Лабораторная работа № 4

Тема: *Объекты клиентских приложений. Обработка событий JavaScript.*

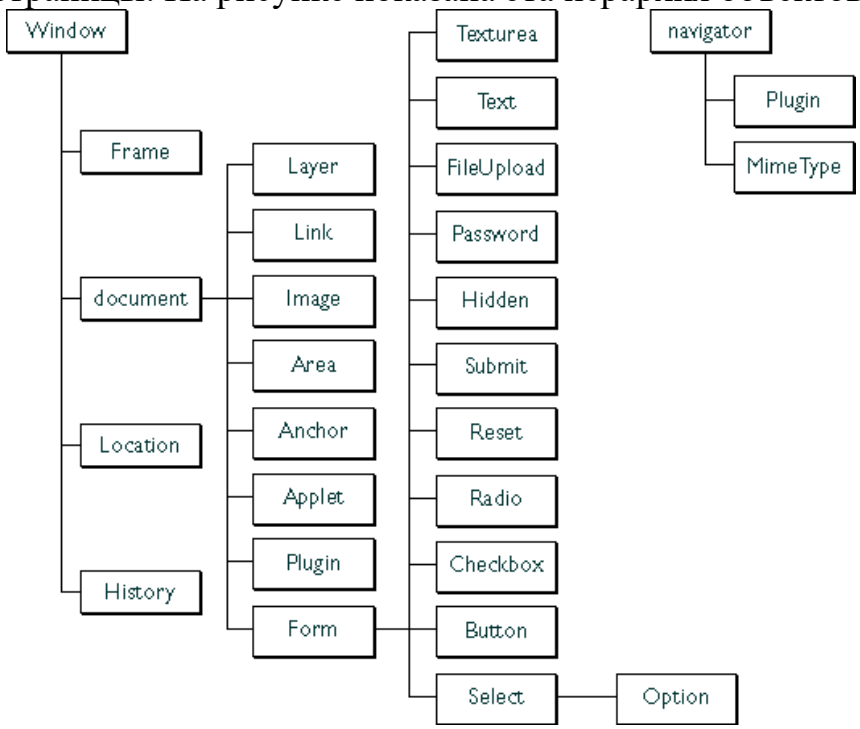
Цель: *ознакомиться с объектами клиентских приложений, изучить принцип работы обработки событий JavaScript, научиться применять полученные знания на практике.*

Краткая теория

1.1 ОБЪЕКТЫ КЛИЕНТСКИХ ПРИЛОЖЕНИЙ

Объекты клиентского JavaScript иногда называются *объектами Navigator'a*, чтобы отличить их от серверных или пользовательских объектов.

Когда Вы загружаете документ в Navigator, он создаёт объекты JavaScript со значениями свойств, базируясь на HTML документа и другой сопутствующей информации. Эти объекты расположены иерархически, что отражает структуру самой HTML-страницы. На рисунке показана эта иерархия объектов.



В данной иерархии "потомки" объектов являются их свойствами. Например, форма form1 является объектом, а также свойством объекта document, и к ней обращаются document.form1.

Список всех объектов, их свойств, методов и обработчиков событий см. в книге *Клиентский JavaScript. Справочник*.

На каждой странице имеются следующие объекты:

- **navigator**: имеет свойства - имя и версию используемого Navigator'a, MIME-типы, поддерживаемые клиентом и plug-in'ы, установленные на клиенте.
- **window**: объект верхнего уровня/top-level; имеет свойства, которые применяются ко всему окну. Каждое "дочернее окно" в документе с фрэймами также является window-объектом.
- **document**: имеет свойства на основе содержимого документа, такого как заголовок, цвет фона, гиперссылки и формы.

- **location**: имеет свойства на основе текущего URL.
- **history**: имеет свойства, представляющие URL'ы, которые клиент запрашивал ранее.

В зависимости от содержимого, документ может содержать и другие объекты. Например, каждая форма (определённая тэгом FORM) в документе имеет соответствующий объект Form.

Объекты window и Frame

Объект window является "родительским" объектом для всех объектов в Navigator'e. Вы можете создать несколько окон в приложении JavaScript. Объект Frame определяется тэгом FRAME в документе FRAMESET. Frame-объекты имеют те же свойства и методы, что и объекты window, и отличаются только способом отображения.

Объект window имеет несколько широко используемых методов, в том числе:

- open и close: открывают и закрывают окно браузера; Вы можете специфицировать размер окна, его содержимое и наличие панели кнопок/button bar, адресной строки/location field и других "chrome"-атрибутов.
- alert - Выводит диалоговое окно Alert с сообщением.
- confirm - Выводит диалоговое окно Confirm с кнопками OK и Cancel.
- prompt - Выводит диалоговое окно Prompt с текстовым полем для ввода значения.
- blur и focus - Убирают и передают фокус окну.
- scrollTo - Прокручивает окно на специфицированные координаты.
- setInterval - Вычисляет выражение или вызывает функцию многократно по истечении специфицированного периода времени.
- setTimeout - Вычисляет выражение или вызывает функцию однократно по истечении специфицированного периода времени.

window имеет также несколько свойств, которые могут устанавливаются Вами, таких как location и status.

Вы можете установить location для перехода клиента к другому URL. Например, следующий оператор перенаправляет клиент на домашнюю страницу Netscape, как если бы пользователь щёлкнул по гиперссылке или как-нибудь иначе загрузил URL:

location = "http://home.netscape.com"

Объект document

Каждая страница имеет единственный объект document.

Поскольку его методы write и writeln генерируют HTML, объект document является одним из наиболее используемых объектов Navigator'a. О методах write и writeln см. раздел "Использование Метода write".

Объект document имеет несколько свойств, отражающих цвет фона, текста и гиперссылок страницы: bgColor, fgColor, linkColor, alinkColor и vlinkColor. Часто используются lastModified, дата последнего изменения страницы, referrer,

предыдущий URL, посещённый клиентом, и URL, URL документа. Свойство cookie даёт возможность устанавливать и получать значения кук; см. также "Использование Кук".

Объект document является предком всех объектов Anchor, Applet, Area, Form, Image, Layer, Link и Plugin страницы.

Пользователи могут печатать и сохранять сгенерированный HTML, используя команды меню File Navigator'a (JavaScript 1.1 и позднее).

