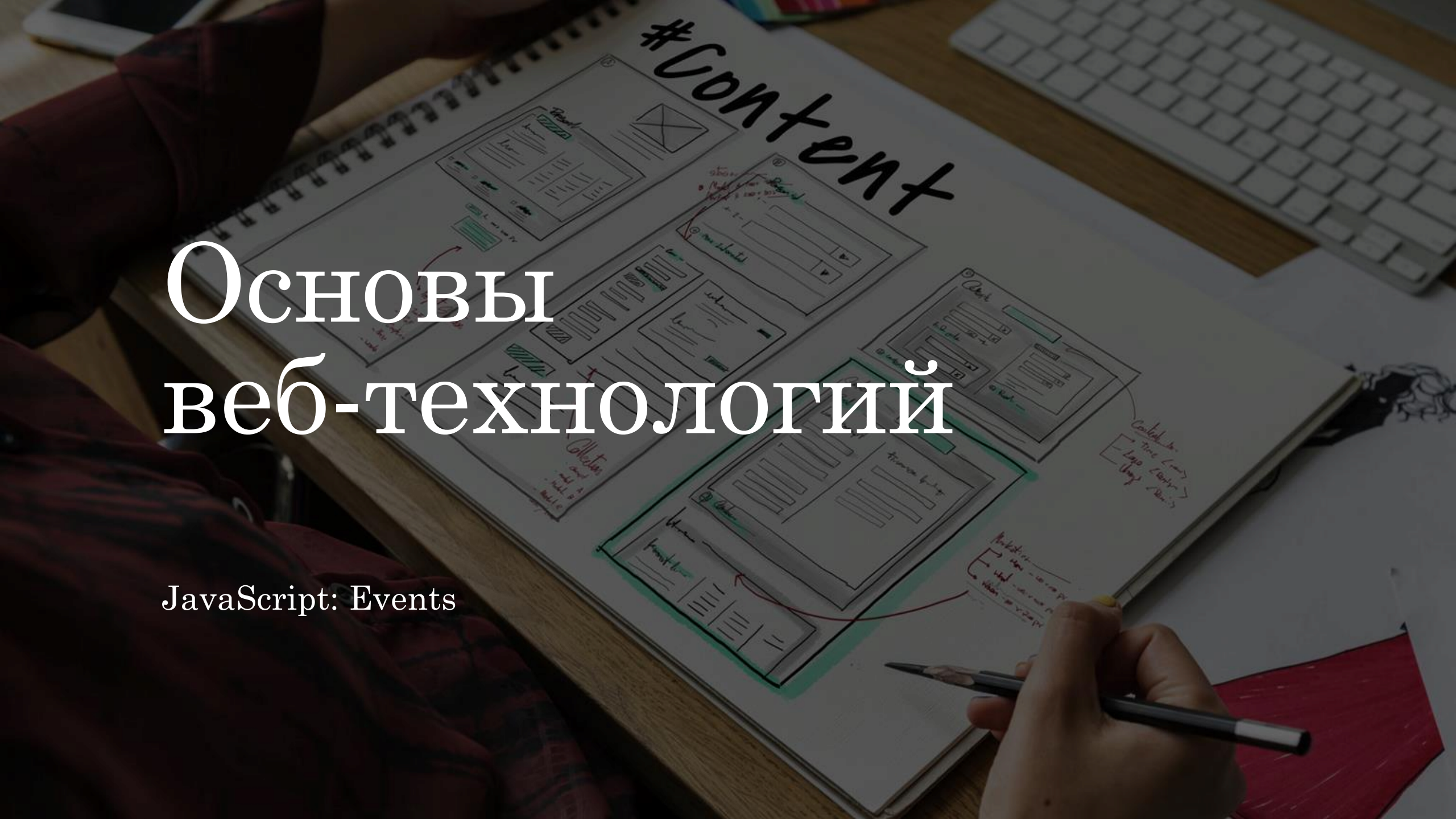


Основы веб-технологий

JavaScript: Events

Свойство HTMLElement.style

Свойство `HTMLElement.style` используется для получения и установки встроенных стилей. Возвращает объект `CSSStyleDeclaration`, который содержит список всех свойств, определенных только во **встроенном** стиле элемента (то есть определенных через аттрибут `style` элемента), а не весь CSS. Свойство можно как читать, так и записывать. При записи свойства записываются в camelCase (верблюжьей нотации), то есть `background-color` превращается в `element.style.backgroundColor` и т. д.

