onload` или `onerror`.
4. Когда значение счётчика равно количеству ресурсов – тогда вызываем `callback()`.

[Открыть решение в песочнице.](#) ↗

[К условию](#)

Событийный цикл: микрозадачи и макрозадачи

Что код выведет в консоли?

1. Создание промиса
2. Конец скрипта

3. Обработка промиса

4. Таймаут

Давайте разберем что здесь происходит.

Изначально в стеке выполнения находится сам скрипт, поэтому сначала выполняется только он.

В первой строке появляется `setTimeout`, который ставит переданный колбэк в очередь макрозадач (macrotask queue) на выполнение.

После этого в переменную `r` запишется промис. Стоит отметить, что создание промиса в данном случае происходит синхронно. Это значит, что код из переданного колбэка выполнится прямо сейчас. В результате в консоль выведется `'Создание промиса'`.

Далее мы уведомляем потребителя `then`, что хотели бы выполнить переданную функцию после успешного выполнения промиса. Так как промис уже имеет состояние `fulfilled` (мы вызвали `resolve()` при его создании), колбэк из `then` будет немедленно передан в очередь микрозадач (microtask queue) на выполнение.

В конце выполнения скрипта выводится `'Конец скрипта'`.

Скрипт является макрозадачей. Как мы уже знаем, после завершения каждой задачи опустошается очередь микрозадач. В ней находится только ранее переданный в `then` колбэк. В результате его выполнения в консоль выведется `'Обработка промиса'`.

Так как очередь микрозадач опустела, можно продолжить выполнять код из очереди макрозадач. Там сейчас находится только колбэк, который мы передавали `setTimeout`. После его выполнения выводится `'Таймаут'`.

[К условию](#)

Что код выведет в консоли?

Вывод в консоли: 1 7 3 5 2 6 4.

Задача довольно простая, нужно лишь понимать, как работают очереди микрозадач и макрозадач.

Давайте разберем, что здесь происходит, по шагам.

```

console.log(1);
// Первая строка выполняется сразу и выводит `1`.
// Очереди микрозадач и макрозадач на данный момент пусты.

setTimeout(() => console.log(2));
// `setTimeout` ставит переданный колбэк в очередь макрозадач
// - содержимое очереди макrozадач:
//   `console.log(2)`

Promise.resolve().then(() => console.log(3));
// В очередь микрозадач ставится колбэк, выводящий `3`
// - содержимое очереди микрозадач:
//   `console.log(3)`

Promise.resolve().then(() => setTimeout(() => console.log(4)));
// В очередь микрозадач ставится колбэк с `setTimeout`
// - содержимое очереди микрозадач:
//   `console.log(3); setTimeout(...4)`

Promise.resolve().then(() => console.log(5));
// В очередь микрозадач ставится колбэк, выводящий `5`
// - содержимое очереди микрозадач:
//   `console.log(3); setTimeout(...4); console.log(5)`

setTimeout(() => console.log(6));
// `setTimeout` ставит переданный колбэк в очередь макрозадач
// - содержимое очереди макrozадач:
//   `console.log(2); console.log(6)`

console.log(7);
// Тут же выводит `7`.

```

Итак, получается, что:

1. Числа `1` и `7` выводятся сразу же, так как они не используют очереди задач вообще.
2. Далее после окончания основного потока кода срабатывает очередь микрозадач.
 - Её содержимое: `console.log(3); setTimeout(...4); console.log(5)`.
 - Выведется `3` и `5`, а `setTimeout(() => console.log(4))` поставит в конец очереди макрозадач вывод `4`.
 - В очереди макрозадач получается теперь: `console.log(2); console.log(6); console.log(4)`.
3. Очередь микрозадач полностью выполнена, срабатывает очередь макрозадач. Она выведет `2, 6, 4`.

Получается вывод `1 7 3 5 2 6 4`.

[К условию](#)