Взаимодействие пользователя с интерфейсом сводится к **созданию событий** и их **обработке по алгоритмам**.

Пользователь нажал на кнопку в форме — ожидает, что произойдёт какое-то действие. Пользователь прокрутил колесико мыши в определённом месте страницы — ждёт, что звук станет громче. Нажимает клавиши-стрелки на клавиатуре — предмет перемещается по экрану.

Каждое из событий, инициированных пользователем (нажатие кнопки на форме, прокрутка колёсика, нажатие клавиш) обрабатывается браузером, и пользователь ожидает определённое действие со стороны программного кода.

**Событие** — это сигнал от браузера о том, что что-то произошло.

События происходят в большом количестве на странице. Не все (далеко не все) из них инициированы пользователем. Некоторые происходят по времени, некоторые — в результате взаимодействия фронтенд-приложения с бекенд, какие-то события наступают при окончании других событий и т. д.

Наиболее популярные события:

* + мышь:

*click* — происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании),

*mousemove* — при движении мыши,

*contextmenu* — происходит, когда кликнули на элемент правой кнопкой мыши,

*mousedown / mouseup* — когда нажали / отжали кнопку мыши на элементе,

*mouseover / mouseout* — когда мышь наводится на / покидает элемент,

* + элементы управления:

*submit* — пользователь отправил форму <form>,

*input* — пользователь изменил текст в <input>,

* + клавиатура:

*keydown и keyup* — когда пользователь нажимает / отпускает клавишу,

* + документ:

*DOMContentLoaded* — когда *HTML* загружен и обработан,

* + *CSS:*

*transitionend* — когда *CSS*-переход завершён.

**Обработка события**

При происшествии события происходит его обработка. Любому событию можно назначить *обработчик*.

**Обработчик** — это функция, которая сработает, как только событие произошло.

Именно благодаря обработчикам *JavaScript*-код может реагировать на действия пользователя. Обработчики устанавливаются (назначаются) разработчиками при создании приложения. Далее рассмотрим варианты назначения обработчика.

Для тестирования возьмём следующую задачу. Дана страница *index.html*:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Тест консоли</title>

</head>

<body>

<h2>Тест консоли</h2>

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

</body>

</html>

Требуется при клике мышью на блок <h2>Тест консоли</h2> выводить в консоли сообщение «Это тот блок».

**Назначение обработчика через использование атрибута**

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется *on<событие>*.

Для выполнения поставленной задачи с помощью этого метода требуется изменить разметку:

<h2 onclick="console.log('Это тот блок')">Тест консоли</h2>

При клике мышкой на кнопку выполнится код, указанный в атрибуте onclick. Для удобства может быть создана отдельная функция и вызвана в атрибуте.

**Назначение обработчика через свойство**

Можно назначить обработчик события через использование свойства объекта.

Для выполнения поставленной задачи с помощью этого метода требуется изменить разметку следующим образом:

<h2 id="testTextField">Тест консоли</h2>

*Id* нам потребуется, чтобы найти элемент на странице (да, речь об одном из методов, оговорённых ранее). Пропишем в *script.js* следующий код:

**const** testTextField = document.querySelector('#testTextField');

testTextField.onclick = **function**() {

// тело обработчика

console.log('Это тот блок');

// конец тела обработчика

}

Если не создавать переменную для хранения элемента, этот же код можно записать следующим образом:

document.querySelector('#testTextField').onclick = **function**() {

// тело обработчика

console.log('Это тот блок');

// конец тела обработчика

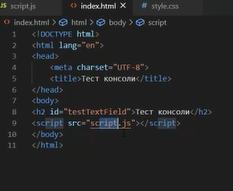
}

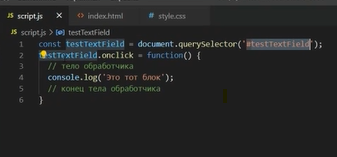
Удалить такой обработчик можно с помощью кода:

document.querySelector('#testTextField').onclick = **null**;

По сути, этот вариант назначения обработчика является более широким вариантом назначения обработчика через атрибут — обработчик записывается в то же свойство, только более «приличной» версией — не захламляется разметка, *JavaScript*-код находится там, где ему и положено — в *js*-файле.

У этого метода (как и у предыдущего) существует один серьёзный недостаток — нельзя назначить более одного обработчика на событие таким образом. Например, текст в поле ввода проверяется на валидность через один обработчик при событии *input*, а через другой обработчик — отправляется запрос на сервер. В этом случае, когда назначается на поле ввода второй обработчик, первый затирается.