Объект Form

Каждая форма документа создаёт объект Form. Поскольку в документе может быть не одна форма, Form-объекты хранятся в массиве forms. Первая форма (самая верхняя на странице) это forms[0], вторая - forms[1], и так далее. Помимо обращения к форме по имени, Вы можете обратиться к первой (например) форме так:

document.forms[0]

Другие элементы формы, такие как текстовые поля, радио-кнопки и т.д., хранятся в массиве elements. Вы можете обратиться к первому элементу (независимо от его вида) первой формы так:

document.forms[0].elements[0]

Каждый элемент формы имеет свойство form, которое является ссылкой на родительскую форму элемента. Это свойство используется в основном в обработчиках событий, где может понадобиться обратиться к другому элементу на текущей форме. В следующем примере форма myForm содержит Text-объект и кнопку. Если пользователь щёлкает по кнопке, значением Text-объекта становится имя формы. Обработчик onClick кнопки использует this.form для обращения к родительской форме, myForm.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
  onClick="this.form.text1.value=this.form.name">
</FORM>
```

Объект location

Объект location имеет свойства на основе текущего URL. Например, свойство hostname это сервер и имя домена сервера - хоста текущего документа.

Объект location имеет два метода:

- reload - форсирует перезагрузку текущего документа окна.
- replace - загружает специфицированный URL поверх текущего вхождения списка history.

1.2 ОБРАБОТКА СОБЫТИЙ JavaScript

Введение в 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).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

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. Мы подробно разберём их в последующих главах.

Обработчики событий

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. Сейчас мы их рассмотрим, начиная с самого простого.

Есть три способа назначения обработчиков событий:

1. Атрибут HTML: onclick="...".
2. DOM-свойство: elem.onclick = function.
3. Специальные методы: elem.addEventListener(event, handler[, phase]) для добавления, removeEventListener для удаления.

HTML-атрибуты используются редко потому, что JavaScript в HTML-теге выглядит немного странно. К тому же много кода там не напишешь.

DOM-свойства вполне можно использовать, но мы не можем назначить больше одного обработчика на один тип события. Во многих случаях с этим ограничением можно мириться.

Последний способ самый гибкий, однако нужно писать больше всего кода. Есть несколько типов событий, которые работают только через него, к примеру transitionend и DOMContentLoaded. Также addEventListener поддерживает объекты в качестве обработчиков событий. В этом случае вызывается метод объекта handleEvent.

Не важно, как вы назначаете обработчик – он получает объект события первым аргументом. Этот объект содержит подробности о том, что произошло.

Всплытие и погружение

При наступлении события – самый глубоко вложенный элемент, на котором оно произошло, помечается как «целевой» (event.target).

- Затем событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(...., true), где true – это сокращение для {capture: true}.
- Далее обработчики вызываются на целевом элементе.
- Далее событие двигается от event.target вверх к корню документа, по пути вызывая обработчики, поставленные через on<event> и addEventListener без третьего аргумента или с третьим аргументом равным false.

Каждый обработчик имеет доступ к свойствам события event:

- event.target – самый глубокий элемент, на котором произошло событие.
- event.currentTarget (=this) – элемент, на котором в данный момент сработал обработчик (тот, на котором «висит» конкретный обработчик)
- event.eventPhase – на какой фазе он сработал (погружение=1, фаза цели=2, всплытие=3).

Любой обработчик может остановить событие вызовом event.stopPropagation(), но делать это не рекомендуется, так как в дальнейшем это событие может понадобиться, иногда для самых неожиданных вещей.

В современной разработке стадия погружения используется очень редко, обычно события обрабатываются во время всплытия. И в этом есть логика.

В реальном мире, когда происходит чрезвычайная ситуация, местные службы реагируют первыми. Они знают лучше всех местность, в которой это произошло, и другие детали. Вышестоящие инстанции подключаются уже после этого и при необходимости.

Тоже самое справедливо для обработчиков событий. Код, который «навесил» обработчик на конкретный элемент, знает максимум деталей об элементе и его предназначении. Например, обработчик на определённом <td> скорее всего подходит только для этого конкретного <td>, он знает все о нём, поэтому он должен отработать первым. Далее имеет смысл передать обработку события родителю – он тоже понимает, что происходит, но уже менее детально, далее – выше, и так далее, до самого объекта document, обработчик на котором реализовывает самую общую функциональность уровня документа.

Всплытие и погружение являются основой для «делегирования событий» – очень мощного приёма обработки событий. Его мы изучим в следующей главе.

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.

Рассмотрим пример – диаграмму Ба-Гуа. Это таблица, отражающая древнюю китайскую философию.

Вот она:

Квадрат*Bagua*: Направление, Элемент, Цвет, Значение

Северо-Запад Металл Серебро Старейшины	**Север** Вода Синий Перемены	**Северо-Восток** Земля Жёлтый Направление
Запад Металл Золото Молодость	**Центр** Всё Пурпурный Гармония	**Восток** Дерево Синий Будущее
Юго-Запад Земля Коричневый Спокойствие	**Юг** Огонь Оранжевый Слава	**Юго-Восток** Дерево Зелёный Роман

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, не важно.

Наша задача – реализовать подсветку ячейки ***<td>*** *при клике.*

Вместо того, чтобы назначать обработчик onclick для каждой ячейки <td> (их может быть очень много) – мы повесим «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; // где был клик?
  if (target.tagName != 'TD') return; // не на TD? тогда не интересует
  highlight(target); // подсветить TD
};
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. Мы можем 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 нашем случае, если взглянуть на HTML-код таблицы внимательно, видно, что ячейка <td> содержит вложенные теги, например :

```
<td>
  <strong>Северо-Запад</strong>
  ...
</td>
```

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

Внутри обработчика table.onclick мы должны по event.targetразобраться, 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)
};
```

Разберём пример:

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае нам 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 мы должны проверить, действительно ли это <td> *нашей таблицы*.
4. И если это так, то подсвечиваем его.

В итоге мы получили короткий код подсветки, быстрый и эффективный, которому совершенно не важно, сколько всего в таблице <td>.

Зачем использовать:

- Упрощает процесс инициализации и экономит память: не нужно вешать много обработчиков.
- Меньше кода: при добавлении и удалении элементов не нужно ставить или снимать обработчики.
- Удобство изменений DOM: можно массово добавлять или удалять элементы путём изменения innerHTML и ему подобных.

Действия браузера по умолчанию

Действий браузера по умолчанию достаточно много:

- mousedown – начинает выделять текст (если двигать мышкой).
- click на <input type="checkbox"> – ставит или убирает галочку в input.
- submit – при нажатии на <input type="submit"> или при нажатии клавиши Enter в форме данные отправляются на сервер.
- keydown – при нажатии клавиши в поле ввода появляется символ.
- contextmenu – при правом клике показывается контекстное меню браузера.
- …и многие другие…

Все эти действия можно отменить, если мы хотим обработать событие исключительно при помощи JavaScript.

