

# ОБЪЕКТ СОБЫТИЯ



**АРТЕМ ШАШКОВ / DOCDOC.RU**



# АРТЕМ ШАШКОВ

Front End Developer DocDoc.ru



[glomix@inbox.ru](mailto:glomix@inbox.ru)



[Артем Шашков](#)

# ПЛАН ЗАНЯТИЯ

1. Проблема свойства `.onclick`
2. `addEventListener`
3. `removeEventListener`
4. Объект события
5. Свойства `Event`
6. События клавиатуры
7. Модифицирующие клавиши



# ПРОБЛЕМА СВОЙСТВА `.onclick`

# ПЕРЕКЛЮЧАТЕЛЬ ВКЛАДОК

Предположим, что у нас есть такой HTML-код и мы хотим, чтобы кнопки, вложенные в тег `<nav>`, переключали вкладки:

```
1 <button>Кнопка</button>
2 <nav id="mainmenu">
3   <button class="current">Пункт 1</button>
4   <button>Пункт 2</button>
5   <button>Пункт 3</button>
6 </nav>
```

## ФУНКЦИЯ ПЕРЕКЛЮЧЕНИЯ ВКЛАДOK

```
1 function selectButton() {  
2     if (this.classList.contains('current')) {  
3         return;  
4     }  
5  
6     const currentButtons = document  
7         .getElementsByClassName('current');  
8     for (const button of currentButtons) {  
9         button.classList.remove('current');  
10    }  
11  
12    this.classList.add('current');  
13    // переключаем вкладки  
14 }
```

# ПОДКЛЮЧАЕМ ОБРАБОТЧИК КЛИКА НА КНОПКАХ

```
1  const menu = document.getElementById('mainmenu');
2  const menuButtons = menu.getElementsByTagName('button');
3  for (const button of menuButtons) {
4      button.onclick = selectButton;
5  }
```

[Весь код](#)

## ЖУРНАЛИРУЕМ КЛИКИ

Потом у нас появляется задача журналировать каждое нажатие кнопки на странице, например, отправляя информацию в Google Analytics. Пишем такой код:

```
1 function logEvent() {  
2     console.log('Кнопка нажата');  
3     // генерируем событие Google Analytics  
4 }  
5 const allButtons = document  
6     .getElementsByTagName('button');  
7 for (const button of allButtons) {  
8     button.onclick = logEvent;  
9 }
```

[Весь код](#)



# ПРОБЛЕМА С ПЕРЕКЛЮЧЕНИЕМ

Нажатия всех кнопок журналируются. А вот переключалка работать перестала. Почему?

Проблема свойств событий в том, что это просто свойства. Они хранят только одно значение. Поэтому мы можем назначить только один обработчик. Повторное назначение обработчика просто перетрет предыдущий обработчик. Есть, конечно, способы обойти эту проблему. Но зачем, если есть способ лучше. События на элемент можно назначать с помощью метода элемента `.addEventListener`.



**addEventListener**

# ИСТОРИЧЕСКАЯ СПРАВКА

Метод `addEventListener` был добавлен у элементов в 2000 году в рамках спецификации **DOM Level 2 Events** как замена старому способу добавления обработчиков через свойства `on*`.

Позволяет добавлять несколько обработчиков события и задавать дополнительные параметры обработки.

# СИНТАКСИС

```
EventTarget.addEventListener(type, listener)
```

Где:

- `EventTarget` – элемент, события на котором мы хотим обработать;
- `type` – тип события, *строка*;
- `listener` – обработчик события, *функция*.

В дальнейшем использование свойств `on*` для добавления обработчика событий будет считаться ошибкой.

# НАВЕСИМ ОБРАБОТЧИК ПРАВИЛЬНО

Нам потребуется тип события `click`. Не путайте с атрибутом или свойством узла `onclick`.

```
1  const menu = document.getElementById('mainmenu');
2  const menuButtons = menu.getElementsByTagName('button');
3  for (const button of menuButtons) {
4      button.addEventListener('click', selectButton);
5  }
```

[Весь код](#)

# ИСПОЛЬЗУЙТЕ ЕДИНЫЙ СТИЛЬ

Код уже работает потому, что `addEventListener` можно совместно использовать с `onclick`.

