**Тема 7. События**

1. **Браузерные события.**
2. **Всплытие и погружение.**
3. **Делегирование событий.**
4. **Приём проектирования "поведение".**
5. **Генерация событий.**
6. **События мыши.**
7. **Мышь: mouseover/out, mouseenter/leave**
8. **Drag'n'Drop с событиями мыши**
9. **Прокрутка: событие scroll.**
10. **Клавиатура: keyup, keydown.**
11. **Загрузка документа.**
12. **Загрузка скриптов.**
13. **Загрузка ресурсов.**

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **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). Ниже перечисляются наиболее часто используемые DOM-событий.

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.

[**Обработчики событий**](https://learn.javascript.ru/introduction-browser-events#obrabotchiki-sobytiy)

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.

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 поставить 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!">

Как известно, 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.

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>

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">

1. HTML + JS:

<input type="button" id="button" value="Кнопка">

<script>

button.onclick = function() {

alert('Клик!');

};

</script>

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() {

alert('Станет');

};

</script>

Кстати, обработчиком можно назначить и уже существующую функцию:

function sayThanks() {

alert('Спасибо!');

}

elem.onclick = sayThanks;

Убрать обработчик можно назначением elem.onclick = null.

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>

Обратите внимание, что 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();

};

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, так как атрибуты всегда строки, и функция станет строкой:

document.body.setAttribute('onclick', function() { alert(1) });

Используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

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, следовательно, надо назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); }

Решить эту проблему позволяют методы addEventListener и removeEventListener.

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 вопросе о [всплытии и погружени](https://learn.javascript.ru/bubbling-and-capturing)и событий). options может быть false/true, это тоже самое, что {capture: false/true}.
* passive: если true, то указывает, что обработчик никогда не вызовет preventDefault() (подробнее это будет рассматриваться в вопросе о [действии браузера по умолчанию](https://learn.javascript.ru/default-browser-action)).

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler[, options]);

Для удаления обработчика 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. Вот так правильно:

function handler() {

alert( 'Спасибо!' );

}

input.addEventListener("click", handler);

input.removeEventListener("click", handler);

Обратим внимание – если функцию обработчик не сохранить где-либо, нельзя будет 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.

Существуют события, которые нельзя назначить через DOM-свойство, но можно через addEventListener. Например, таково событие transitionend, то есть окончание CSS-анимации. Код ниже демонстрирует это. В большинстве браузеров, сработает лишь второй обработчик, но не первый:

<style>

input {

transition: width 1s;

width: 100px;

}

.wide {

width: 300px;

}

</style>

<input type="button" id="elem" onclick="this.classList.toggle('wide')" value="Нажми меня">

<script>

elem.ontransitionend = function() {

alert("DOM property"); // не сработает

};

elem.addEventListener("transitionend", function() {

alert("addEventListener"); // сработает по окончании анимации

});

</script>

[**Объект события**](https://learn.javascript.ru/introduction-browser-events#obekt-sobytiya)

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 можно 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 будут рассмотрены далее.

При назначении обработчика в HTML, тоже можно использовать объект event, вот так:

<input type="button" onclick="alert(event.type)" value="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.

Можно 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. Также можно 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, нужно 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.

1. Всплытие и погружение.

Рассмотрим пример. Этот обработчик для <div> сработает, если вы кликните по любому из вложенных тегов, будь то <em> или <code>:

<div onclick="alert('Обработчик!')">

<em>Если вы кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>

</div>

**[Всплытие](https://learn.javascript.ru/bubbling-and-capturing" \l "vsplytie)**

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>

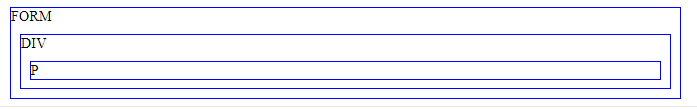
<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

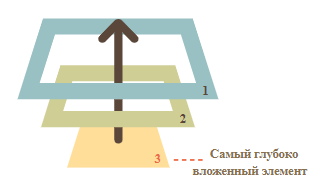
</div>

</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 появятся 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. Не все события всплывают. Например, событие focus не всплывает. Однако, это исключение, всё-таки большинство событий всплывают.

[**event.target**](https://learn.javascript.ru/bubbling-and-capturing#event-target)

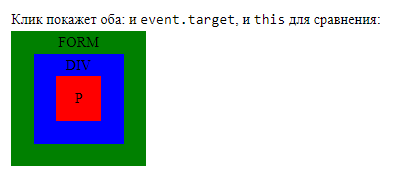
Всегда можно узнать, на каком конкретно элементе произошло событие. Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target. Отличия от this (=event.currentTarget):

* 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 обработчикаform.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.

Результат



// script.js

form.onclick = function(event) {

event.target.style.backgroundColor = 'yellow';

// браузеру нужно некоторое время, чтобы зарисовать всё жёлтым

setTimeout(() => {

alert("target = " + event.target.tagName + ", this=" + this.tagName);

event.target.style.backgroundColor = ''

}, 0);

};

// example.css

form {

background-color: green;

position: relative;

width: 150px;

height: 150px;

text-align: center;

cursor: pointer;

}

div {

background-color: blue;

position: absolute;

top: 25px;

left: 25px;

width: 100px;

height: 100px;

}

p {

background-color: red;

position: absolute;

top: 25px;

left: 25px;

width: 50px;

height: 50px;

line-height: 50px;

margin: 0;

}

body {

line-height: 25px;

font-size: 16px;

}

// index.html

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<link rel="stylesheet" href="example.css">

</head>

<body>

Клик покажет оба: и <code>event.target</code>, и <code>this</code> для сравнения:

<form id="form">FORM

<div>DIV

<p>P</p>

</div>

</form>

<script src="script.js"></script>

</body>

</html>

Возможна и ситуация, когда event.target и this – один и тот же элемент, например, если клик был непосредственно на самом элементе <form>, не на его подэлементе.