Чтобы отменить действие браузера по умолчанию, используйте event.preventDefault() или return false. Второй метод работает, только если обработчик назначен через on<событие>.

Опция passive: true для addEventListener сообщает браузеру, что действие по умолчанию не будет отменено. Это очень полезно для некоторых событий на мобильных устройствах, таких как touchstart и touchmove, чтобы сообщить браузеру, что он не должен ожидать выполнения всех обработчиков, а ему следует сразу приступать к выполнению действия по умолчанию, например, к прокрутке.

Если событие по умолчанию отменено, то значение event.defaultPrevented становится true, иначе false.

Генерация пользовательских 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 нашего меню, реализованного при помощи JavaScript, может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. А другой код может слушать эти события и узнавать, что происходит с меню.

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как click, mousedown и другие. Это бывает полезно для автоматического тестирования.

Конструктор Event

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс Event.

Событие встроенного класса Event можно создать так:

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

Где:

- *type* – тип события, строка, например "click" или же любой придуманный нами – "my-event".
- *options* – объект с тремя необязательными свойствами:
  - bubbles: true/false – если true, тогда событие всплывает.
  - cancelable: true/false – если true, тогда можно отменить действие по умолчанию. Позже мы разберём, что это значит для пользовательских событий.
  - composed: true/false – если true, тогда событие будет всплывать наружу за пределы Shadow DOM. Позже мы разберём это в разделе Веб-компоненты.

По умолчанию все три свойства установлены в **false**: {bubbles: false, cancelable: false, composed: false}.

Метод dispatchEvent

После того, как объект события создан, мы должны запустить его на элементе, вызвав метод elem.dispatchEvent(event).

Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан флаг bubbles, то оно будет всплывать.

В примере ниже событие click инициируется JavaScript-кодом так, как будто кликнули по кнопке:

```
<button id="elem" onclick="alert('Клик!');">Автоклик</button>
<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

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.

Пример всплытия

Мы можем 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>
  // ловим на document...
  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);
  // обработчик на document сработает и выведет сообщение.
</script>
```

Обратите внимание:

1. Мы должны использовать addEventListener для наших собственных событий, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает.
2. Мы обязаны передать флаг bubbles:true, иначе наше 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.

MouseEvent, KeyboardEvent и другие

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:

- UIEvent
- FocusEvent
- MouseEvent
- WheelEvent
- KeyboardEvent
- …

Стоит использовать их вместо new Event, если мы хотим 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: этого нельзя было бы сделать с обычным конструктором Event.

Давайте 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 вы найдёте в спецификации, например, MouseEvent.

Пользовательские события

Для генерации событий совершенно новых типов, таких как "hello", следует использовать конструктор new CustomEvent. Технически 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>
  // дополнительная информация приходит в обработчик вместе с событием
  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.

Кроме того, класс события описывает, что это за событие, и если оно не браузерное, а пользовательское, то лучше использовать CustomEvent, чтобы явно об этом сказать.

event.preventDefault()

Для многих браузерных событий есть «действия по умолчанию», такие как переход по ссылке, выделение и т.п.

Для новых, пользовательских событий браузерных действий, конечно, нет, но код, который генерирует такое событие, может предусматривать какие-то свои действия после события.

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, узнает, что продолжать не нужно.

Посмотрим практический пример – прячущегося кролика (могло бы быть скрывающееся меню или что-то ещё).

Ниже вы можете видеть кролика #rabbit и функцию hide(), которая при вызове генерирует на нём событие "hide", уведомляя всех интересующихся, что кролик собирается спрятаться.

Любой обработчик может узнать об этом, подписавшись на событие hideчерез rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:

```
<pre id="rabbit">
  |\   /|
   \|_|/
   /. .\
  =\_Y_/=
   {>o<}
</pre>
<button onclick="hide()">Hide()</button>
<script>
  // hide() будет вызван автоматически через 2 секунды
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true // без этого флага preventDefault не сработает
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('Действие отменено обработчиком');
```

```
    } else {
      rabbit.hidden = true;
    }
  }
  rabbit.addEventListener('hide', function(event) {
    if (confirm("Вызвать preventDefault?")) {
      event.preventDefault();
    }
  });
</script>
```

Обратите внимание: событие должно содержать флаг cancelable: true. Иначе, вызов event.preventDefault() будет проигнорирован.

Вложенные события обрабатываются синхронно

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);
    // alert("вложенное событие")
    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));
    alert(2);
  };
  document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>
```

Порядок вывода: 1 → вложенное событие → 2.

Обратите внимание, что вложенное событие menu-open успевает всплыть и запустить обработчик 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 ситуаций. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Если нам это не подходит, то мы можем либо поместить dispatchEvent(или любой другой код, инициирующий события) в конец обработчика onclick, либо,

если это неудобно, можно обернуть генерацию события в setTimeout с нулевой задержкой:

```
<button id="menu">Меню (нажми меня)</button>
<script>
 menu.onclick = function() {
  alert(1);
  // alert(2)
  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.

Новый порядок вывода: 1 → 2 → вложенное событие.

Интерфейсные события

Основы событий мыши

В этой главе мы более детально рассмотрим события мыши и их свойства.

Сразу заметим: эти события бывают не только из-за мыши, но и эмулируются на других устройствах, в частности, на мобильных, для совместимости.

Типы событий мыши

Мы можем разделить события мыши на две категории: «простые» и «комплексные».

Простые события

Самые часто используемые простые события:

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.

contextmenu

Вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши. Но, заметим, это не совсем событие мыши, оно может вызываться и специальной клавишей клавиатуры.

…Есть также несколько иных типов событий, которые мы рассмотрим позже.

Комплексные события

click

Вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши.

dblclick

Вызывается двойным кликом на элементе.

Комплексные события состоят из простых, поэтому в теории мы могли бы без них обойтись. Но хорошо, что они существуют, потому что работать с ними очень удобно.

Порядок событий

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.

Получение информации о кнопке: which

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.

Но если мы отслеживаем mousedown и mouseup, то оно нам нужно, потому что эти события срабатывают на любой кнопке, и which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки».

Есть три возможных значения:

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

Средняя кнопка сейчас – скорее экзотика, и используется очень редко.

Модификаторы: shift, alt, ctrl и meta

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах.

Свойства объекта события:

- shiftKey: Shift
- altKey: Alt (или Opt для Mac)
- ctrlKey: Ctrl
- metaKey: Cmd для Mac

Они равны true, если во время события была нажата соответствующая клавиша.

