JS в браузере

События

- Событие - это "нечто", происходящее с DOM элементами
- Каждое событие имеет имя и элемент, с которым оно произошло (target)
- Чтобы отреагировать на событие, можно добавить к элементу один или несколько обработчиков этого события
- Когда происходит событие, вызываются обработчики этого события
- Список событий которые могут возникнуть, зависят от типа элемента

Добавление обработчиков событий

- Атрибут HTML

```
<div onclick="alert('hi!');"></div>
```

- смешение кода и представления
- нечитаемый код (нет подсветки синтаксиса, экранирование кавычек)
- можно использовать только для HTML элементов
- можно объявить только один обработчик для одного события
- можно объявить обработчик только для стадии всплытия
- в целом - deprecated

Добавление обработчиков событий

- Свойство DOM-элемента

```
let div = document.getElementById( '#myDiv' );
div.onclick = () => alert( 'hi!' );
```

- фактически то же самое что и атрибут HTML
- можно объявить только один обработчик для одного события
- можно объявить обработчик только для стадии всплытия
- работает не для всех событий (ex.: `transitionend`)
- в целом - deprecated

Добавление обработчиков событий

- `addEventListener`/`removeEventListener`

```
let f = () => alert( 'hi!'),
 div = document.getElementById( 'myDiv');
div.addEventListener( 'click', f);
div.removeEventListener( 'click', () => alert( 'hi')); // не сработает
div.removeEventListener( 'click', f);
```

- можно задать несколько обработчиков события
- работает для всех типов событий
- работает для всех фаз обработки события
- работает для всех типов элементов (в т.ч. `window`, `document`)
- нельзя получить список установленных обработчиков
- чтобы удалить обработчик события, приходится хранить ссылку на него

Порядок обработки событий

- При возникновении события вызовутся все объявленные обработчики
- Обработчику передается один аргумент - объект события (класс `Event` и производные)
- `this` в обработчике события указывает на элемент, к которому прикреплен обработчик

```
<div id="example" onclick="console.log(event.type); console.log('html handler')" >
    example div
</div>


let div = document.getElementById('example');
div.addEventListener('click', e => console.log('handler 1'));
div.addEventListener('click', e => console.log('handler 2'));
div.addEventListener('change', e => console.log('handler for change event'));

// => 'click'
// => 'html handler'
// => 'handler 1'
// => 'handler 2'
```

Объект события

- В зависимости от события, объект события имеет тот или иной класс
- Разные классы событий могут иметь дополнительные поля и методы, специфичные для них

```
elem.addEventListener('click', function(e) {
    e instanceof MouseEvent;  // true
    e.type;           // 'click'
    e.target;         // ссылка на сам elem
    e.currentTarget; // ссылка на сам elem
    e.clientX;        // 450, специфично для событий мыши
    e.ctrlKey;        // нажата ли клавиша ctrl
});

elem.addEventListener('keypress', function(e) {
    e instanceof KeyboardEvent;  // true
    e.ctrlKey;        // нажата ли клавиша ctrl
    e.key;            // 'a'
    e.keyCode;        // 'KeyA'
    e.repeat;         // повторяется ли символ
})
```

Объект события

`e.type`	Имя события (e.g. `'click'`, `'change'`)
`e.currentTarget`	Элемент, на котором сработал обработчик
`e.target`	Элемент-инициатор события
`e.bubbles`	Всплывает ли событие
`e.cancelable`	Может ли быть отменено действие по умолчанию
`e.eventPhase`	Фаза в которой находится событие (число)
`e.isTrusted`	вызвано действием пользователя - `true`, создано программно - `false`

Всплытие событий

- Всплытие события - вызов всех обработчиков события, начиная с элемента-инициатора вверх по дереву DOM - до `document` и `window`
- Некоторые события не всплывают (e.g. `focus`, `blur`)
- У невсплывающих событий часто есть всплывающие аналоги (e.g. `focusin`, `focusout`)
- `e.bubbles` - флаг; всплывает событие или нет

```
<div onclick="console.log('div 1')">1
  <div onclick="console.log('div 2')">2
    <div onclick="console.log('div 3')">3</div>
  </div>
</div>
// => 'div 3'
// => 'div 2'
// => 'div 1'
```