[**Прекращение всплытия**](https://learn.javascript.ru/bubbling-and-capturing#prekraschenie-vsplytiya)

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать 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>

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. Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами. Например:

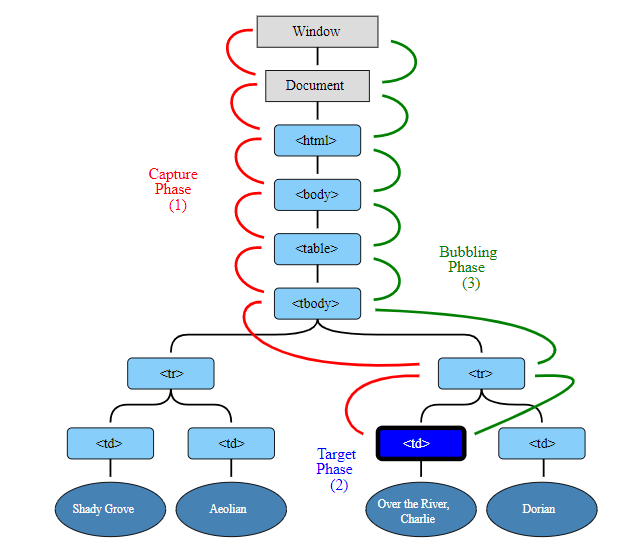
1. При создании вложенного меню, каждое подменю обрабатывает клики на своих элементах и делает для них stopPropagation, чтобы не срабатывало внешнее меню.
2. Позже было решено отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще кликают пользователи. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.
3. Аналитика не будет работать над областью, где клики прекращаются stopPropagation, получилась «мёртвая зона».

[**Погружение**](https://learn.javascript.ru/bubbling-and-capturing#pogruzhenie)

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](http://www.w3.org/TR/DOM-Level-3-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.

Картинка 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.

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. 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>

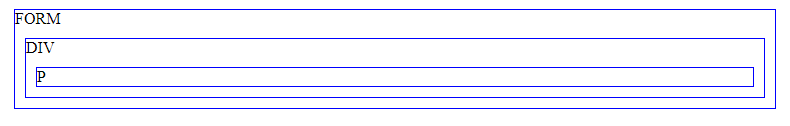
for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));

}

</script>



Здесь обработчики назначаются каждому элементу в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события. Если кликнуть 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 выведется 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, обычно это и так известно в обработчике.

Если обработчик добавлен с помощью addEventListener(..., true), то надо передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

Если есть 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));

1. Делегирование событий.

Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – *делегирование*. Идея в том, что если есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, назначается 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 –  таблица. 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. Задача – реализовать подсветку ячейки <td> при клике. Вместо того, чтобы назначать обработчик onclick для каждой ячейки <td> (их может быть очень много) – назначим «единый» обработчик на элемент <table>. Он будет использовать event.target, чтобы получить элемент, на котором произошло событие, и подсветить его. Код будет таким:

let selectedTd;

table.onclick = function(event) {

let target = event.target;

if (target.tagName != 'TD') return;

highlight(target);

};

function highlight(td) {

if (selectedTd) { // убрать существующую подсветку, если есть

selectedTd.classList.remove('highlight');

}

selectedTd = td;

selectedTd.classList.add('highlight'); // подсветить новый td

}

Такому коду нет разницы, сколько ячеек в таблице. Можно добавлять, удалять <td> из таблицы динамически в любое время, и подсветка будет стабильно работать. Однако, у текущей версии кода есть недостаток. Клик может быть не на теге <td>, а внутри него. В рассматриваемом примере, если взглянуть на HTML-код таблицы внимательно, видно, что ячейка <td> содержит вложенные теги, например, <strong>:

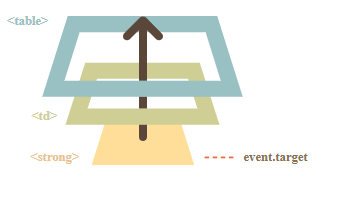
<td>

<strong>Северо-Запад</strong>

...

</td>

Естественно, если клик произойдёт на элементе <strong>, то он станет значением event.target.



Внутри обработчика table.onclick нужно с помощью event.target определить, 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)

};

Рассмотрим пример:

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае 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 надо проверить действительно ли это <td> рассматриваемой таблицы.
4. И если это так, то подсветить его.

В итоге получится короткий код подсветки, быстрый и эффективный, не зависящий от того, сколько всего в таблице <td>.

Есть и другие применения делегирования. Например, нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами save, load, search. Надо 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. 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 нужно.

Здесь преимущество делегирования заключается в следующем:

* 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. Можно добавлять/удалять кнопки в любое время.

Также можно использовать классы .action-save, .action-load, но подход с использованием атрибутов data-action является более семантичным. Их можно использовать и для стилизации в правилах CSS.