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>
```

Координаты: clientX/Y, pageX/Y

Все события мыши имеют координаты двух видов:

1. Относительно окна: clientX и clientY.
2. Относительно документа: pageX и pageY.

Например, если у нас есть окно размером 500x500, и курсор мыши находится в левом верхнем углу, то значения clientX и clientY равны 0. А если мышь находится в центре окна, то значения clientX и clientY равны 250 независимо от того, в каком месте документа она находится и до какого места документ прокручен. В этом они похожи на position:fixed.

Наведите курсор мыши на поле ввода, чтобы увидеть clientX/clientY(пример находится в iframe, поэтому координаты определяются относительно этого iframe):

```
<input onmousemove="this.value=event.clientX+':'+event.clientY" value="Наведи на меня мышь">
```

Координаты относительно документа pageX, pageY отсчитываются не от окна, а от левого верхнего угла документа.

Движение мыши: mouseover/out, mouseenter/leave

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 target, а другой relatedTarget.

Для события mouseover:

- event.target – это элемент, *на который* курсор перешёл.
- event.relatedTarget – это элемент, *с которого* курсор ушёл (relatedTarget → target).

Для события mouseout наоборот:

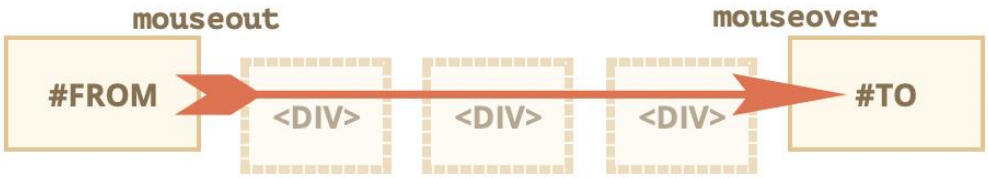
- event.target – это элемент, *с которого* курсор ушёл.
- event.relatedTarget – это элемент, *на который* курсор перешёл (target → relatedTarget).

Пропуск элементов

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:

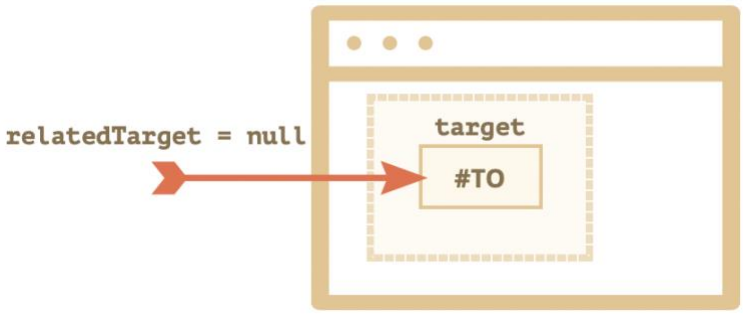


Если курсор мыши передвинуть очень быстро с элемента #FROM на элемент #TO, как это показано выше, то лежащие между ними элементы (или некоторые из них) могут быть пропущены. Событие mouseout может запуститься на элементе #FROM и затем сразу же сгенерируется mouseover на элементе #TO.

Это хорошо с точки зрения производительности, потому что если промежуточных элементов много, вряд ли мы действительно хотим обрабатывать вход и выход для каждого.

С другой стороны, мы должны иметь в виду, что указатель мыши не «посещает» все элементы на своём пути. Он может и «прыгать».

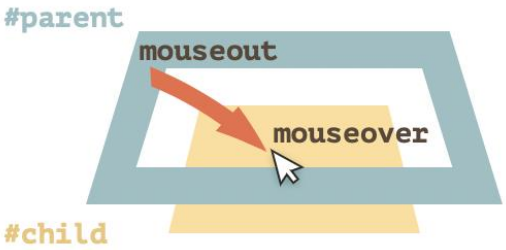
В частности, возможно, что указатель запрыгнет в середину страницы из-за пределов окна браузера. В этом случае значение relatedTargetбудет null, так как курсор пришёл «из ниоткуда»:



Событие mouseout при переходе на потомка

Важная особенность события mouseout – оно генерируется в том числе, когда указатель переходит с элемента на его потомка.

То есть, визуально указатель всё ещё на элементе, но мы получим mouseout!



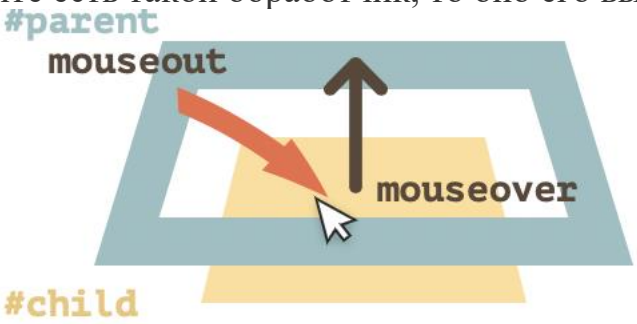
Это выглядит странно, но легко объясняется.

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.

Обратите внимание на важную деталь.

Событие mouseover, происходящее на потомке, всплывает. Поэтому если на родительском элементе есть такой обработчик, то оно его вызовет.



События mouseenter и 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, на его потомки и с них, не считаются.
2. События mouseenter/mouseleave не всплывают.

События mouseenter/mouseleave предельно просты и понятны.

Когда указатель появляется над элементом – генерируется mouseenter, причём не имеет значения, где именно указатель: на самом элементе или на его потомке.

Событие mouseleave происходит, когда курсор покидает элемент.

Клавиатура: keydown и keyup

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

event.code и event.key

Свойство key объекта события позволяет получить символ, а свойство code – «физический код клавиши».

К примеру, одну и ту же 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".

«KeyZ» и другие клавишные коды

У каждой клавиши есть код, который зависит от её расположения на клавиатуре. Подробно о клавишных кодах можно прочитать в спецификации о кодах событий UI.

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.

! Регистр важен: "KeyZ", а не "keyZ" !

Выглядит очевидно, но многие всё равно ошибаются.

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

А что, если клавиша не буквенно-цифровая? Например, Shift или F1, или какая-либо другая специальная клавиша? В таких случаях 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

Обратите внимание, что event.code точно указывает, какая именно клавиша нажата. Так, 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 «смысле» клавиши: что вообще было нажато (Shift).

Допустим, мы хотим обработать горячую клавишу Ctrl+Z (или Cmd+Z для Mac). Большинство текстовых редакторов к этой комбинации подключают действие «Отменить». Мы можем поставить обработчик событий на keydown и проверять, какая клавиша была нажата.

Здесь возникает дилемма: в нашем обработчике стоит проверять значение event.key или event.code?

С одной стороны, значение event.key – это символ, он 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.

Например, вот схема стандартной (US) раскладки («QWERTY») и под ней немецкой («QWERTZ») раскладки (из Википедии):

Для одной и той же клавиши в американской раскладке значение event.code равно «Z», в то время как в немецкой «Y».

Буквально, для пользователей с немецкой раскладкой event.code при нажатии на Y будет равен KeyZ.

Если мы будем проверять в нашем коде event.code == 'KeyZ', то для людей с немецкой раскладкой такая проверка сработает, когда они нажимают Y.

Звучит очень странно, но это и в самом деле так. В спецификации прямо упоминается такое поведение.

Так что event.code может содержать неправильный символ при неожиданной раскладке. Одни и те же буквы на разных раскладках могут сопоставляться с разными физическими клавишами, что приводит к разным кодам. К счастью, это происходит не со всеми кодами, а с несколькими, например KeyA, KeyQ, KeyZ (как мы уже видели), и не происходит со специальными клавишами, такими как Shift.

Чтобы отслеживать символы, зависящие от раскладки, event.key надёжнее.

С другой стороны, преимущество event.code заключается в том, что его значение всегда остаётся 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.

Хотим поддерживать клавиши, меняющиеся при раскладке? Тогда event.key – верный выбор.

Или мы хотим, чтобы горячая клавиша срабатывала даже после переключения на другой язык? Тогда event.code может быть лучше.

Автоповтор

При долгом нажатии клавиши возникает автоповтор: keydownсрабатывает снова и снова, и когда клавишу отпускают, то отрабатывает keyup. Так что ситуация, когда много keydownи один keyup, абсолютно нормальна.

Для событий, вызванных автоповтором, у объекта события свойство event.repeat равно true.

Действия по умолчанию

Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре.

Для примера:

- Появление символа (самое очевидное).
- Удаление символа (клавиша Delete).
- Прокрутка страницы (клавиша PageDown).
- Открытие диалогового окна браузера «Сохранить» (Ctrl+S)
- …и так далее.

Свойства и методы формы

Формы и элементы управления, такие как <input>, имеют множество специальных свойств и событий.

Работать с формами станет намного удобнее, когда мы их изучим.

Навигация: формы и элементы

Формы в документе входят в специальную коллекцию document.forms.

Это так называемая «именованная» коллекция: мы можем использовать для получения формы как её имя, так и порядковый номер в документе.