Но в целом способ навешивания обработчика через свойство устарел, и плюс, нужно придерживаться единого подхода. Поэтому перепишем и второй блок

```
1  const allButtons = document
2    .getElementsByTagName('button');
3  for (const button of allButtons) {
4    button.addEventListener('click', logEvent);
5  }
```

[Весь код](#)

---

## ПОХОЖАЯ РАБОТА

В остальном работа обработчиков ничем не отличается от того, что мы узнали на первых лекциях. Функции обработчиков также вызываются в контексте элемента, на котором был назначен обработчик. Как видите, нам даже не потребовалось ничего менять в функциях `selectButton` и `logEvent`.

# ОБРАБОТЧИКИ ВЫЗЫВАЮТСЯ В ПОРЯДКЕ НАЗНАЧЕНИЯ

Стоит разобраться, в каком порядке будут вызываться обработчики, если их будет несколько:

```
1 <button id="testOrder">Проверить порядок вызова</button>
2 <script>
3   const button = document.getElementById('testOrder');
4   button.addEventListener('click', () =>
5     console.log('Первый'));
6   button.addEventListener('click', () =>
7     console.log('Второй'));
8 </script>
```



## ПЕРВЫМ ВЫЗЫВАЕТСЯ ОБРАБОТЧИК, КОТОРЫЙ БЫЛ НАЗНАЧЕН РАНЬШЕ

Если мы нажмем на кнопку «Проверить порядок вызова», то в консоли мы увидим:

Первый  
Второй

Что доказывает правило: обработчики вызываются в том порядке, в котором они были добавлены. Осталось выяснить, как удалить обработчик события, если он оказался больше не нужен.



**removeEventListener**

## КАК УДАЛИТЬ ОБРАБОТЧИК?

Мы помним, что свойству `onclick` мы могли просто присвоить значение `null`, и обработчик события больше не выполнялся при наступлении события.

Как же удалять обработчики событий, которые мы добавили с помощью `addEventListener`?

Для этого есть метод `removeEventListener`.

## «ИГРАЕМ» С ОБРАБОТЧИКОМ

Напишем следующую HTML-разметку:

```
1 <button id="action">Событие</button>
2 <button id="addListener">Добавить обработчик</button>
3 <button id="removeListener">Удалить обработчик</button>
```

# СКРИПТ ОБРАБОТКИ

Добавим немного JS для обработчика и кнопок:

```
1  let counter = 0;
2  function handleEvent() {
3      counter++;
4      console.log(`Кнопка нажата ${counter} раз!`);
5  }
6
7  const actionButton = document.getElementById('action');
8  const addButton = document.getElementById('addListener');
9  const removeButton = document.
10 getElementById('removeListener');
```

# СНИМАЕМ ОБРАБОТЧИК

Добавляем и снимаем обработчик:

```
1 addButton.addEventListener('click', () => {  
2   actionButton.addEventListener('click', handleEvent);  
3 });  
4  
5 removeButton.addEventListener('click', () => {  
6   actionButton.removeEventListener('click', handleEvent);  
7 });
```

Обратите внимание! Для снятия нам нужно вызвать метод у того же самого элемента, снимать то же самое событие, передав ту же самую функцию обработки.

[Весь код](#)

# ВАЖНЫЕ ВЫВОДЫ

Какие выводы важно сделать из этого примера:

1. Не приводит к ошибке удаление обработчика, которого нет.
2. Игнорируется повторное добавление того же самого обработчика на то же самое событие.
3. Если обработчик добавлен функциональным выражением, то мы не сможем снять его. Например обработчик, добавленный на `addButton`

## ПОДДЕРЖКА В IE МЛАДШЕ 9 ВЕРСИИ

В Internet Explorer младше 9 версии нет метода `.addEventListener`, вместо него используется `.attachEvent(type, listener)`. Ключевое отличие — `listener` вызывается в контексте `window`.

Если вам требуется сделать реализацию, которая будет поддерживать старые версии IE, то код будет выглядеть примерно так:

```
1 function handleEvent() {};  
2 const tag = document.getElementById('tagId');  
3 if (tag.addEventListener) {  
4     tag.addEventListener('click', handleEvent);  
5 } else if (tag.attachEvent) {  
6     tag.attachEvent('onclick', handleEvent);  
7 }
```





# ОБЪЕКТ СОБЫТИЯ

# КЛИК ПО ССЫЛКЕ

В отличие от кликов на других элементах, клик на гиперссылку приводит к переходу на другую страницу. Сработает ли при этом событие клика?