[**Приём проектирования «поведение»**](https://learn.javascript.ru/event-delegation#priyom-proektirovaniya-povedenie)

Делегирование событий можно использовать для добавления элементам «поведения» (behavior), декларативно задавая обработчики установкой специальных HTML-атрибутов и классов. Приём проектирования «поведение» состоит из двух частей:

1. Элементу устанавливается пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования назначается обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут – производит соответствующее действие.

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>





Если нажать на кнопку – значение увеличится.

Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования, фактически, добавляется новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Когда устанавливается обработчик событий на объект document, всегда надо 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.

Ещё один пример поведения: при клике на элемент с атрибутом data-toggle-id скрывается/показывается 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>





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.

Можно комбинировать несколько вариантов поведения на одном элементе. Шаблон «поведение» может служить альтернативой для фрагментов JS-кода в вёрстке.

1. 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 событие обрабатывается в JavaScript, то зачастую такое действие браузера не нужно. Его можно отменить.

**[Отмена действия браузера](https://learn.javascript.ru/default-browser-action" \l "otmena-deystviya-brauzera)**

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 произойдет:

<a href="/" onclick="return false">Нажми здесь</a>

или

<a href="/" onclick="event.preventDefault()">здесь</a>

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.

Рассмотрим пример меню для сайта:

<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:



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 использовать <button> или <span>, то данное сочетание работать не будет;
* поисковые движки переходят по ссылкам <a href="..."> при индексации.

Поэтому в разметке используется <a>. Но в примере требуется 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 );

return false; // отменить действие браузера (переход по ссылке)

};

Если убрать return false, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из href. А это здесь не нужно, так как клик обрабатывается.

Использование здесь делегирования событий делает меню очень гибким. Можно добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

Некоторые события естественным образом вытекают друг из друга. Если отменить первое событие, то последующие не возникнут. Например, событие mousedown для поля <input> приводит к фокусировке на нём и запускает событие focus. Если отменить событие mousedown, то фокусирования не произойдёт.

В следующем примере если нажать на первом <input> – произойдет событие focus. Но если нажать 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 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.

Необязательная опция passive: true для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault(). Это может быть полезно, так как есть некоторые события, как touchmove на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но можно 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.

Свойство event.defaultPrevented установлено в true, если действие по умолчанию было предотвращено, и false, если нет. Рассмотрим практическое применение этого свойства для улучшения архитектуры. event.defaultPrevented можно использовать вместо  остановки всплытия с помощью event.stopPropagation(), чтобы просигналить другим обработчикам, что событие обработано.

Рассмотрим практический пример. По умолчанию браузер при событии contextmenu (клик правой кнопкой мыши) показывает контекстное меню со стандартными опциями. Можно 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>



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");

};

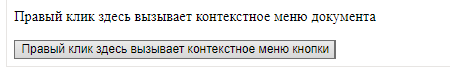
document.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Проблема заключается в том, что когда происходит клик по элементу elem, то получаем два меню: контекстное меню для кнопки и (событие всплывает вверх) контекстное меню для документа. Чтобы это исправить, надо остановить всплытие когда обрабатывается правый клик в обработчике на кнопке, 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");

};

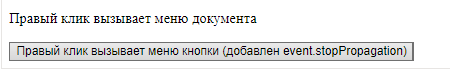
document.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Теперь контекстное меню для кнопки работает как задумано. Но навсегда запрещен доступ к информации о правых кликах для любого внешнего кода, включая счётчики, которые могли бы собирать статистику, и т.п. Это неудобно.

Альтернативным решением было бы проверить в обработчике document, было ли отменено действие по умолчанию? Если да, тогда событие было обработано, и 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) {

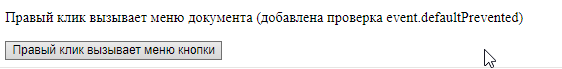
if (event.defaultPrevented) return;

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Сейчас всё работает правильно. Если есть вложенные элементы и каждый из них имеет контекстное меню, то код также будет работать. Просто надо убедиться, что осуществляется проверка event.defaultPrevented в каждом обработчике contextmenu.

event.stopPropagation() и event.preventDefault() (также известный как return false) – это две разные функции. Они никак не связаны друг с другом.

1. Генерация 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 меню может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. Также можно генерировать встроенные события, такие как click, mousedown и другие, что бывает полезно для автоматического тестирования.

[**Конструктор Event**](https://learn.javascript.ru/dispatch-events#konstruktor-event)

Встроенные классы для событий формируют иерархию, аналогично классам для DOM-элементов. Её корнем является встроенный класс [Event](http://www.w3.org/TR/dom/#event). Событие встроенного класса Event можно создать так:

let event = new Event(type[, options]);

Где:

* type – тип события, строка, например, "click" или другая;
* options – объект с двумя необязательными свойствами:
* bubbles: true/false – если true, тогда событие всплывает;
* cancelable: true/false – если true, тогда можно отменить действие по умолчанию.

По умолчанию оба свойства установлены в false: {bubbles: false, cancelable: false}.

После того, как объект события создан, надо запустить событие на элементе, вызвав метод elem.dispatchEvent(event). Затем обработчики отреагируют на него, как будто это обычное встроенное событие. Если при создании указан флаг bubbles, то оно будет всплывать. В примере ниже событие click инициируется JavaScript-кодом так же, как если бы кликнули по кнопке:

<button id="elem" onclick="alert('Клик!');">Автоклик</button>

<script>

let event = new Event("click");

elem.dispatchEvent(event);

</script>

В то же время 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.