```
document.forms.my - форма с именем "my" (name="my")
document.forms[0] - первая форма в документе
```

Когда мы уже получили форму, любой элемент доступен в именованной коллекции form.elements.

Например:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>
<script>
  // получаем форму
  let form = document.forms.my; // <form name="my"> element
  // получаем элемент
  let elem = form.elements.one; // <input name="one"> element
  alert(elem.value); // 1
</script>
```

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio.

В этом случае form.elements[name] является коллекцией, например:

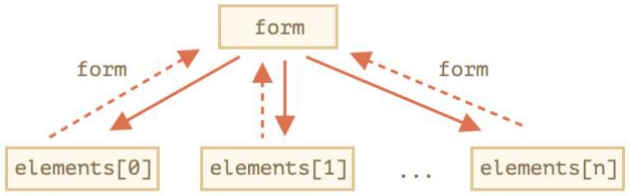
```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>
<script>
let form = document.forms[0];
let ageElems = form.elements.age;=
alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы управления формы, как бы глубоко они не находились в форме, доступны в коллекции form.elements.

Обратная ссылка: element.form

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму.

Вот иллюстрация:



Пример:

```
<form id="form">
  <input type="text" name="login">
</form>
<script>
  // form -> element
  let login = form.login;
  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

Элементы формы

Рассмотрим элементы управления, используемые в формах.

input и textarea

К их значению можно получить доступ через свойство input.value(строка) или input.checked (булево значение) для чекбоксов.

Вот так:

```
input.value = "Новое значение";
textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей
```

! Используйте textarea.value вместо textarea.innerHTML !

Обратим внимание: хоть элемент <textarea>...</textarea> и хранит своё значение как вложенный HTML, нам не следует использовать textarea.innerHTML для доступа к нему.

Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

select и option

Элемент <select> имеет 3 важных свойства:

1. select.options – коллекция из подэлементов <option>,
2. select.value – значение выбранного в данный момент <option>,
3. select.selectedIndex – номер выбранного <option>.

Они дают три разных способа установить значение в <select>:

1. Найти соответствующий элемент <option> и установить в option.selected значение true.
2. Установить в select.value значение нужного <option>.
3. Установить в select.selectedIndex номер нужного <option>.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе.

Вот эти способы на примере:

```
<select id="select">
  <option value="apple">Яблоко</option>
  <option value="pear">Груша</option>
  <option value="banana">Банан</option>
</select>
<script>
  // все три строки делают одно и то же
  select.options[2].selected = true;
  select.selectedIndex = 2;
  select.value = 'banana';
</script>
```

В отличие от большинства других элементов управления, <select>позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут multiple. Эту возможность используют редко, но в этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство selected у подэлементов <option>.

Их коллекцию можно получить как select.options, например:

```
<select id="select" multiple>
  <option value="blues" selected>Блюз</option>
  <option value="rock" selected>Рок</option>
  <option value="classic">Классика</option>
</select>
<script>
  // получаем все выбранные значения из select с multiple
```

```
  let selected = Array.from(select.options)
    .filter(option => option.selected)
    .map(option => option.value);
  alert(selected); // blues,rock
</script>
```

new Option

Элемент <option> редко используется сам по себе, но и здесь есть кое-что интересное.

В спецификации есть красивый короткий синтаксис для создания элемента <option>:

```
option = new Option(text, value, defaultSelected, selected);
```

Параметры:

- text – текст внутри <option>,
- value – значение,
- defaultSelected – если true, то ставится HTML-атрибут selected,
- selected – если true, то элемент <option> будет выбранным.

Тут может быть небольшая путаница с defaultSelected и selected. Всё просто: defaultSelected задаёт HTML-атрибут, его можно получить как option.getAttribute('selected'), а selected – выбрано значение или нет, именно его важно поставить правильно. Впрочем, обычно ставят оба этих значения в true или не ставят вовсе (т.е. false).

Пример:

```
let option = new Option("Текст", "value");
// создаст <option value="value">Текст</option>
```

Тот же элемент, но выбранный:

```
let option = new Option("Текст", "value", true, true);
```

Элементы <option> имеют свойства:

option.selected

Выбрана ли опция.

option.index

Номер опции среди других в списке <select>.

option.text

Содержимое опции (то, что видит посетитель).

Фокусировка: focus/blur

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу Tab. Также существует HTML-атрибут autofocus, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса, о них – далее.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализировать или загрузить что-нибудь.