Давайте выведем в консоль адрес гиперссылки, на которую мы кликнули.

```
1 <nav>
2   <a href="http://netology.ru/">Нетология</a>
3   <a href="https://foxford.ru/">Фоксфорд</a>
4 </nav>
```

## ВЫВОДИМ URL В КОНСОЛЬ

```
1 function showHref() {  
2   console.log(this.href);  
3 }  
4  
5 const links = document.getElementsByTagName('a');  
6 Array.from(links).forEach(link => {  
7   link.addEventListener('click', showHref);  
8 });
```

[Live Demo](#)

# СЧИТАЕМ ВРЕМЯ

Из примера видно, что адрес ссылки успеваает показаться перед переходом на страницу. А что, если нам потребуется сделать какое-то более длительное действие? Поправим нашу функцию `showHref` так:

```
1 function showHref() {  
2   console.log(this.href);  
3   let counter = 0;  
4   setInterval(() => console.log(++counter), 10);  
5 }
```

[Live Demo](#)

## РЕЗУЛЬТАТ РАБОТЫ

Видно, что пока браузер «думает», наш счетчик успевает досчитать до 20-30. Т.е., думает он где-то 200-300 миллисекунд. Какой-то синхронный процесс может и вовсе отложить переход на всё время его выполнения. Например, навсегда:

```
1 function showHref() {  
2     console.log(this.href);  
3     while (true); // !!! НЕ ПОВТОРЯЙТЕ В РЕАЛЬНОМ ПРОЕКТЕ  
4 }
```

Это просто демонстрация того, как взаимосвязаны обработчик клика по ссылке и переход на соответствующую страницу. Из которого мы должны сделать вывод, что переход происходит только после выполнения функции обработчика события. А точнее, всех обработчиков.

# ОТМЕНЯЕМ ДЕЙСТВИЕ ПО УМОЛЧАНИЮ

Для того, чтобы отменить переход по ссылке, есть возможность, которую предоставляет нам сам браузер: отмена действия по умолчанию. Чтобы им воспользоваться, нужно немного узнать об объекте события.

Вернемся пока к примеру без использования ссылок:

```
1 <button id="mainButton">Нажми меня</button>
2 <script>
3   function showEvent(...args) {
4     console.log(this);
5     console.log(args);
6   }
7
8   const button = document.getElementById('mainButton');
9
```

# РЕЗУЛЬТАТ КЛИКА

Посмотрим вывод в консоль при клике:

```
<button id="mainButton">Нажим меня</button>  
[ MouseEvent ]
```

Функция `showEvent` вызвана в контексте кнопки, и в неё передан аргумент.

# ОБЪЕКТ СОБЫТИЯ


Это объект, содержащий подробную информацию о событии, которое мы обрабатываем. Он является экземпляром `Event`. Но так как это событие мыши, то он также является экземпляром класса `MouseEvent`, который дополняет информацию о событии деталями, которые актуальны для событий мыши.

```
1 function showEvent(event) {  
2     console.log(event instanceof Event);  
3     console.log(event instanceof MouseEvent);  
4     console.log(event);  
5 }
```

[Live Demo](#)

`MouseEvent` мы подробно разберем на следующих лекциях.





**СВОЙСТВА Event**

# ЗАПИСЫВАЕМ СОБЫТИЯ

Допустим мы повесили функцию, фиксирующую разные события и записывающую информацию о них в консоль:

```
1 function logEvent(event) {  
2     console.log('Событие наступило!');  
3 }
```

## type у СОБЫТИЯ

Теперь при наступлении разных событий информация о них выводится в консоль. Но какое событие наступило? Об этом можно узнать из объекта события, которое передается в обработчик. У него есть свойство `type`, в котором будет тип наступившего события, ровно такой, какой мы задаем при вызове `addEventListener`:

```
1 function logEvent(event) {  
2   console.log(`Событие ${event.type} наступило!`);  
3 }
```

# currentTarget

Другой вопрос: на каком элементе наступило это событие? Ответ на него очевиден, если мы повесили обработчик только на один элемент. А что, если мы повесили его на несколько разных элементов? Для того, чтобы понять, на каком из них произошло событие, у объекта `Event` есть свойство `currentTarget`, в котором будет ссылка на тот самый элемент, на который был назначен обработчик события:

```
1 function logEvent(event) {  
2   const tagName = event.currentTarget.tagName ?  
3   event.currentTarget.tagName : 'document';  
4   console.log(`Событие ${event.type} наступило на  
5   элементе ${tagName}!`);  
6 }
```

## ОСОБЕННОСТЬ `document`

Если событие было назначено на `document`, то `currentTarget` будет указывать на `document`, и у него нет свойства `tagName`.