Можно 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>

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);

</script>

Надо использовать addEventListener для собственных событий, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает. Надо передавать флаг bubbles:true, иначе 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.

**Конструкторы** [**MouseEvent, KeyboardEvent и другие**](https://learn.javascript.ru/dispatch-events#mouseevent-keyboardevent-i-drugie)

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](https://www.w3.org/TR/uievents): UIEvent, FocusEvent, MouseEvent, WheelEvent, KeyboardEvent.

Стоит использовать их вместо new Event, если надо 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: это нельзя было бы сделать с обычным конструктором Event. 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 можно найти в спецификации, например для [MouseEvent](https://www.w3.org/TR/uievents/#mouseevent).

**[Пользовательские события](https://learn.javascript.ru/dispatch-events" \l "polzovatelskie-sobytiya)**

Для генерации пользовательских событий, таких как "hello", следует использовать конструктор new CustomEvent. Технически [CustomEvent](https://dom.spec.whatwg.org/#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>

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. Класс события содержит информацию о том, что это за событие, и если оно не браузерное, а пользовательское, то стоит всё-таки использовать CustomEvent.

На сгенерированном событии обработчик может вызвать метод event.preventDefault(), если задан флаг cancelable:true. Для пользовательских событий, названия которых браузеру неизвестны, соответственно, нет никаких действий браузера по умолчанию. Но код, который генерирует событие, может предусматривать какие-то ещё действия после dispatchEvent.

Вызов event.preventDefault() является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия надо отменить. Тогда вызов elem.dispatchEvent(event) возвратит false. И код, сгенерировавший событие, приостановит выполнение.

В примере ниже есть функция hide(), которая при вызове генерирует событие "hide" на элементе #rabbit, уведомляя всех интересующихся, что кролик собирается спрятаться. Любой обработчик может узнать об этом, подписавшись на событие hide через rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:

<pre id="rabbit">

|\ /|

\|\_|/

/. .\

=\\_Y\_/=

{>o<}

</pre>

<script>

function hide() {

let event = new CustomEvent("hide", {

cancelable: true

});

if (!rabbit.dispatchEvent(event)) {

alert('действие отменено обработчиком');

} else {

rabbit.hidden = true;

}

}

rabbit.addEventListener('hide', function(event) {

if (confirm("Вызвать preventDefault?")) {

event.preventDefault();

}

});

setTimeout(hide, 2000);

</script>

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);

menu.dispatchEvent(new CustomEvent("menu-open", {

bubbles: true

}));

alert(2);

};

document.addEventListener('menu-open', () => alert('вложенное событие'))

</script>

Обратите внимание, что вложенное событие menu-open всплывает и обрабатывается 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 случаев. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Также можно либо поместить dispatchEvent (или любой другой код инициирующий события) в конец обработчика onclick, либо обернуть такой код в setTimeout(..., 0):

<button id="menu">Меню (нажми меня)</button>

<script>

menu.onclick = function() {

alert(1);

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.

1. События мыши.

События мыши происходят не только от манипуляций мышью, но и эмулируются на сенсорных устройствах, чтобы сделать их совместимыми.

[**Типы событий мыши**](https://learn.javascript.ru/mouse-events-basics#tipy-sobytiy-myshi)

Можно разделить события мыши на две категории: простые и комплексные.

[Простые события](https://learn.javascript.ru/mouse-events-basics" \l "prostye-sobytiya) (наиболее часто используемые):

* 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.

[Комплексные события](https://learn.javascript.ru/mouse-events-basics" \l "kompleksnye-sobytiya):

* click – вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши;
* contextmenu – вызывается при mousedown правой кнопкой мыши;
* dblclick – вызывается двойным кликом на элементе.

Комплексные события состоят из простых. С ними удобнее работать.

[**Порядок событий**](https://learn.javascript.ru/mouse-events-basics#poryadok-sobytiy)

Одно действие 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. События обрабатываются в той же последовательности: onmouseup завершается до того, как запускается onclick.

События, связанные с кликом, всегда имеют свойство which, которое позволяет определить нажатую кнопку мыши. Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой. События mousedown и mouseup срабатывают на любой кнопке и свойство which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки» мыши.

Есть три возможных значения:

* event.which == 1 – левая кнопка;
* event.which == 2 – средняя кнопка;
* event.which == 3 – правая кнопка.

[**Модификаторы: shift, alt, ctrl и meta**](https://learn.javascript.ru/mouse-events-basics#modifikatory-shift-alt-ctrl-i-meta)

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах. Их свойства: shiftKey, altKey, ctrlKey, metaKey (Cmd для Mac). 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>



В Windows и Linux клавишами-модификаторами являются Alt, Shift и Ctrl. На Mac есть ещё одна: Cmd, она соответствует свойству metaKey. В большинстве случаев, когда в Windows/Linux используется Ctrl, на Mac пользователи используют Cmd. Поэтому, когда пользователь Windows нажимает Ctrl+Enter и Ctrl+A, пользователь Mac нажимает Cmd+Enter или Cmd+A, и так далее, большинство приложений используют Cmd вместо Ctrl. Поэтому, если надо поддерживать такие комбинации, как Ctrl+клик, то для Mac имеет смысл использовать Cmd+клик. Это удобней для пользователей Mac.

Левый клик в сочетании с Ctrl интерпретируется как правый клик на Mac и генерирует событие contextmenu, а не click как на Windows/Linux. Поэтому, если надо, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нужно использовать metaKey. Для JS-кода это означает, что надо проверить if (event.ctrlKey || event.metaKey).