Момент потери фокуса («blur») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает Tab, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса, о них – далее.

Потеря фокуса обычно означает «данные введены», и мы можем выполнить проверку введённых данных или даже отправить эти данные на сервер и так далее.

События focus/blur

Событие focus вызывается в момент фокусировки, а blur – когда элемент теряет фокус.

Используем их для валидации(проверки) введённых данных.

В примере ниже:

- Обработчик blur проверяет, введён ли email, и если нет – показывает ошибку.
- Обработчик focus скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>
Ваш email: <input type="email" id="input">
<div id="error"></div>
<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // не email
    input.classList.add('invalid');
    error.innerHTML = 'Пожалуйста, введите правильный email.'
  }
};
input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // удаляем индикатор ошибки, т.к. пользователь хочет ввести данные заново
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```

Включаем фокусировку на любом элементе: tabindex

Многие элементы по умолчанию не поддерживают фокусировку.

Какие именно – зависит от браузера, но одно всегда верно: поддержка focus/blur гарантирована для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д.

С другой стороны, элементы форматирования , , <table>– по умолчанию не могут получить фокус. Метод elem.focus() не работает для них, и события focus/blur никогда не срабатывают.

Это можно изменить HTML-атрибутом tabindex.

Любой элемент поддерживает фокусировку, если имеет tabindex. Значение этого атрибута – порядковый номер элемента, когда клавиша Tab (или что-то аналогичное) используется для переключения между элементами.

То есть: если у нас два элемента, первый имеет tabindex="1", а второй tabindex="2", то находясь в первом элементе и нажав Tab – мы переместимся во второй.

Порядок перебора таков: сначала идут элементы со значениями tabindex от 1 и выше, в порядке tabindex, а затем элементы без tabindex (например, обычный <input>).

При совпадающих tabindex элементы перебираются в том порядке, в котором идут в документе.

Есть два специальных значения:

- tabindex="0" ставит элемент в один ряд с элементами без tabindex. То есть, при переключении такие элементы будут после элементов с tabindex ≥ 1.

Обычно используется, чтобы включить фокусировку на элементе, но не менять порядок переключения. Чтобы элемент мог участвовать в форме наравне с обычными <input>.

- tabindex="-1" позволяет фокусироваться на элементе только программно. Клавиша Tab проигнорирует такой элемент, но метод elem.focus() будет действовать.

События: change, input, cut, copy, paste

Событие	Описание	Особенности
`change`	Значение было изменено.	Для текстовых полей срабатывает при потере фокуса.
`input`	Срабатывает при каждом изменении значения.	Запускается немедленно, в отличие от `change`.
`cut/copy/paste`	Действия по вырезанию/копированию/вставке.	Действие можно предотвратить. Свойство `event.clipboardData` предоставляет доступ на чтение/запись в буфер обмена...

Отправка формы: событие и метод submit

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод form.submit() позволяет инициировать отправку формы из JavaScript. Мы можем использовать его для динамического создания и отправки наших собственных форм на сервер.

Событие: submit

Есть два основных способа отправить форму:

1. Первый – нажать кнопку <input type="submit"> или <input type="image">.
2. Второй – нажать Enter, находясь на каком-нибудь поле.

Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

Метод: submit

Чтобы отправить форму на сервер вручную, мы можем вызвать метод form.submit().

При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку.