## Назначение обработчика через addEventListener

Наиболее современный способ назначения обработчика — назначение через метод addEventListener.

element.addEventListener(event, handler);

// element - элемент, событие которого отслеживается

// event - событие

// handler - обработчик

Как и в предыдущем варианте, для выполнения поставленной задачи с помощью этого метода требуется сначала найти объект на странице. Используем для этого id:

<h2 id="testTextField">Тест консоли</h2>

Этот способ назначения обработчика наиболее предпочтителен, так как он лишён основного их недостатка — addEventListener() позволяет**назначить более одного обработчика на событие:**

**const** testTextField = document.querySelector('#testTextField');

testTextField.addEventListener('click',

(event) => {

// тело обработчика

console.log('Это тот блок');

// конец тела обработчика

})

Если не создавать переменную для хранения элемента, этот же код можно записать следующим образом:

document.querySelector('#testTextField').addEventListener('click',

(event) => {

console.log('Это тот блок');

})

## Доступ к элементу через this

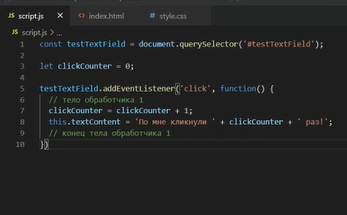
Внутри метода-обработчика можно получить доступ к элементу, на который назначен обработчик. Делается это через псевдоэлемент this:

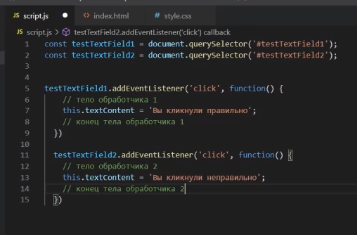
document.querySelector('#testTextField').addEventListener('click',

(event) => {

**this**.textContent = 'По мне кликнули!';

})





**Объект события**

При обработке события могут понадобиться детали этого события. Например, координаты указателя мыши или какая именно нажата клавиша. Для определения этих сведений используется *объект события*.

**Объект события** — это объект, внутри которого находится различная информация о событии.

Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику. То есть объект события приходит первым аргументом в функцию-обработчик.

В объекте события могут быть различные данные:

* + *event.type* — тип события (например, ‘*click*’);
  + *event.target* — элемент, на котором было вызвано событие;
  + *event.target.value* — текущее значение поля ввода, на которое добавлен обработчик события (только для *input*);
  + *event.currentTarget* — элемент, на котором сработал обработчик;
  + *event.clientX / event.clientY* — координаты курсора в момент клика (относительно окна, для событий мыши).

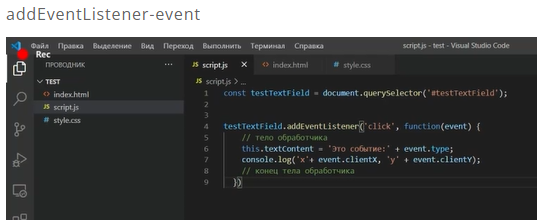
В зависимости от типа события и элемента, могут быть различные данные в *event*.

document.querySelector('#testTextField').addEventListener('click',

(event) => {

**this**.textContent = 'Это событие: ' + event.type;

})



## Предотвращение поведения по умолчанию

Нажатие на кнопку, имеющую type=submit, приведёт к отправке формы, но что если мы не хотим этого делать? Например, пользователь не ввёл какие-то данные, и мы хотим показать ему alert, проинформировать об этом, а форму не отправлять? Для этого у события в браузере есть специальный метод — preventDefault().

Пример использования preventDefault() для сброса события по умолчанию ссылки (тег <a>):

<a href="testTextField" id="testTextField">Кликните по ссылке</a>

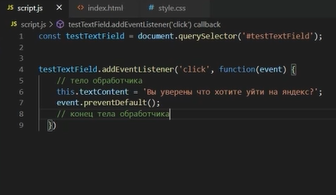
document.querySelector('#testTextField').addEventListener('click',

(event) => {

**this**.preventDefault();

alert('Вы кликнули по ссылке и ничего не произошло');

})



## Всплытие

При наступлении события обработчики сначала срабатывают на самом элементе, затем на его родителе, затем на родителе родителя и так далее, вверх по цепочке вложенности:

<div id="parent">

<p id="child">

Если кликнуть этот текст, то произойдёт три события

</p>

<p>

Если кликнуть этот текст, то произойдет одно событие

</p>

</div>

document.querySelector('body').addEventListener('click',

(event) => {

alert('Событие body');

})

document.querySelector('#parent').addEventListener('click',

(event) => {

alert('Событие родителя');

})

document.querySelector('#child').addEventListener('click',

(event) => {

alert('Событие потомка');

})

В приведённом примере видно, как событие click поднимается из элемента #child в элемент #parent, а из него — в body. Этот эффект называется **всплытием**. Если изобразить вложенность элементов следующим образом:

**Всплывают не все события.** Например, не всплывает событие focus.

Если по каким-то причинам требуется остановить всплытие, то используется метод события stopPropagation():

<div id="parent">

<p id="child">

Если кликнуть этот текст, то произойдёт три события

</p>

<p>

Если кликнуть этот текст, то произойдет одно событие

</p>

</div>

document.querySelector('body').addEventListener('click',

(event) => {

alert('Событие body');

})

document.querySelector('#parent').addEventListener('click',

(event) => {

alert('Событие родителя');

})

document.querySelector('#child').addEventListener('click',

(event) => {

alert('Событие потомка');

event.stopPropagation();

})

Если на элементе несколько обработчиков, иногда требуется не только перехватить всплытие, но и **прекратить работу остальных обработчиков** для этого элемента. Для этого используется метод stopImmediatePropagation().

Всплытие — это удобно. Не прекращайте его без причины. Причина при этом должна быть оправданной и очевидной.

Прекращение всплытия может создавать неожиданные проблемы, которые потом приходится обходить. Перед всплытием происходит «погружение», но этот вопрос советуем к самостоятельному изучению в будущем.