Все события мыши имеют координаты двух видов: относительно окна (event.clientX и event.clientY) и относительно документа (event.pageX и event.pageY).

Клики мышью имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: двойной клик мышью выделяет текст. Это действие браузера по умолчанию при наступлении события mousedown. Поэтому альтернативным решением проблемы будет обработать событие mousedown и предотвратить его:

<b ondblclick="alert('Клик!')" onmousedown="return false">

Сделайте двойной клик на мне

</b>

Теперь выделенный жирным элемент не выделяется при двойном клике. Текст внутри него по-прежнему можно выделить. Однако, выделение должно начаться не на самом тексте, а до него или после.

Вместо предотвращения выделения, можно отменить его «постфактум» в обработчике событий. Например, так:

<b ondblclick="getSelection().removeAllRanges()">

Сделайте двойной клик на мне

</b>

При двойном клике на элементе, выделенном жирным шрифтом, выделение появится и тут же будет немедленно снято. Выглядит это не очень красиво.

Если надо отключить выделение для защиты контента от копирования, то можно использовать другое событие: oncopy.

<div oncopy="alert('Копирование запрещено!');return false">

Уважаемый пользователь,

Копирование информации запрещено.

</div>

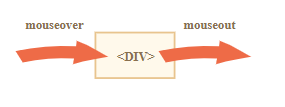
Скопировать текст в <div> не получится, потому что срабатывание события oncopy по умолчанию запрещено.

1. Мышь: mouseover/out, mouseenter/leave

Рассмотрим подробнее события, возникающие при движении указателя (курсора) мыши над элементами страницы.

[**Mouseover/mouseout, relatedTarget**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#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 храниться в target, а другой в relatedTarget.

Для события mouseover:

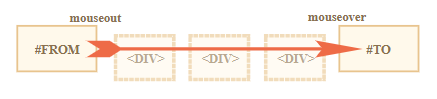
* event.target – это элемент, на который курсор перешёл;
* event.relatedTarget – это элемент, с которого курсор ушёл.

Для события mouseout наоборот:

* event.target – это элемент, с которого курсор ушёл;
* event.relatedTarget – это элемент, на который курсор перешёл.

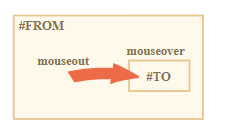
Свойство relatedTarget может быть null, это означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или, наоборот, ушёл за пределы окна.

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 двигается очень быстро с #FROM на #TO элемент, 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.

Представьте ситуацию – курсор мыши перешёл на элемент. Сгенерировано событие mouseover. Затем курсор перешёл на дочерний элемент, и сгенерировано mouseout. То есть курсор всё ещё на элементе, но событие mouseout.



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.

[**События mouseenter и mouseleave**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#sobytiya-mouseenter-i-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 по дочерним элементам не считаются.
2. События mouseenter/mouseleave не всплывают.

Когда курсор становится над элементом – генерируется mouseenter, и не имеет значения, где именно находится курсор внутри элемента. Событие mouseleave происходит, когда курсор покидает элемент.

События mouseenter/leave простые и легкие в использовании, но они не всплывают, а это значит что их нельзя делегировать.

1. Drag'n'Drop с событиями мыши

События Drag’n’Drop генерируются, когда пользователь захватывает элемент мышкой и перетаскивает его. Основной алгоритм Drag’n’Drop выглядит так:

1. Отслеживание события mousedown на переносимом элементе.
2. При нажатии – подготовка элемента к перемещению (например, создание его копии).
3. При наступлении mousemove перемещение элемента на новые координаты путём смены left/top и position:absolute.
4. На mouseup (при отпускании кнопки мыши) – остановка переноса элемента.

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

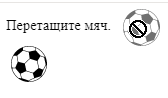
ball.onmouseup = function() {

document.removeEventListener('mousemove', onMouseMove);

ball.onmouseup = null;

};

};

Если запустить этот код, то можно заметим, что при начале переноса мяч «раздваивается» и переносится не сам мяч, а его копия. 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;

};

Ещё одна особенность правильного Drag’n’Drop – событие mousemove отслеживается на document, а не на ball. С первого взгляда кажется, что мышь всегда над мячом и обработчик mousemove можно повесить на сам мяч, а не на документ. Но не стоит забывать, что событие mousemove возникает хоть и часто, но не для каждого пикселя. Поэтому из-за быстрого движения курсор может сместиться с мяча и оказаться где-нибудь в середине документа (или даже за пределами окна). Поэтому стоит отслеживать mousemove на всём document, чтобы поймать его.

В примерах выше мяч позиционируется так, что его центр оказывается под курсором мыши:

ball.style.left = pageX - ball.offsetWidth / 2 + 'px';

ball.style.top = pageY - ball.offsetHeight / 2 + 'px';

У такого позиционирования есть один недостаток: в самом начале переноса, особенно если мячик «взят» за край – он резко «прыгает» центром под курсор мыши. Было бы лучше, если бы изначальный сдвиг курсора относительно элемента сохранялся, т.е. где захватили, за ту «часть элемента» и переносим:

1. Когда пользователь нажимает на мячик (mousedown) –нужно зафиксировать расстояние от курсора до левого верхнего угла шара в переменных shiftX/shiftY и запомнать это расстояние при перетаскивании.