Для получения всех стилей элемента (наследуемых, заданных в css-файле и вычисленных в соответствии со специфичностью необходимо использовать метод `window.getComputedStyle(element, pseudoElt)`

```
const elem = document.querySelector(".active a");
console.log(elem.style.color);// получили информацию об встроенном свойстве color
elem.style.color = "green";  // изменили цвет
elem.style.fontSize = "40px"; //изменили шрифт

const computedStyle = window.getComputedStyle(elem);
console.log(computedStyle.lineHeight); // получили информацию об свойстве из css
console.log(parseInt(computedStyle.lineHeight)); // Преобразовали 50px в 50
```

Атрибуты

Элементам-узлам соответствуют HTML-теги у которых есть текстовые атрибуты. Доступ к атрибутам осуществляется при помощи стандартных методов. Эти методы работают со значением, которое находится в HTML.

- elem.hasAttribute(name) - проверяет наличие аттрибута, возвращает true или false
- elem.getAttribute(name) - получает значение атрибута и возвращает его
- elem.setAttribute(name, value) - устанавливает атрибут
- elem.removeAttribute(name) - удаляет атрибут
- elem.attributes – свойство, возвращает коллекцию всех атрибутов элемента

```
<img src="img.jpg" alt="Тут должна быть картинка" class="image">
```

Для картинки выше выполним скрипт:

```
const image = document.querySelector(".image");
console.log(image.attributes); // Объект со  всеми аттрибутами элемента
//NamedNodeMap {0: src, 1: class, 2: alt, src: src, class: class, alt: alt, length: 3}
console.log(image.hasAttribute("src")); // true
console.log(image.getAttribute("alt")); // Тут должна быть картинка
image.setAttribute("alt", "Это картинка!");
console.log(image.getAttribute("alt")); // Это картинка!
```

Свойства узлов-элементов

Во время построения DOM-дерева многие стандартные HTML-атрибуты становятся свойствами узлов. Посмотрим на несколько часто использующихся свойств.

`hidden` - контролирует видимость элемента.
`value` - содержит текущий текстовый контент элементов `input`, `select`, `textarea`.
`checked` - хранит состояние чекбокса или радиокнопки.
`name` - хранит значение, указанное в HTML-атрибуте `name`.
`src` - путь к изображению тега `<img>`.

Все свойства и методы узлов-элементов можно посмотреть здесь:
https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API в соответствующем элементу разделе (HTML**Link**Element, HTML**Image**Element, HTML**Input**Elementи др.)

```
const headH1 = document.getElementById("head");
headH1.hidden = true; // h1 стал невидимым

const message = document.querySelector("#message");
console.log(message.value); //  Текст поля textarea
console.log(message.name);  // Значение атрибута name

const link = document.querySelector(".active a");
console.log(link.href); // https://mephi.ru

const image = document.querySelector(".image");
image.src = '01.jpg';  // изменили src
image.alt = "Это африка!"  // изменили alt
```

Data-атрибуты

Дата-атрибут — это пользовательский атрибут на HTML-элементе, название которого начинается с `data-` . Это встроенная возможность стандарта HTML5. В дата-атрибутах можно хранить дополнительную информацию в стандартных HTML-элементах и работать с этой информацией в JavaScript при помощи собственного API.

Обращение к свойству `.dataset` вернёт объект со всеми дата-атрибутами, которые есть на элементе.

```
<div class="book" data-book-id="12">Описание книги </div>
<div class="book" data-book-id="13">Описание книги</div>
<div class="book" data-book-id="14">Описание книги</div>
<div class="book" data-book-id="15">Описание книги</div>

<script>
    const books = document.querySelectorAll(".book");
    books.forEach( book => {
        console.log(book.dataset.bookId)
    });
</script>
```

Браузерные события

Событие – это определённый сигнал от браузера. Он сообщает нам о том, что что-то произошло. Например: щелчок мыши, нажатие клавиши на клавиатуре, изменение размера области просмотра, завершение загрузки документа и т.д. При этом сигнал всегда связан с объектом. Подавать сигналы могут различные объекты: `window`, `document`, DOM-элементы и т.д.