Перехват событий

- Перед всплытием есть еще один этап - фаза перехвата
- Перехват работает так же как и всплытие, но в обратном порядке
- `addEventListener(eventName, handler, true)` назначает обработчик на фазу перехвата

```
<div id="div1">1
  <div id="div2">2
    <div id="div3">3</div>
  </div>
</div>
```

```
let div1 = document.getElementById('div1'),
    div2 = document.getElementById('div2'),
    div3 = document.getElementById('div3');

div1.addEventListener('click', e => console.log(`div 1`), true);
div2.addEventListener('click', e => console.log(`div 2`), true);
div3.addEventListener('click', e => console.log(`div 3`), true);

// 'div 1', 'div 2', 'div 3'
```

Перехват и всплытие: все вместе

- (1. Перехват) Вызываются все обработчики фазы перехвата, начиная от `window` вниз к элементу - инициатору
- (1.1) Несколько обработчиков фазы всплытия вызываются в порядке объявления
- (2. Цель) Вызываются все обработчики события на элементе-инициаторе в порядке объявления, вне зависимости от указанной фазы
- (3. Всплытие) Вызываются все обработчики фазы всплытия, начиная от элемента - инициатора вверх к `window`
- (4. Default action) Выполняется действие по умолчанию (если не было отменено ни одним из обработчиков)

Перехват и всплытие: все вместе

```
<div id="div1">1
  <div id="div2">2
    <div id="div3">3</div>
  </div>
</div>
```

```
let div1 = document.getElementById('div1'),
    div2 = document.getElementById('div2'),
    div3 = document.getElementById('div3');

div1.addEventListener('click', (e) => console.log(`div1 down ${e.eventPhase}`));
div2.addEventListener('click', (e) => console.log(`div2 down ${e.eventPhase}`));
div3.addEventListener('click', (e) => console.log(`div3 down ${e.eventPhase}`));
div1.addEventListener('click', (e) => console.log(`div1 up ${e.eventPhase}`), true);
div2.addEventListener('click', (e) => console.log(`div2 up ${e.eventPhase}`), true);
div3.addEventListener('click', (e) => console.log(`div3 up ${e.eventPhase}`), true);

// div1 up 1
// div2 up 1
// div3 down 2
// div3 up 2
// div2 down 3
// div1 down 3
```

Прерывание перехвата/всплытия

- `e.stopPropagation()`- отмена всплытия события. Обработчики события на текущем элементе при этом вызываются
- `e.stopImmediatePropagation()`- отмена всплытия события, и отмена невызванных обработчиков текущего элемента

```
<div onclick="console.log('div 1')">1
  <div onclick="console.log('div 2'); event.stopPropagation();" >2
    <div onclick="console.log('div 3')">3</div>
  </div>
</div>

<!-- 'div 3', 'div 2' -->
```

Действие по умолчанию

- Некоторые события предполагают действие по умолчанию: e.g. клик на ссылке вызывает переход по url
- `e.preventDefault()` - отменяет действие по умолчанию
- `return false;` - также отменяет действие по умолчанию
- `e.cancelable` - флаг; можно ли отменить действие по умолчанию
- `e.defaultPrevented` - флаг; было ли отменено действие по умолчанию

```
<input type="checkbox" onclick="return false;" />
<!--"Кому нужны книжки без картинок и чекбоксы без поведения по умолчанию?", - думала Алиса
```

```
let checkbox = document.querySelector('input[type="checkbox"]');
checkbox.addEventListener('click', e => {console.log(e.defaultPrevented)});
// => true, событие по умолчанию отменено обработчиком из атрибута
```