## ЗАМЕНЯЕМ `this`

Так как свойство `currentTarget` всегда указывает на элемент, на котором был назначен обработчик, то его можно использовать вместо `this` и при передаче объекта `Event` в другие функции.

# ОТМЕНЯЕМ ПЕРЕХОД ПО ССЫЛКЕ

Но вернемся к нашей задаче. Как отменить переход по ссылке?

```
1 <nav>
2   <a href="http://netology.ru/">Нетология</a>
3   <a href="https://foxford.ru/">Фоксфорд</a>
4 </nav>
```

```
1 function showHref() {
2   console.log(this.href);
3 }
4
5 const links = document.getElementsByTagName('a');
6 Array.from(links).forEach(link => {
7   link.addEventListener('click', showHref)
8 });
```

# preventDefault

Для этого в объекте `Event` есть метод `preventDefault`, которой отменяет действие браузера на событие по умолчанию. Только действия самого браузера. Все обработчики, назначенные нами в скрипте, отработают. В частности, метод укажет браузеру не обрабатывать это событие клика по ссылке, т.е. не переходить по адресу, указанному в атрибуте `href`.



# ПРОВЕРЯЕМ

```
1 function showHref(event) {  
2     event.preventDefault();  
3     console.log(this.href);  
4 }
```

[Live Demo](#)

# ПЛЕЙ-ЛИСТ

Как это можно использовать? Например, мы можем реализовать плей-лист, организовав HTML-код так:

```
1 <audio id="myPlayer"></audio>
2 <nav>
3   <a href="./mp3/LA%20Chill%20Tour.mp3">
4     LA Chill Tour
5   </a>
6   <a href="./mp3/This%20is%20it%20band.mp3">
7     This is it band
8   </a>
9   <a href="./mp3/LA%20Fusion%20Jam.mp3">
10    LA Fusion Jam
11  </a>
12 </nav>
```

## ШАГ 1. ОБРАБОТЧИК

Добавляем обработчики события на ссылки:

```
1 function playSong() {}  
2  
3 const player = document.getElementById('myPlayer');  
4 const playlist = document.getElementsByTagName('a');  
5 for (const song of playlist) {  
6     song.addEventListener('click', playSong);  
7 }
```

## ШАГ 2. ПОЛУЧАЕМ `href`

Получаем `href` ссылки, добавляем её в `src` плеера и запускаем проигрывание:

```
1 function playSong() {  
2     player.src = this.href;  
3     player.play();  
4 }
```

[Live Demo](#)

## ПРОМЕЖУТОЧНЫЙ ИТОГ

Музыка начинает играть, но потом вместо нашей страницы открывается страница с файлом выбранной песни. Это потому что браузер после нашего обработчика выполнил действие по умолчанию — при клике на тег `a` открыть страницу, указанную в свойстве `href`. И теперь мы знаем, как сообщить ему этого не делать:

```
1 function playSong(event) {  
2     event.preventDefault();  
3     player.src = this.href;  
4     player.play();  
5 }
```

[Live Demo](#)

# preventDefault НЕ МЕШАЕТ

`preventDefault` не отменяет остальные обработчики. Несмотря на то, что внутри `playSong` мы вызвали `preventDefault` события, обработчик `logEvent` тоже выполнится:

```
1 function logEvent(event) {  
2   console.log(`Событие ${event.type} наступило на  
3   элементе ${event.currentTarget.tagName}!`);  
4 }  
5  
6 for (const song of playlist) {  
7   song.addEventListener('click', playSong);  
8   song.addEventListener('click', logEvent);  
9 }
```

[Live Demo](#)

## ПРИНИМАЕМ РЕШЕНИЕ

Можно прямо в обработчике принять решение о том, отменять действие браузера по умолчанию или нет. Например, мы можем проверить, что находится по ссылке. Если это музыкальный файл, то проигрываем его, в ином случае переходим по ссылке:

```
1 function playSong(event) {  
2     const isSong = /\. (mp3|wav|ogg)$/i  
3     if (isSong.test(this.href)) {  
4         event.preventDefault();  
5         player.src = this.href;  
6         player.play();  
7     }  
8 }
```

## УПРАВЛЯЕМ С КЛАВИАТУРЫ

А что, если мы хотим дополнительно реализовать управление нашим плеером с клавиатуры? Например, клавиша **P** будет запускать трек. Клавиша **S** — останавливать. Для этого нам для начала нужно познакомиться с событиями клавиатуры.