Вот список самых часто используемых 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 (формы):
- `submit` – пользователь отправил форму `<form>`.
- `focus` – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:
- `keydown, keypress` и `keyup` – когда пользователь нажимает / отпускает клавишу.

События документа:
- `DOMContentLoaded` – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

Браузерные события

Тип события – это строка, определяющая тип действия, вызвавшего событие. Тип «`mousemove`», например, означает, что пользователь переместил указатель мыши. Тип «`keydown`» означает, что была нажата клавиша на клавиатуре. А тип «`load`» означает, что завершилась загрузка документа (или какого-то другого ресурса) из сети. Поскольку тип события - это просто строка, его иногда называют именем события.

Цель события – это объект, в котором возникло событие или с которым это событие связано. Когда говорят о событии, обычно упоминают тип и цель события. Например, событие «`load`» объекта `window` или событие «`click`» элемента `<button>`. Самыми типичными целями событий в клиентских приложениях на языке JavaScript являются объекты `Window`, `Document` и `Element`, но некоторые типы событий могут происходить и в других типах объектов.

Обработчик события – это функция, которая обрабатывает, или откликается на событие. Приложения должны зарегистрировать свои функции обработчиков событий в веб-браузере, указав тип события и цель. Когда в указанном целевом объекте возникнет событие указанного типа, браузер вызовет обработчик. Когда обработчики событий вызываются для какого-то объекта, мы иногда говорим, что браузер «возбудил» или «сгенерировал» событие.

Объект события – это объект, связанный с определенным событием и содержащий информацию об этом событии. Объекты событий передаются функции обработчика события в виде аргумента. Все объекты событий имеют свойство `type`, определяющее тип события, и свойство `target`, определяющее цель события.

Браузерные события

Назначить обработчик событию можно разными способами:

- через HTML-атрибут on{событие} (не является хорошей практикой);
- посредством свойства DOM-элемента on{событие};
- используя специальный метод addEventListener.

Инициализация обработчика через атрибут

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`.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

```
<h1 onclick="change()"> Кликни здесь! </h1>
<script>
        function change() {
            const body = document.body;
            body.classList.remove('one');
            body.classList.add('two');
        }
</script>
```

Инициализация обработчика через атрибут

```
<input type="text" id="input1">
<input type="text" id="input2">
<div id="result"></div>
<input type="button" value="Найти сумму" onclick="sum()">

<script>
    function sum() {
        const val1 = parseInt(document.querySelector("#input1").value);
        const val2 = parseInt(document.querySelector("#input2").value);
        const s = val1+val2;
        const divResult = document.querySelector("#result")
        divResult.textContent = s;
    }
</script>
```

Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

```
<input type="text" id="input1">
<input type="text" id="input2">
<div id="result"></div>
<input type="button" value="Найти сумму" id="btn-sum">

<script>
    function sum() {
      const val1 = parseInt(document.querySelector("#input1").value);
      const val2 = parseInt(document.querySelector("#input2").value);
      const s = val1+val2;
      const divResult = document.querySelector("#result")
      divResult.textContent = s;
    }

    const btn = document.querySelector("#btn-sum");
    btn.onclick = sum;   // Если указать sum(), то функция будет вызвана сразу!
</script>
```

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя (например через атрибут и свойство).

Доступ к элементу внутри обработчика

Внутри обработчика можно обратиться к текущему элементу, т.е. к тому для которого в данный момент был вызван этот обработчик. Осуществляется это с помощью ключевого слова this.

```
<div>
    <button> Кнопка 1 </button>
    <button> Кнопка 2 </button>
    <button> Кнопка 3 </button>
</div>
<script>
    function randColor(){
        return Math.round(Math.random()*255);
    }

    function btnChange() {
    this.style.backgroundColor = `rgb(${randColor()},${randColor()},${randColor()} )`;
    }

    const btns = document.querySelectorAll("button");
    btns.forEach((button) => button.onclick = btnChange);
</script>
```

Подписка на событие через addEventListener

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. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}.
  - passive: если true, то указывает, что обработчик никогда не вызовет preventDefault()

Для удаления обработчика следует использовать removeEventListener:

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

```
btns.forEach((button) => button.addEventListener("click", btnChange));
btns.forEach((button) => button.removeEventListener("click", btnChange));
```