Делегирование обработки событий

- Объявление обработчиков событий требует времени и памяти
- Добавление обработчиков событий для динамически генерируемых элементов - дополнительный уровень сложности

```
<ul id="items-list">
    <li data-item-id="1">Товар 1</li>
    <li data-item-id="2">Товар 2</li>
    <li data-item-id="3">Товар 3</li>
</ul>


let ul = document.getElementById('items-list');

for (let li of ul.children) {
    // Для каждого li - свой обработчик
    li.addEventListener('click', function(e) {
        addToBasket(li.dataset.itemId);
    });
}
```

Делегирование обработки событий

- С помощью `this` или `e.currentTarget` мы можем обойтись одним обработчиком для всех однотипных элементов
- С помощью всплытия и поля `e.target` мы можем делегировать обработку этого события родительскому элементу

```
<ul id="items-list">
    <li data-item-id="1">Товар 1</li>
    <li data-item-id="2">Товар 2</li>
    <li data-item-id="3">Товар 3</li>
</ul>
```

```
let ul = document.getElementById('items-list');

// Один обработчик для всех li
let basketHandler = function(e) {
    addToBasket(this.dataset.itemId);
};

for (let li of ul.children) {
    li.addEventListener('click', basketHandler);
}
```

```
let ul = document.getElementById('items-list');

// Один обработчик для всех li
let basketHandler = function(e) {
    addToBasket(e.target.dataset.itemId);
};

// Родительский элемент обрабатывает событие
ul.addEventListener('click', basketHandler);
```

Делегирование обработки событий

- Один обработчик вместо множества однотипных => экономия времени/памяти
- Динамическое добавление/удаление дочерних элементов не требует дополнительной обработки
- Проблема: событие может возникнуть не на том элементе который нам нужен, или внутри искомого элемента

Делегирование обработки событий

```
<ul id="items-list">
    <li class="item" data-item-id="1">Товар 1</li>
    <li class="item" data-item-id="2">Товар 2 <img src="..." /></li>
    <li class="item" data-item-id="3">Товар 3 <span>Скидка -10%</span></li>
    <li><button>Кнопка управления</button></li>
</ul>


function addDelegateListener(eventName, rootElement, childSelector, callback ) {
    rootElement.addEventListener(eventName,  function(e) {
        // Ищем ближайшего подходящего родителя от элемента-инициатора вверх
        let elem = e.target.closest(childSelector);
        if (!elem) return;

        // Находится ли найденный элемент внутри корневого
        if (!rootElement.contains(elem)) return;

        // Вызываем обработчик в контексте нужного элемента
        callback.call(elem, e);
    })
};

let ul = document.getElementById('items-list');
addDelegateListener('click', ul, 'li.item', function(e) {
    addToBasket(this.dataset.itemId);
});
```

Загрузка документа: `load` и `DOMContentLoaded`

- `DOMContentLoaded`- происходит когда документ загружен и готов к работе
- `load` - происходит когда загружен документ и зависящие от него файлы

```
<html>
<head>
    <script type="text/javascript">
        // ничего не найдет: документ еще не загружен
        document.getElementById('div1');

        window.addEventListener('DOMContentLoaded', function() {
            // сработает: документ полностью загружен и готов к работе
            document.getElementById('div1');
        });
    </script>
</head>
<body>
    <div id="div1"></div>

    <script type="text/javascript">
        // сработает: документ еще не загружен полностью, но тег div уже доступен
        document.getElementById('div1');
    </script>
</body>
</html>
```

Создание собственных событий

- Конструктор `Event(eventName [, flags])`создает новый объект события
- Можно также использовать конструкторы `MouseEvent`, `WheelEvent`
- Конструктор `CustomEvent` позволяет с флагами указать поле `detail`
- Можно также вызвать встроенные события: `elem.focus()`

```
document.addEventListener('myEvent',
    (e) => console.log(`${e.type}, ${e.target.tagName}, ${e.detail.message}`));

let e = new CustomEvent('myEvent', {bubbles: true, detail: {message: 'hello'}});
document.body.dispatchEvent(e);
// 'myEvent, BODY, hello'
```