Чтобы получить этот сдвиг, можно вычесть координаты:

// onmousedown

let shiftX = event.clientX - ball.getBoundingClientRect().left;

let shiftY = event.clientY - ball.getBoundingClientRect().top;

1. Далее при переносе он позиционируется с тем же сдвигом относительно указателя мыши (position:absoute):

// onmousemove

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);

function moveAt(pageX, pageY) {

ball.style.left = pageX - shiftX + 'px';

ball.style.top = pageY - shiftY + 'px';

}

function onMouseMove(event) {

moveAt(event.pageX, event.pageY);

}

document.addEventListener('mousemove', onMouseMove);

ball.onmouseup = function() {

document.removeEventListener('mousemove', onMouseMove);

ball.onmouseup = null;

};

};

ball.ondragstart = function() {

return false;

};

**[Цели переноса (droppable)](https://learn.javascript.ru/mouse-drag-and-drop" \l "tseli-perenosa-droppable)**

В предыдущих примерах мяч можно было бросить просто где угодно в пределах окна. В реальности пользователь обычно берёт один элемент и перетаскивает в другой. Например, файл в папку или что-то ещё. Т.е. он берёт перетаскиваемый (draggable) элемент и помещает его в другой элемент «цель переноса» (droppable).

Т.е. нужно знать, куда пользователь положил элемент в конце переноса и над какой потенциальной целью (например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

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). Таким образом, из любого обработчика событий мыши можно обнаружить потенциальную цель переноса вот так:

ball.hidden = true; // (\*)

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);

ball.hidden = false;

// elemBelow - элемент под мячом (цель переноса)

Обратите внимание, что нужно спрятать мяч перед вызовом функции (\*). Иначе получим координаты мяча, ведь это и есть элемент непосредственно под указателем: elemBelow=ball. Можно использовать этот код для проверки того, над каким элементом находится мяч, в любое время. И обработать окончание переноса, когда оно случится. Расширенный код onMouseMove с поиском потенциальных целей переноса:

let currentDroppable = null; // потенциальная цель переноса, над которой мяч сейчас находится

function onMouseMove(event) {

moveAt(event.pageX, event.pageY);

ball.hidden = true;

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);

ball.hidden = false;

if (!elemBelow) return; // (1)

let droppableBelow = elemBelow.closest('.droppable'); // (2)

if (currentDroppable != droppableBelow) { // (3)

if (currentDroppable) {

leaveDroppable(currentDroppable); // (4)

}

currentDroppable = droppableBelow;

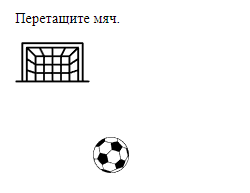
if (currentDroppable) {

enterDroppable(currentDroppable);

}

}

}

В приведённом примере, когда мяч перетаскивается через футбольные ворота, ворота подсвечиваются. Событие mousemove может произойти, когда курсор находится за пределами окна, тогда elementfromPoint вернёт null (1). Потенциальные цели переноса помечены классом droppable (2). Надо проверить перемещался элемент или нет, тогда он либо располагается над целью, либо нет (3). При этом и currentDroppable, и droppableBelowl могут быть равны null. currentDroppable=null, если элемент располагался не над droppable до этого события (например, над пустым пространством), droppableBelow=null, если элемент располагается не над droppable именно сейчас, во время этого события. Если элемент перемещен из droppable, то удаляем подсветку (4).

Теперь в течение всего процесса в переменной currentDroppable хранится текущая потенциальная цель переноса, над которой сейчас располагается мяч, можно её подсветить или сделать что-то ещё.

1. Прокрутка: событие scroll.

События прокрутки позволяют реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые можно сделать при этом.

Например:

* Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
* Загрузить больше данных, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

window.addEventListener('scroll', function() {

document.getElementById('showScroll').innerHTML = pageYOffset + 'px';

});

В действии:

Текущая прокрутка = 563.3333740234375px

Событие scroll работает как на window, так и на прокручиваемых элементах.

[Предотвращение прокрутки](https://learn.javascript.ru/onscroll" \l "predotvraschenie-prokrutki)

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 wheel – прокрутка колеса мыши («прокручивание» тачпада также его генерирует).
* На событии keydown для клавиш pageUp и pageDown.

Если поставить на них обработчики, в которых вызвать event.preventDefault(), то прокрутка не начнётся.

Иногда это может помочь, но более надёжный способ – использовать CSS, чтобы сделать что-то непрокручиваемым, например, свойство overflow.

Вот несколько задач, которые вы можете решить или просмотреть, чтобы увидеть применение onscroll.

1. Клавиатура: keyup, keydown.

События клавиатуры должны использоваться, если необходимо обрабатывать взаимодействие пользователя с клавиатурой (в том числе виртуальной). К примеру, если нужно реагировать на стрелочные клавиши Up и Down или горячие клавиши (включая комбинации клавиш).

Событие keydown происходит при нажатии клавиши, а keyup – при отпускании.

У объекта события есть свойство key, чьё значение – символ, и свойство code – «физический код клавиши». Например, 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** | **event.key** | **event.code** |
| --- | --- | --- |
| Z | z (нижний регистр) | KeyZ |
| Shift+Z | Z (Верхний регистр) | KeyZ |

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".

У каждой клавиши есть код, который зависит от её расположения на клавиатуре. Подробно о клавишных кодах можно прочитать в [спецификации о кодах событий UI](https://www.w3.org/TR/uievents-code/). 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 клавишные коды к каждой из них.