Подписка на событие через addEventListener

Метод addEventListener в отличие от предыдущих способов позволяет назначить одному событию несколько обработчиков:

```
function handler1() { ... }
function handler2() { ... }

document.addEventListener('click', handler1);
document.addEventListener('click', handler2);
```

Пример спойлеров

```
<h2>Квест №17</h2>
<div class="spoiler-button">Спойлер</div>
<div class="spoiler-text hide">
    К 4-му пульту забираемся по ящику и наклонной лестнице у стены. Крутим платформу,
    передвигаем вагон в центр. По двум площадкам прыгаем к 5-му пульту на центральном
    островке. Если вагон не едет, то нужно повернуть площадку впереди.
</div>
<div class="spoiler-button">Спойлер</div>
<div class="spoiler-text hide">
    Спускаемся вниз, проходим ворота. Садимся в красный автомобиль, едем дальше.
    Слева можно осмотреть несколько домов.
</div>

<script>
document.querySelectorAll(".spoiler-button").forEach( sp =>{
    sp.addEventListener("click", function (){
        this.nextElementSibling.classList.toggle("hide");
    });
});
</script>
```

Пример спойлеров

```
<style>
.spoiler-button {
    margin-bottom: 10px;
    padding: 5px;
    border-bottom: 1px dashed grey;
    cursor: pointer;
    color: cadetblue;
}
.spoiler-button:hover {
    border-bottom: 1px solid grey;
}
.spoiler-text {
    background-color: #eee;
    border: 1px solid #ddd;
    margin: 5px;
    padding: 5px;
}
.hide{
    display: none;
}
</style>
```

Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее.

Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

```
<body>
    <button id="btn1">Нажми меня!</button>
    <script>
        function handleClick(evt){
            console.log(this);
            console.log(evt);
            console.log(evt.type);
            console.log(evt.x);
            console.log(evt.y);
            console.log(evt.target.tagName);
            console.log(evt.currentTarget.tagName);
        }
        const btn1 = document.querySelector("#btn1");
        btn1.addEventListener("click", handleClick);
    </script>
</body>
```

Всплытие и погружение событий

Рассмотрим пример:

```
<div>
    <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Est, quibusdam sed.
    <strong>Officiis dolorum</strong>, minus necessitatibus quasi quam temporibus
    asperiores labore rerum laudantium ipsa adipisci eos explicabo expedita
    dignissimos, earum modi.</p>
</div>

<script>
    function divClick(event){ console.log("Div here!");}
    function pClick(event){ console.log("P here!");}
    function strongClick(event){ console.log("Strong here!");}

    document.querySelector("div").addEventListener('click', divClick);
    document.querySelector("p").addEventListener('click', pClick);
    document.querySelector("strong").addEventListener('click', strongClick);
</script>
```

Если кликнуть на содержимое тега strong, то сработают все три обработчика.

Всплытие

Принцип всплытия очень простой: когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

Например, есть 3 вложенных элемента `DIV > P > STRONG` с обработчиком на каждом. Клик по внутреннему `<strong>` вызовет обработчик onclick:

- сначала на самом `<strong>`
- потом на внешнем `<p>`
- затем на внешнем `<div>`
- и так далее вверх по цепочке до самого document.

Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде.

Почти все события всплывают. Ключевое слово в этой фразе – «почти».
Например, событие `focus` не всплывает. Однако, стоит понимать, что это скорее исключение, чем правило, всё-таки большинство событий всплывают.

Всплытие: 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.

```
function divClick(event){
    console.log(`${this.tagName} ${event.target.tagName}`);
}
function pClick(event){
    console.log(`${this.tagName} ${event.target.tagName}`);
}
function strongClick(event){
    console.log(`${this.tagName} ${event.target.tagName}`);
}
```

Прекращение всплытия

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()`.

```
function divClick(event){
    console.log(`1 ${this.tagName} ${event.target.tagName}`);
}
function pClick(event){
    console.log(`2 ${this.tagName} ${event.target.tagName}`);
}
function strongClick(event){
    console.log(`3 ${this.tagName} ${event.target.tagName} `);
    event.stopPropagation();
}
```