# СОБЫТИЯ КЛАВИАТУРЫ

# СОБЫТИЯ ОБРАБОТКИ

Для обработки событий клавиатуры доступно 3 события:

- `keypress` – нажата клавиша,
- `keydown` – клавиша вжата,
- `keyup` – клавиша отпущена.

# ПОРЯДОК СРАБАТЫВАНИЯ

При нажатии клавиши события срабатывают в таком порядке:

1. Первым срабатывает событие `keydown`.
2. Потом срабатывает событие `keypress`, не дожидаясь поднятия клавиши.
3. Событие `keyup` наступает только тогда, когда клавиша отпущена.

Нам пока достаточно будет события `keypress`.

# СТАВИМ ОБРАБОТЧИК

Обработчик события можно поставить на `document`:

```
1 function updatePlayer(event) {  
2     console.log(event);  
3 }  
4  
5 document.addEventListener('keydown', updatePlayer);
```

## ПОНИМАЕМ КЛАВИШИ

Как понять, какая клавиша была нажата? Для этого нам опять нужно обратиться к объекту события. На клавиатурные события создается объект `KeyboardEvent`:

```
1 function updatePlayer(event) {  
2   console.log(event instanceof KeyboardEvent);  
3 }
```

## СВОЙСТВА `KeyboardEvent`

У события `KeyboardEvent` есть два свойства, которые помогут решить нашу задачу `key` и `code`:

```
1 function updatePlayer(event) {  
2   console.log(event.key, event.code);  
3 }
```

## СВОЙСТВО `key`

Свойство `key` содержит символ, который набран на клавиатуре, в зависимости от языка и регистра. Т.е., это может быть `s`, `S`, `Ы`, `ы` и другие символы, если мы будем нажимать на клавишу `S` на клавиатуре.

## СВОЙСТВО `code`

Свойство `code` в этом случае всегда будет `KeyS`, что удобно, когда нам важен не набранный символ, а нажатая клавиша. Теперь мы можем поймать нужную клавишу и управлять плеером ([Live Demo](#)):

```
1 function updatePlayer(event) {  
2     switch (event.code) {  
3         case 'KeyS':  
4             player.pause();  
5             player.currentTime();  
6             break;  
7         case 'KeyP':  
8             player.play();  
9             break;  
10    }  
11 }
```



# ОБРАБАТЫВАЕМ МНОГО КЛАВИШ

Если клавиш планируется обработать много, то проще вынести связь клавиши и действия в отдельный объект:

```
1  const bindings = {  
2    KeyS() {  
3      player.pause();  
4      player.currentTime();  
5    },  
6    KeyP() {  
7      player.play();  
8    }  
9  }  
10 function updatePlayer(event) {  
11   if (event.code in bindings) {  
12     bindings[event.code]();  
13   }
```

# ОТЛИЧИЕ СОБЫТИЙ КЛАВИАТУРЫ

Событие `keypress` срабатывает только в том случае, если нажатая клавиша печатает какой-то символ. Поэтому в обработчике этого события не поймать нажатия клавиш стрелок и различных функциональных клавиш `Esc`, `Alt` и другие.

Вот простой пример, нажатие стрелок и функциональных клавиш не генерируют событие `keypress`, но генерируют `keydown`, для остальных символов срабатывают оба события:

```
1 function showKey(event) {  
2     console.log(event.type, event.key);  
3 }  
4  
5 document.addEventListener('keypress', showKey);  
6 document.addEventListener('keydown', showKey);
```

## ЗАЖАТАЯ КЛАВИША

Другое важное отличие в повторении. Если мы зажмем клавишу и будем удерживать её нажатой, то события `keydown` и `keypress` будут повторяться с определенным интервалом. А событие `keyup` нет.

```
1 function showKey(event) {  
2     console.log(event.type, event.code);  
3 }  
4  
5 document.addEventListener('keydown', showKey);  
6 document.addEventListener('keyup', showKey);
```

## СВОЙСТВО `repeat`

Если мы зажмем и подержим клавишу нажатой, то событие `keydown` сработает несколько раз, хотя мы нажали её только один раз, а событие `keyup` только однажды. Чтобы отличить в `keydown` и `keypress` повторные нажатия, можно проверить свойство `repeat`. Оно будет содержать `true` для событий сгенерированных автоматически повторно при удержании клавиши:

```
1 function showKey(event) {  
2   if (event.repeat) {  
3     console.log(`Повторное нажатие ${event.code}`);  
4   } else {  
5     console.log(`Нажата ${event.code}`);  
6   }  
7 }  
8  
9 document.addEventListener('keydown', showKey);
```