Иногда это используют для генерации формы и отправки её вручную, например так:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';
form.innerHTML = '<input name="q" value="test">';
// перед отправкой формы, её нужно вставить в документ
document.body.append(form);
form.submit();
```

Открытие окон и методы window

Всплывающие окна используются нечасто. Ведь загрузить новую информацию можно динамически, а показать – в элементе , расположенным над страницей (z-index). Ещё одна альтернатива – тег <iframe>.

Если мы открываем попап, хорошей практикой будет предупредить пользователя об этом. Иконка открывающегося окошка на ссылке поможет посетителю понять, что происходит и не потерять оба окна из поля зрения.

- Новое окно можно открыть с помощью вызова open(url, name, params). Этот метод возвращает ссылку на это новое окно.
- По умолчанию браузеры блокируют вызовы open, выполненные не в результате действий пользователя. Обычно браузеры показывают предупреждение, так что пользователь все-таки может разрешить вызов этого метода.
- Вместо попапа открывается вкладка, если в вызове open не указаны его размеры.

- У попапа есть доступ к породившему его окну через свойство window.opener.
- Если основное окно и попап имеют один домен и протокол, то они свободно могут читать и изменять друг друга. В противном случае, они могут только изменять положение друг друга и взаимодействовать с помощью сообщений.

Чтобы закрыть попап: метод close(). Также попап может закрыть и пользователь (как и любое другое окно). После закрытия окна свойство window.closed имеет значение true.

- Методы focus() и blur() позволяют установить или убрать фокус с попапа. Но работают не всегда.
- События focus и blur позволяют отследить получение и потерю фокуса новым окном. Но, пожалуйста, не забывайте, что окно может остаться видимым и после blur.

Общение между окнами

Политика "Одинакового источника"

Два URL имеют «одинаковый источник» в том случае, если они имеют совпадающие протокол, домен и порт.

Эти URL имеют одинаковый источник:

- http://site.com
- http://site.com/
- http://site.com/my/page.html

А эти – разные источники:

- http://**www.**site.com (другой домен: www. важен)
- http://**site.org** (другой домен: .org важен)
- **https://**site.com (другой протокол: https)
- http://site.com:**8080** (другой порт: 8080)

Политика «Одинакового источника» говорит, что:

- если у нас есть ссылка на другой объект window, например, на всплывающее окно, созданное с помощью window.open или на window из <iframe> и у этого окна тот же источник, то к нему будет полный доступ.
- в противном случае, если у него другой источник, мы не сможем обращаться к его переменным, объекту document и так далее. Единственное исключение – объект location: его можно изменять (таким образом перенаправляя пользователя). Но нельзя читать location (нельзя узнать, где находится пользователь, чтобы не было никаких утечек информации).

Доступ к содержимому ифрейма

Внутри <iframe> находится по сути отдельное окно с собственными объектами document и window.

Мы можем обращаться к ним, используя свойства:

- iframe.contentWindow ссылка на объект window внутри <iframe>.

- iframe.contentDocument – ссылка на объект document внутри <iframe>, короткая запись для iframe.contentWindow.document.

Если окна имеют одинаковый источник (протокол, домен, порт), то они могут делать друг с другом всё, что угодно.

В противном случае возможны только следующие действия:

- Изменение свойства location другого окна (доступ только на запись).
- Отправить туда сообщение.

Исключения:

- Окна, которые имеют общий домен второго уровня: a.site.com и b.site.com. Установка свойства document.domain='site.com' в обоих окнах переведёт их в состояние «Одинакового источника».
- Если у ифрейма установлен атрибут sandbox, это принудительно переведёт окна в состояние «разных источников», если не установить в атрибут значение allow-same-origin. Это можно использовать для запуска ненадёжного кода в ифрейме с того же сайта.

Метод postMessage позволяет общаться двум окнам с любыми источниками:

1. Отправитель вызывает targetWin.postMessage(data, targetOrigin).
2. Если targetOrigin не '*', тогда браузер проверяет имеет ли targetWin источник targetOrigin.
3. Если это так, тогда targetWin вызывает событие message со специальными свойствами:
   - origin – источник окна отправителя (например, http://my.site.com)
   - source – ссылка на окно отправитель.
   - data – данные, может быть объектом везде, кроме IE (в IE только строки).

В окне-получателе следует добавить обработчик для этого события с помощью метода addEventListener.

Куки, document.cookie

Куки – это небольшие строки данных, которые хранятся непосредственно в браузере. Они являются частью HTTP-протокола, определённого в спецификации RFC 6265.

Куки обычно устанавливаются веб-сервером при помощи заголовка Set-Cookie. Затем браузер будет автоматически добавлять их в (почти) каждый запрос на тот же домен при помощи заголовка Cookie.

Один из наиболее частых случаев использования куки – это аутентификация:

1. При входе на сайт сервер отсылает в ответ HTTP-заголовок Set-Cookie для того, чтобы установить куки со специальным уникальным идентификатором сессии («session identifier»).
2. Во время следующего запроса к этому же домену браузер посылает на сервер HTTP-заголовок Cookie.
3. Таким образом, сервер понимает, кто сделал запрос.

Мы также можем получить доступ к куки непосредственно из браузера, используя свойство document.cookie.

Настройки куки:

- path=/, по умолчанию устанавливается текущий путь, делает куки видимым только по указанному пути и ниже.
- domain=site.com, по умолчанию куки видно только на текущем домене, если явно указан домен, то куки видно и на поддоменах.
- expires или max-age устанавливает дату истечения срока действия, без них куки умрёт при закрытии браузера.
- secure делает куки доступным только при использовании HTTPS.
- samesite запрещает браузеру отправлять куки с запросами, поступающими извне, помогает предотвратить XSRF-атаки.

Дополнительно:

- Сторонние куки могут быть запрещены браузером, например Safari делает это по умолчанию.
- Установка отслеживающих куки пользователям из стран ЕС требует их явного согласия на это в соответствии с законодательством GDPR.

LocalStorage, sessionStorage

Объекты веб-хранилища localStorage и sessionStorage позволяют хранить пары ключ/значение в браузере.

Что в них важно – данные, которые в них записаны, сохраняются после обновления страницы (в случае sessionStorage) и даже после перезапуска браузера (при использовании localStorage). Скоро мы это увидим.

Но ведь у нас уже есть куки. Зачем тогда эти объекты?

- В отличие от куки, объекты веб-хранилища не отправляются на сервер при каждом запросе. Поэтому мы можем хранить гораздо больше данных. Большинство браузеров могут сохранить как минимум 2 мегабайта данных (или больше), и этот размер можно поменять в настройках.
- Ещё одно отличие от куки – сервер не может манипулировать объектами хранилища через HTTP-заголовки. Всё делается при помощи JavaScript.
- Хранилище привязано к источнику (домен/протокол/порт). Это значит, что разные протоколы или поддомены определяют разные объекты хранилища, и они не могут получить доступ к данным друг друга.

Объекты хранилища localStorage и sessionStorage предоставляют одинаковые методы и свойства:

- setItem(key, value) – сохранить пару ключ/значение.
- getItem(key) – получить данные по ключу key.
- removeItem(key) – удалить данные с ключом key.
- clear() – удалить всё.
- key(index) – получить ключ на заданной позиции.
- length – количество элементов в хранилище.

Как видим, интерфейс похож на Map (setItem/getItem/removeItem), но также запоминается порядок элементов, и можно получить доступ к элементу по индексу – key(index).

Давайте посмотрим, как это работает.

Демо localStorage

Основные особенности localStorage:

- Этот объект один на все вкладки и окна в рамках источника (один и тот же домен/протокол/порт).
- Данные не имеют срока давности, по которому истекают и удаляются. Сохраняются после перезапуска браузера и даже ОС.

Например, если запустить этот код…

```
localStorage.setItem('test', 1);
```

…И закрыть/открыть браузер или открыть ту же страницу в другом окне, то можно получить данные следующим образом:

```
alert( localStorage.getItem('test') ); // 1
```

Нам достаточно находиться на том же источнике (домен/протокол/порт), при этом URL-путь может быть разным.

Объект localStorage доступен всем окнам из одного источника, поэтому, если мы устанавливаем данные в одном окне, изменения становятся видимыми в другом.

Доступ как к обычному объекту

Также можно получать/записывать данные, как в обычный объект:

```
// установить значение для ключа
localStorage.test = 2;
// получить значение по ключу
alert( localStorage.test ); // 2
// удалить ключ
delete localStorage.test;
```

Это возможно по историческим причинам и, как правило, работает, но обычно не рекомендуется, потому что:

1. Если ключ генерируется пользователем, то он может быть каким угодно, включая length или toString или другой встроенный метод localStorage. В этом случае getItem/setItem сработают нормально, а вот чтение/запись как свойства объекта не пройдут:

```
2.  let key = 'length';
localStorage[key] = 5; // Ошибка, невозможно установить length
```

3. Когда мы модифицируем данные, то срабатывает событие storage. Но это событие не происходит при записи без setItem, как свойства объекта. Мы увидим это позже в этой главе.

Перебор ключей

Методы, которые мы видим, позволяют читать/писать/удалять данные. А как получить все значения или ключи?

К сожалению, объекты веб-хранилища нельзя перебрать в цикле, они не итерируемы.

Но можно пройти по ним, как по обычным массивам:

```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Другой способ – использовать цикл, как по обычному объекту `for key in localStorage`.