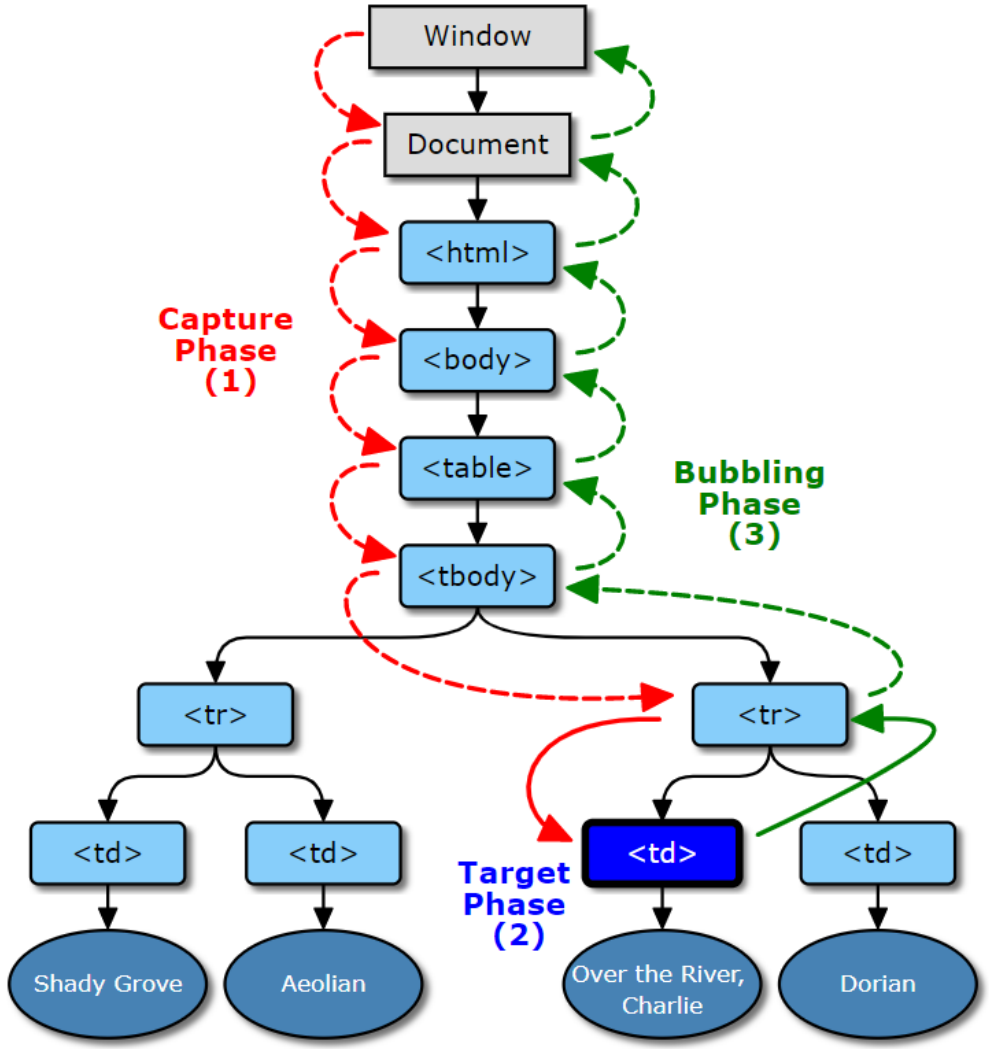
Погружение

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт `DOM Events` описывает 3 фазы прохода события:

- Фаза погружения (`capturing phase`) – событие сначала идёт сверху вниз.
- Фаза цели (`target phase`) – событие достигло целевого(исходного) элемента.
- Фаза всплытия (`bubbling stage`) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке `<td>`, расположенной внутри таблицы. 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`:

```
elem.addEventListener(..., {capture: true})
// или просто "true", как сокращение для {capture: true}
elem.addEventListener(..., true)
```

Делегирование событий

Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – делегирование.