# МОДИФИЦИРУЮЩИЕ КЛАВИШИ

## ГОРЯЧИЕ КЛАВИШИ

Часто для реализации горячих клавиш нас интересует комбинации клавиши с клавишами-модификаторами. Например с `Alt`, `Ctrl`, `Shift`. Для того, чтобы выяснить, была ли клавиша нажата просто, или она была нажата с модификатором, в объекте `KeyboardEvent` есть свойства, соответствующие этим клавишам:

- `altKey` – модификатор `Alt`;
- `ctrlKey` – модификатор `Ctrl`;
- `shiftKey` – модификатор `Shift`;
- `metaKey` – модификатор, соответствующий кнопке `Windows` в операционной системе Windows и кнопке `Command` в OSX.

## ДОБАВИМ Ctrl

Поменяем запуск плеера на **Ctrl** + **P** и **Ctrl** + **S**. Проверим в самом начале, нажата ли клавиша **Ctrl**. Нет – выйдем из обработчика:

```
1 function updatePlayer(event) {  
2   if (!event.ctrlKey) {  
3     return;  
4   }  
5   switch (event.code) {  
6     case 'KeyS':  
7       player.pause();  
8       player.currentTime();  
9     break;  
10    case 'KeyP':  
11      player.play();  
12    break;  
13  }
```

## ЗАРЕЗЕРВИРОВАННЫЕ СОЧЕТАНИЯ

Обратите внимание, что в браузере уже используются различные комбинации клавиш, и не стоит использовать общепринятые комбинации.

Учитывайте, что на разных операционных системах комбинации клавиш организуются по разным принципам. И пример выше будет не только запускать плеер, но и выводить страницу на печать. Конечно, обойти печать страницы можно, вызвав `event.preventDefault` в обработчике, но так делать крайне не рекомендуется, лучше поменять комбинацию клавиш.



## Ctrl МЕНЯЕМ НА Alt

```
1 function updatePlayer(event) {  
2     if (!event.altKey) {  
3         return;  
4     }  
5     switch (event.code) {  
6         case 'KeyS':  
7             player.pause();  
8             player.currentTime();  
9             break;  
10        case 'KeyP':  
11            player.play();  
12            break;  
13        }  
14    }  
15 }
```

# ОБРАБОТКА СОБЫТИЙ

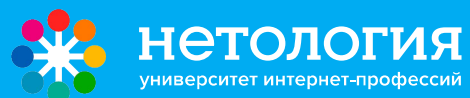
- Практика добавления обработчика события через свойство `on*` является устаревшей и несёт в себе серьёзные ограничения.
- Правильным является использование метода `addEventListener`.
- Обработчики вызываются в порядке их назначения.
- Обработчик события удаляется методом `removeEventListener`.
- Если обработчик добавлен функциональным выражением, то мы не сможем снять его.
- В Internet Explorer младше 9 версии нет метода `.addEventListener`, вместо него используется `.attachEvent`.

# СВОЙСТВА СОБЫТИЙ

- О том какое событие наступило можно узнать из объекта события, которое передается в обработчик.
- Для того, чтобы понять, на каком элементе произошло событие, у объекта `Event` есть свойство `currentTarget`.
- Метод `preventDefault` отменяет действие браузера на событие по умолчанию.
- `preventDefault` не отменяет остальные обработчики.

# СОБЫТИЯ КЛАВИАТУРЫ

- Для обработки событий клавиатуры доступно 3 события: `keypress`, `keydown` и `keyup`.
- Первым срабатывает `keydown`, последним `keyup`.
- На клавиатурные события создается объект `KeyboardEvent` со свойствами `key` и `code`.
- Свойство `key` объекта `KeyboardEvent` зависимо от регистра и языка.
- Свойство `code` используется когда нам важен не набранный символ, а нажатая клавиша
- Чтобы отличить в `keydown` и `keypress` повторные нажатия, можно проверить свойство `repeat`.
- Для отслеживания клавиш-модификаторов (Alt, Ctrl и т.д.) в объекте `KeyboardEvent` есть соответствующие свойства.



**Задавайте вопросы и напишите отзыв о лекции!**

**АРТЕМ ШАШКОВ**

 [glomix@inbox.ru](mailto:glomix@inbox.ru)

 [Артем Шашков](#)