Создание собственных событий

- В 99% процентах случаев - грязный хак, и решение лежит уровнем выше
- Иногда неизбежно для работы со сторонними библиотеками
- Оправдано при автоматизированном тестировании (имитация действий пользователя)
- Оправдано при создании собственных UI элементов

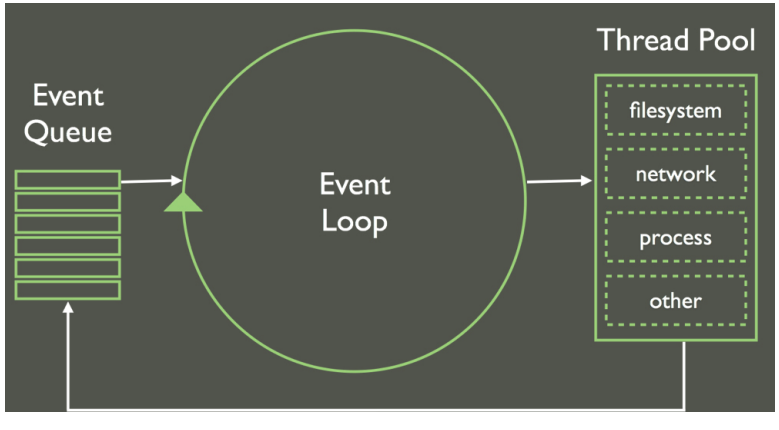
Отложенное (асинхронное) выполнение кода

- `setTimeout(fn, delay)` - выполнить функцию `fn` не раньше чем через `delay` миллисекунд
- `setInterval(fn, delay)` - выполнять функцию `fn` с интервалами не меньше `delay` миллисекунд
- `clearTimeout(timerId)` - отменить ранее созданный таймер
- `clearInterval(timerId)` - отменить ранее созданный интервал
- минимальная задержка - 4мс

```
console.log('start');

let timer1 = setTimeout(() => console.log('timer 1 fired'), 10);
let timer2 = setTimeout(() => console.log('timer 2 fired'), 10);

clearTimeout(timer2);

console.log('finish');

// 'start', 'finish', 'timer 1 fired'
```

Асинхронное выполнение кода

- JS - *однопоточный* язык с асинхронной моделью обработки событий
- Главный цикл обработки событий зашит глубоко внутри среды JS
- Асинхронный код готовый к выполнению ставится в очередь ожидания
- На каждой итерации из очереди берется очередной кусок кода и выполняется синхронно
- Зациклившийся скрипт останавливает всю работу UI



Асинхронное выполнение кода

```
console.log('script started');
setTimeout(() => console.log('timed out function'), 5);

let endDate = Date.now() + 10;
while (Date.now() < endDate) ; // делаем "ничего" 10 мс
console.log('script finished');

// 'script started'
// ? 'script finished'
// ? 'timed out function'


let timeoutId;
setTimeout(() => {
    console.log('timed out function');
    clearTimeout(timeoutId);
}, 5);

timeoutId = setTimeout(() => console.log('timed out function 2'), 5);

let endDate = Date.now() + 10;
while (Date.now() < endDate) ; // делаем "ничего" 10 мс
// 'timed out function'
```

Асинхронное выполнение кода

`setTimeout/setInterval`	async
`alert/prompt/confirm`	sync
обработчик события вызванного пользователем	async
обработчик события вызванного программно	sync
обработчик события `DOMSubtreeModified`	sync
AJAX запросы	async

Асинхронное выполнение кода

- Сначала выполняются обработчики события, затем действие по умолчанию
  - Кнопка не "нажмется", пока не отработает обработчик клика
  - Обработчик клика не запустится, пока очередь не пуста
  - "Тяжелые" вычисления блокируют весь UI браузера
- Всегда следует отдавать предпочтение асинхронным вызовам
- Для выполнения больших объемов вычислений можно использовать `WebWorkers`