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

```
<main id="main">
    <h1>События</h1>
    <button class="btn" data-btn-id="1"><span>Нажми меня</span></button>
    <button class="btn" data-btn-id="2"><span>Нажми меня</span></button>
    <button class="btn" data-btn-id="3"><span>Нажми меня</span></button>
    <button class="btn" data-btn-id="4"><span>Нажми меня</span></button>
</main>
<script>
    function btnClick(btn) {
        console.log(`click ${btn.tagName} ${btn.dataset.btnId}`);
    }
    document.getElementById("main").addEventListener('click', function (event) {
        if (event.target.closest(".btn")) {
            btnClick(event.target.closest(".btn"));
        }
    });`
</script>
```

Действия браузера по умолчанию

Некоторые события автоматически вызывают действие браузера, встроенное по умолчанию как реакция на определенный тип события: переход по ссылке, отправка формы и т. п. Как правило их можно, и зачастую нужно, отменить.

Например:
- Клик по ссылке инициирует переход на новый URL, указанный в href ссылки.
- Отправка формы — перезагрузку страницы.

```
<a href="https://mephi.ru" class="no-link">МИФИ</a>
```

В случае, если используется свойство on{событие}, то для предотвращения поведения необходимо в функции обработчике вернуть false:

```
function noLink(){
    console.log("link2");
    return false;
}
document.querySelector(".no-link").onclick = noLink;
```

В случае использования addEventListener для отмены действия браузера по умолчанию на объекте события есть стандартный метод event.preventDefault()

```
document.querySelector(".no-link").addEventListener("click", function (e){
    console.log("link");
    e.preventDefault();
});
```

События мыши

- mousedown/mouseup – кнопка мыши нажата/отпущена над элементом.
- mouseover/mouseout – курсор мыши появляется над элементом и уходит с него.
- mousemove – каждое движение мыши над элементом генерирует это событие.
- click – вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши.
- dblclick – вызывается двойным кликом на элементе.
- contextmenu – вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши.

События, связанные с кликом, всегда имеют свойство button, которое позволяет получить конкретную кнопку мыши. Возможными значениями event.button являются:

- 0 - Левая кнопка (основная)
- 1 - Средняя кнопка (вспомогательная)
- 2 - Правая кнопка (вторичная)
- 3 - Кнопка X1 (назад)
- 4 - Кнопка X2 (вперёд)

Все события мыши имеют координаты двух видов:

- Относительно окна: clientX и clientY (x и y псевдонимы).
- Относительно документа: pageX и pageY.

События элементов форм

Фокусировка

Элемент получает фокус при нажатии на нем мышкой, клавиши Tab или выбрав на планшете. Момент получения фокуса и потери очень важен, при получении фокуса мы можем подгрузить данные для автозаполнения, начать отслеживать изменения. При потере фокуса — проверить введенные данные.

При фокусировке на элемент происходит событие `focus`, а когда фокус исчезает, например посетитель кликает в другом месте экрана, происходит событие `blur`.

По умолчанию многие элементы не могут получить фокус. Например, если кликнуть по div, то фокусировка на нем не произойдет. Кстати, фокус может быть только на одном элементе в единицу времени и текущий элемент, на котором фокус, доступен как **`document.activeElement`**.

Активировать или отменить фокус можно программно, вызвав в коде одноименные методы `elem.focus()` и `elem.blur()` у элемента.

Событие change

Происходит по окончании изменения элемента формы, когда изменение зафиксировано. Для `input:text` или `textarea` событие произойдёт не при каждом вводе, а при потере фокуса, что не всегда удобно.

Например пока вы набираете что-то в текстовом поле — события нет. Но как только фокус пропал, произойдет событие `change`. Для остальных же элементов, например `select, input:checkbox и input:radio`, оно срабатывает сразу при выборе значения.

Событие input

Срабатывает только на текстовых элементах, `input:text и textarea`, при изменении значения элемента. Не ждет потери фокуса, в отличие от change.

В современных браузерах `input` — самое главное событие для работы с текстовым элементом формы. Именно его, а не `keydown` или `keypress`, следует использовать.