Здесь перебираются ключи, но вместе с этим выводятся несколько встроенных полей, которые нам не нужны:

```
// bad try
for(let key in localStorage) {
  alert(key); // покажет getItem, setItem и другие встроенные свойства
}
```

…Поэтому нам нужно либо отфильтровать поля из прототипа проверкой hasOwnProperty:

```
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
    continue; // пропустит такие ключи, как "setItem", "getItem" и так далее
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

…Либо просто получить «собственные» ключи с помощью Object.keys, а затем при необходимости вывести их при помощи цикла:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Последнее работает, потому что `Object.keys` возвращает только ключи, принадлежащие объекту, игнорируя прототип.

Только строки

Обратите внимание, что ключ и значение должны быть строками.

Если мы используем любой другой тип, например число или объект, то он автоматически преобразуется в строку:

```
sessionStorage.user = {name: "John"};
alert(sessionStorage.user); // [object Object]
```

Мы можем использовать JSON для хранения объектов:

```
sessionStorage.user = JSON.stringify({name: "John"});
// немного позже
let user = JSON.parse( sessionStorage.user );
```

```
alert( user.name ); // John
```

Также возможно привести к строке весь объект хранилища, например для отладки:

```
// для JSON.stringify добавлены параметры форматирования, чтобы объект выглядел лучше
alert( JSON.stringify(localStorage, null, 2) );
```

sessionStorage

Объект sessionStorage используется гораздо реже, чем localStorage.

Свойства и методы такие же, но есть существенные ограничения:

- sessionStorage существует только в рамках текущей вкладки браузера.
  - Другая вкладка с той же страницей будет иметь другое хранилище.
  - Но оно разделяется между ифреймами на той же вкладке (при условии, что они из одного и того же источника).
- Данные продолжают существовать после перезагрузки страницы, но не после закрытия/открытия вкладки.

Давайте посмотрим на это в действии.

Запустите этот код…

```
sessionStorage.setItem('test', 1);
```

…И обновите страницу. Вы всё ещё можете получить данные:

```
alert( sessionStorage.getItem('test') ); // после обновления: 1
```

…Но если вы откроете ту же страницу в другой вкладке и попробуете получить данные снова, то код выше вернёт null, что значит «ничего не найдено».

Так получилось, потому что sessionStorage привязан не только к источнику, но и к вкладке браузера. Поэтому sessionStorageиспользуется нечасто.

Событие storage

Когда обновляются данные в localStorage или sessionStorage, генерируется событие storage со следующими свойствами:

- key – ключ, который обновился (null, если вызван .clear()).
- oldValue – старое значение (null, если ключ добавлен впервые).
- newValue – новое значение (null, если ключ был удалён).
- url – url документа, где произошло обновление.
- storageArea – объект localStorage или sessionStorage, где произошло обновление.

Важно: событие срабатывает на всех остальных объектах window, где доступно хранилище, кроме того окна, которое его вызвало.

Давайте уточним.

Представьте, что у вас есть два окна с одним и тем же сайтом. Хранилище localStorage разделяется между ними.

Вы можете открыть эту страницу в двух окнах браузера, чтобы проверить приведённый ниже код.

Теперь, если оба окна слушают window.onstorage, то каждое из них будет реагировать на обновления, произошедшие в другом окне.

```
// срабатывает при обновлениях, сделанных в том же хранилище из других документов
window.onstorage = event => {
  if (event.key != 'now') return;
  alert(event.key + ':' + event.newValue + " at " + event.url);
};
localStorage.setItem('now', Date.now());
```

Обратите внимание, что событие также содержит: event.url – url-адрес документа, в котором данные обновились.

Также event.storageArea содержит объект хранилища – событие одно и то же для sessionStorage и localStorage, поэтому event.storageArea ссылается на то хранилище, которое было изменено. Мы можем захотеть что-то записать в ответ на изменения.

!Это позволяет разным окнам одного источника обмениваться сообщениями. !

Современные браузеры также поддерживают Broadcast channel APIспециальный API для связи между окнами одного источника, он более полнофункциональный, но менее поддерживаемый. Существуют библиотеки (полифилы), которые эмулируют это API на основе localStorage и делают его доступным везде.

Задание 1

Создайте галерею изображений, в которой основное изображение изменяется при клике на уменьшенный вариант. Используйте делегирование.

Например:

Задание 2

Создайте список, в котором элементы могут быть выделены, как в файловых менеджерах.

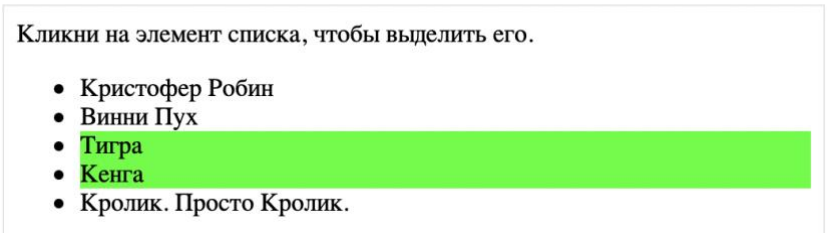
- *При клике на элемент списка выделяется только этот элемент (добавляется класс .selected), отменяется выделение остальных элементов.*

- *Если клик сделан вместе с* `Ctrl` *(*`Cmd` *для Mac), то выделение переключается на элементе, но остальные элементы при этом не изменяются.*

! В этом задании все элементы списка содержат только текст. Без вложенных тегов.

! Предотвратите стандартное для браузера выделение текста при кликах.

Демо:



Задание 3

Создайте функцию `runOnKeys(func, code1, code2, ... code_n)`*, которая запускает* `func` *при одновременном нажатии клавиш с кодами* `code1`*,* `code2`*, ...,* `code_n`*.*

Например, код ниже выведет `alert` *при одновременном нажатии клавиш* `"Q"` *и* `"W"` *(в любом регистре, в любой раскладке)*

```
runOnKeys(
  () => alert("Привет!"),
  "KeyQ",
  "KeyW"
);
```

Задание 4

Создайте интерфейс, позволяющий ввести сумму банковского вклада и процент, а затем рассчитать, какая это будет сумма через заданный промежуток времени.

Любое изменение введённых данных должно быть обработано немедленно.

Формула:

```
// initial: начальная сумма денег
// interest: проценты, например, 0.05 означает 5% в год
// years: сколько лет ждать
let result = Math.round(initial * (1 + interest * years));
```

Демо-версия:

! Контрольные вопросы !

1. Какие объекты есть на каждой странице?
2. Как называются объекты клиентского JavaScript?
3. Опишите работу объектов window и form.
4. Опишите работу объектов location и document.
5. Что такое событие? Виды событий.
6. Перечислите события мыши, клавиатурные события, события документа.
7. Назовите три способа назначения обработчиков событий?
8. Дайте определение понятия *делегирование*. Опишите суть работы.
9. Дайте определение понятия *фокусировка*, события при фокусировке.
10. Опишите принцип работы события storage.
11. Какой метод можно вызвать, чтобы отправить форму на сервер вручную?
12. Что такое куки? В каких ситуациях используются куки?

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.