Регистр важен: правильно KeyZ, а не keyZ. Условие event.code=="keyZ" работать не будет: первая буква в слове "Key" должна быть заглавная.

Если клавиша не буквенно-цифровая, например, Shift или F1, или какая-либо другая специальная клавиша, то значение свойства event.key примерно тоже, что и у event.code:

| **Клавиша** | **event.key** | **event.code** |
| --- | --- | --- |
| F1 | F1 | F1 |
| Backspace | Backspace | Backspace |
| Shift | Shift | ShiftRight или ShiftLeft |

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 том, что вообще было нажато (Shift).

Допустим, надо обработать горячую клавишу Ctrl+Z (или Cmd+Z для Mac). Можно поставить обработчик событий на keydown и проверять, какая клавиша была нажата. Возникает вопрос: надо проверять значение event.key или event.code?

С одной стороны, значение event.key 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:

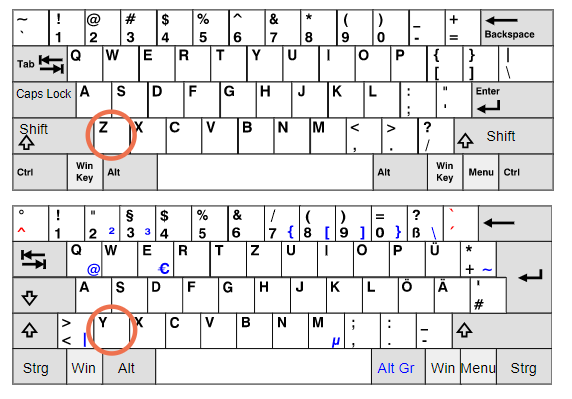
document.addEventListener('keydown', function(event) {

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 (буквы). Например, вот схема стандартной (US) раскладки («QWERTY») и под ней немецкой («QWERTZ») раскладки (предоставлены Википедией):

Для одной и той же клавиши в американской раскладке значение event.code равно «Z», в то время как в немецкой «Y». Таким образом, для пользователей с немецкой раскладкой event.code при нажатии на «Y» будет равен KeyZ.

В [спецификации](https://www.w3.org/TR/uievents-code/#table-key-code-alphanumeric-writing-system) напрямую упоминается такое поведение.

* Преимущество event.code заключается в том, что его значение всегда остается 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.
* event.code может содержать неправильный символ на нестандартной раскладке. Одни и те же буквы на разных раскладках могут сопоставляться с разными физическими клавишами, что приводит к разным кодам. Это происходит не со всеми кодами, а с несколькими, например, keyA, keyQ, keyZ, и не происходит со специальными клавишами, такими как Shift. Полный список проблемных кодов можно найти в [спецификации](https://www.w3.org/TR/uievents-code/#table-key-code-alphanumeric-writing-system).

Таким образом, чтобы отслеживать символы, зависящие от раскладки, event.key надёжнее.

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 весьма разнообразны, много чего можно инициировать нажатием на клавиатуре: появление символа, удаление символа, прокрутка страницы, открытие диалогового окна браузера «Сохранить» 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, как бы эта комбинация не обрабатывалась 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 номер телефона" type="tel">





Заметьте, что специальные клавиши по типу Backspace, Left, Right, Ctrl+V не работают как должны в поле для ввода. Это побочный эффект жесткого фильтра checkPhoneKey. Доработаем код:

<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 номер телефона" type="tel">





1. Загрузка документа.

У жизненного цикла HTML-страницы есть три важных события:

* DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> 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. Можно проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти;
* unload – пользователь почти ушёл, но всё ещё можно запустить некоторые операции, например, отправить статистику.

Рассмотрим эти события подробнее.

[**DOMContentLoaded**](https://learn.javascript.ru/onload-ondomcontentloaded#domcontentloaded)

Событие DOMContentLoaded срабатывает на объекте document. Необходимо 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), поэтому размер будет 0x0

alert(`Размер изображения: ${img.offsetWidth}x${img.offsetHeight}`);

}

document.addEventListener("DOMContentLoaded", ready);

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

В этом примере обработчик DOMContentLoaded запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже <img>. 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.

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"></script>

<script>

alert("Библиотека загружена, встроенный скрипт выполнен");

</script>

В примере выше сначала будет выведено «Библиотека загружена…», а затем «DOM готов!» (все скрипты выполнены).

Есть два исключения из этого правила:

1. Скрипты с атрибутом async (будет рассмотрен [немного позже](https://learn.javascript.ru/script-async-defer)) не блокируют DOMContentLoaded.
2. Скрипты, сгенерированные динамически при помощи document.createElement('script') и затем добавленные на страницу, также не блокируют это событие.

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт. Но если после стилей 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.

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**](https://learn.javascript.ru/onload-ondomcontentloaded#window-onload)

Событие load на объекте window наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы. В примере ниже правильно показаны размеры картинки, потому что window.onload дожидается всех изображений:

<script>

window.onload = function() {

alert('Страница загружена');

// к этому моменту страница загружена

alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);

};

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

[**window.onunload**](https://learn.javascript.ru/onload-ondomcontentloaded#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, собраны данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее. Естественно, событие unload – это тот момент, когда пользователь покидает страницу и надо 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.
* Можно послать не только строку, но так же формы и другие форматы, 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.  Если надо отменить переход на другую страницу, то здесь этого сделать нельзя. Но сможем в другом месте – в событии onbeforeunload.