События элементов форм

```
<form action="" method="get" id="first-form">
    <input type="text" id="inp-1">
    <input type="text" id="inp-2">
    <div id="result"></div>
</form>

<script>
    const inputs = document.querySelectorAll("#first-form input[type=text]");
    const resultDiv = document.getElementById("result");

    inputs.forEach( inp => {
        inp.addEventListener("focus", function(e){
            console.log(e.type, this);
        });
        inp.addEventListener("blur", function(e){
            console.log(e.type, this);
        });
        inp.addEventListener("input", function(e){
            console.log(e.type, this);
        });
        inp.addEventListener("change", function(e){
            console.log(e.type, this);
            let res = "";
            inputs.forEach(inp =>  {res += inp.value} );
            resultDiv.textContent = res;

        });
    });
</script>
```

Событие submit

Возникает при отправке формы. Его применяют для валидации (проверки) формы перед отправкой. Чтобы отправить форму у посетителя есть два способа:

- Нажать кнопку с `type="submit"`
- Нажать клавишу Enter, находясь в каком-нибудь поле формы

Какой бы способ ни выбрал посетитель – будет сгенерировано событие `submit`. В обработчике этого события можно проверить данные и выполнить действия по результатам проверки.
Подробнее о работах с формами тут: https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement

```
<form action="#" method="get" id="reg-form">
    <fieldset>
        <legend>Регистрация</legend>
        <dir>
            <label for="login">Имя: </label>
            <input type="text" name="login" id="login" minlength="4" required>
        </dir>
        <dir>
            <label for="passwd1">Пароль: </label>
            <input type="password" name="pass1" id="passwd1" minlength="8" autocomplete="off">
        </dir>
        <dir>
            <label for="passwd2">Повторите пароль: </label>
            <input type="password" name="pass2" id="passwd2" minlength="8" autocomplete="off">
        </dir>
        <input type="submit" value="Регистрация">
    </fieldset>
</form>
```

Событие submit

```
<script>
    // Можно обратится к форме через getElementById или querySelector
    // А можно обратится к формам документа по id через свойство forms
    const regForm = document.forms["reg-form"];
    regForm.addEventListener('submit', function (evt) {
        evt.preventDefault();
        //обращение к элементам формы
        //или через свойство соответствующее name: this.pass1 (если название позволяет)
        //или через свойство elements по name: this.elements["pass2"]

        if (this.checkValidity() && this.pass1.value === this.elements["pass2"].value) {
            this.submit();
        } else {
            alert("Пароли не совпадают!")
        }
    });
</script>
```

События клавиатуры

Есть три основных события клавиатуры: `keydown, keypress и keyup`. При нажатии клавиши сначала происходит `keydown`, после чего `keypress` и только потом `keyup`, когда клавишу отжали.

События `keydown` и `keyup` срабатывают при нажатии любой клавиши, включая служебные. А вот `keypress` срабатывает, только если нажата символьная клавиша, т. е. нажатие приводит к появлению символа. Управляющие клавиши, такие как `Ctrl`, `Shift`, `Alt` и другие, не генерируют событие `keypress`.

Свойство `KeyboardEvent.key` доступно для чтения и возвращает значение клавиши, нажатой пользователем, принимая во внимание состояние клавиш модификаторов, таких как shiftKey, а так же текущий язык и модель клавиатуры.
https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key

Свойство `KeyboardEvent.code` представляет собой физическую клавишу на клавиатуре (в отличие от символа, сгенерированного нажатием клавиши). Другими словами, это свойство возвращает значение, которое не изменяется с помощью раскладки клавиатуры или состояния клавиш-модификаторов.
https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/code

События клавиатуры

```
<div id="log"></div>
<button id="clear-log">Очистить</button>

<script>
    const logDiv = document.getElementById("log");

    function logMessage(event){
        console.log(event.type, event.code, event.key);
        const p = document.createElement("p");
        p.innerHTML = `<strong>${event.type}</strong> ${event.code} - ${event.key}`;
        logDiv.appendChild(p);
    }

    document.addEventListener("keydown", logMessage);
    document.addEventListener("keypress", logMessage);
    document.addEventListener("keyup", logMessage);

    document.getElementById('clear-log').addEventListener('click', function(){
        document.getElementById('log').innerHTML = "";
    });
</script>
```

Как правильно прикрепить обработчики к элементам?

Для прикрепления обработчиков к элементам, необходимо чтобы эти элементы на странице были доступны. Определить, когда они будут доступны с момента загрузки документам можно с помощью события DOMContentLoaded. Данное событие возникает на document когда DOM полностью построено.

```
document.addEventListener('DOMContentLoaded', function () {
  // DOM полностью построен и доступен
  //...
});
```

Справочник по событиям

https://developer.mozilla.org/ru/docs/Web/Events