**[window.onbeforeunload](https://learn.javascript.ru/onload-ondomcontentloaded" \l "window.onbeforeunload)**

Если посетитель собирается уйти со страницы или закрыть окно, обработчик beforeunload попросит дополнительное подтверждение. Если отменить это событие, то браузер спросит посетителя, уверен ли он. Запустите следующий код и затем перезагрузите страницу:

window.onbeforeunload = function() {

return false;

};

**[readyState](https://learn.javascript.ru/onload-ondomcontentloaded" \l "readystate)**

Если установить обработчик DOMContentLoaded после того, как документ загрузился, то он никогда не запустится.

Бывают ситуации, когда неизвестно, готов документ или нет. Но надо, чтобы функция исполнилась, когда DOM загрузился, неважно сейчас или позже. Свойство document.readyState показывает 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 можно 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 можно 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.

Рассмотрим весь поток событий. Здесь документ с <iframe>, <img> 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>

<img src="http://en.js.cx/clipart/train.gif" id="img">

<script>

img.onload = () => log('img onload');

</script>

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 (± 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. 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.

1. Загрузка скриптов

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:

1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
2. 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"></script>

<!-- Это не отобразится, пока скрипт не загрузится -->

<p>...содержимое после скрипта...</p>

Конечно, есть пути, как это обойти. Например, можно 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 у этого решения есть недостатки. 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. Эту проблему можно решить с помощью атрибутов тега <script>: defer и async.

[**defer**](https://learn.javascript.ru/script-async-defer#defer)

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится. 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"></script>

<!-- 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 скрипта с 'defer'!")); // (2)

</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...содержимое после скрипта...</p>

Содержимое страницы отобразится мгновенно. 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>

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 загрузится первым. Но спецификация требует последовательного выполнения скриптов согласно порядку в документе, поэтому он подождёт выполнения long.js.

Атрибут defer предназначен только для внешних скриптов. Поэтому он будет проигнорирован, если в теге <script> нет src.

[**async**](https://learn.javascript.ru/script-async-defer#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 casync не ждут другие скрипты.

Так что, если 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.

Можно 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).

Можно 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, в примере ниже добавляются 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);

}

loadScript("/article/script-async-defer/long.js");

loadScript("/article/script-async-defer/small.js");

1. Загрузка ресурсов.

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ифреймов, изображений и др. Для этого существуют два события:

* load – успешная загрузка,
* error – во время загрузки произошла ошибка.

[**Загрузка скриптов**](https://learn.javascript.ru/onload-onerror#zagruzka-skriptov)

Допустим, нужно загрузить сторонний скрипт и вызвать функцию, которая объявлена в этом скрипте. Можно загрузить этот скрипт динамически:

let script = document.createElement('script');

script.src = "my.js";

document.head.append(script);

Но чтобы вызвать функцию, которая объявлена внутри того скрипта, нужно подождать, пока скрипт загрузится, и только потом можно её вызвать.

Для наших собственных скриптов можно использовать [JavaScript-модули](https://learn.javascript.ru/modules), но они не слишком широко распространены в сторонних библиотеках.

[**script.onload**](https://learn.javascript.ru/onload-onerror#script-onload)

Событие load срабатывает после того, как скрипт был загружен и выполнен. Например:

let script = document.createElement('script');

// можно 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 создается вспомогательная функция с именем "\_"

alert(\_); // функция доступна

};

Таким образом, в обработчике onload можно использовать переменные, вызывать функции и т.д., которые предоставляет сторонний скрипт.

[**script.onerror**](https://learn.javascript.ru/onload-onerror#script-onerror)

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события error. Например, 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("Error loading " + this.src); // Ошибка загрузки https://example.com/404.js

};

Обратите внимание, что нельзя получить описание HTTP-ошибки. Неизвестно, была ли это ошибка 404 или 500, или какая-то другая. Известно только, что во время загрузки произошла ошибка.

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.

События load и error также срабатывают и для других 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 <img>. Изображения начинают загружаться, когда получают src (\*).
* Для <iframe> событие load срабатывает по окончании загрузки как в случае успеха, так и в случае ошибки.

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 c 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 используется скрипт с другого домена, и в нем имеется ошибка, то нельзя узнать детали этой ошибки.

Для примера рассматрим 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 располагается:

<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>

Видим отчёт об ошибке:

Uncaught ReferenceError: noSuchFunction is not defined

https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1

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.

Детали ошибки могут быть полезны. Существует много сервисов, которые обрабатывают глобальные ошибки при помощи window.onerror, сохраняют отчёт о них и предоставляют доступ к этому отчёту для анализа. Это здорово, потому что можно увидеть реальные ошибки, которые случились у наших пользователей. Но если скрипт – с другого домена, то информации об ошибках в нём почти нет.

Чтобы разрешить кросс-доменный доступ, нужно установить тегу <script> атрибут crossorigin, и, кроме того, удалённый сервер должен указать 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.

В рассматриваемом примере атрибут crossorigin отсутствовал. Поэтому кросс-доменный доступ был запрещён. Добавим его. Можно выбрать "anonymous" (куки не отправляются, требуется один серверный заголовок) или "use-credentials" (куки отправляются, требуются два серверных заголовка) в качестве значения атрибута.

<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-onerror/crossorigin/error.js"></script>

Теперь при условии, что сервер предоставил заголовок Access-Control-Allow-Origin, получен полный отчёт по ошибкам.