Интерфейсные события

События мыши

Все события мыши имеют координаты двух видов:

- Относительно **окна**: **clientX** и **clientY**.
- Относительно **документа**: **pageX** и **pageY**.

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">
```

mousedown/mouseup

Кнопка мыши нажата/отпущена над элементом (**клик**).

mouseover/mouseout

Курсор мыши появляется над элементом и уходит с него (**ховер**) **с учетом внутренних элементов**. 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).

Свойство relatedTarget может быть **null**. Это нормально и означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или же, наоборот, ушёл за пределы окна.Следует держать в уме такую возможность при использовании event.relatedTarget в своём коде. Если, например, написать event.relatedTarget.tagName, то при отсутствии event.relatedTarget будет ошибка.

События **mouseover/out** возникают, даже когда происходит переход с **родительского элемента на потомка**. С точки зрения браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным. Событие mouseover, происходящее на потомке, всплывает.

mouseenter/mouseleave

События mouseenter/mouseleave похожи на mouseover/mouseout (**ховер**). Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его. Но есть и пара важных отличий:

- Переходы внутри элемента, на его потомки и с них, не считаются.
- События mouseenter/mouseleave не всплывают.

mousemove

Каждое движение мыши над элементом генерирует это событие (**ховер**). 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.

Несмотря на то, что при быстрых переходах промежуточные элементы могут игнорироваться, в одном мы можем быть уверены: элемент может быть пропущен только **целиком**. Если указатель «официально» зашёл на элемент, то есть было событие **mouseover**, то при выходе с него обязательно будет **mouseout**.

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.

click - вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши (**клик**).

dblclick - вызывается двойным кликом на элементе (**клик**).

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.

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.

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>
```

Внимание: обычно на Mac используется клавиша Cmd вместо Ctrl. Если мы хотим, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нам нужно проверять metaKey. Для JS-кода это означает, что мы должны проверить **if (event.ctrlKey || event.metaKey)**

Отменяем выделение при двойном клике мышью:

Двойной клик мыши имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: он выделяет текст. Есть несколько способов запретить выделение (см. **Selection** и **Range**). Действие по умолчанию события **mousedown** – начало выделения, если в интерфейсе оно скорее мешает, его можно отменить:

```
<b ondblclick="alert('Клик!')" onmousedown="return false">
  Сделайте двойной клик на мне
</b>
```

Для предотвращения копирования мы можем использовать другое событие: **oncopy**.

```
<div oncopy="alert('Копирование запрещено!');return false">
```

Drag'n'Drop с событиями мыши

Базовый алгоритм Drag’n’Drop выглядит так:

- При mousedown – готовим элемент к перемещению, если необходимо (например, создаём его копию).
- Затем при mousemove передвигаем элемент на новые координаты путём смены left/top и position:absolute.
- При mouseup – остановить перенос элемента и произвести все действия, связанные с окончанием Drag’n’Drop.

Чтобы не было «дублирования» элемента, нужно отменить браузерный Drag’n’Drop:

```
elem.ondragstart = function() {
  return false;
};
```

Также можно сохранить координаты элемента, за который пользователь «захватил» элементЖ

```
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

Выявляем потенциальные цели переноса под указателем (так как элемент переносится поверх других элементов) с помощью `document.elementFromPoint`:

```
// внутри обработчика события мыши
ball.hidden = true; // (*) прячем переносимый элемент, иначе мы получим именно его координаты
let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow - элемент под мячом (возможная цель переноса)
ball.hidden = false;
```

События клавиатуры.

Нажатие клавиши всегда генерирует клавиатурное событие, будь то буквенно-цифровая клавиша или специальная типа Shift или Ctrl и т.д. Единственным исключением является клавиша **Fn**, которая присутствует на клавиатуре некоторых ноутбуков. События на клавиатуре для неё нет, потому что она обычно работает на уровне более низком, чем даже ОС. В прошлом события клавиатуры иногда использовались для отслеживания ввода данных пользователем в полях формы. Это ненадёжно, потому как ввод данных не обязательно может осуществляться с помощью клавиатуры. Существуют события input и change специально для обработки ввода. Они срабатывают в результате любого ввода, включая Копировать/Вставить мышью и распознавание речи.
События клавиатуры же должны использоваться только по назначению – для клавиатуры. Например, чтобы реагировать на горячие или специальные клавиши.

Событие **keydown** происходит при нажатии клавиши (может срабатывать автоповтор), а **keyup** – при отпускании.
event.key объекта события позволяет получить символ,
event.code – «физический код клавиши».

```
inputName.addEventListener("keydown", function (event) {
  console.log(event.key); // q Q
  console.log(event.code); // KeyQ KeyQ
});
```

У каждой клавиши есть **код**, который зависит от её расположения на клавиатуре. Например:

- Буквенные клавиши имеют коды по типу "Key<буква>": "KeyA", "KeyB" и т.д. (регистр важен!)
- 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.

Клавиша	event.key	event.code
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight или ShiftLeft

В обработчиках лучше проверять ***event.code***. Однако, event.code может содержать неправильный символ при разных раскладках. 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. Можно найти полный список проблемных кодов в спецификации (https://www.w3.org/TR/uievents-code/#table-key-code-alphanumeric-writing-system). Чтобы отслеживать символы, зависящие от раскладки, е***vent.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.

При долгом нажатии клавиши возникает **автоповтор**: keydown срабатывает снова и снова, и когда клавишу отпускают, то отрабатывает **keyup**. Так что ситуация, когда много keydown и один keyup, абсолютно нормальна.
Для событий, вызванных автоповтором, у объекта события свойство **event.repeat = true**

События **keypress**, а также свойства **keyCode**, **charCode**, **which** у объекта события устарели, но еще поддерживаются.

События с действиями копировать, вырезать и вставить

Событие копирования - **copy**.Может быть с помощью клавиатуры, мыши. Можно копирование слушать у **window**, если мы хотим слушать для всего документа. Пример **запрета копирования** для всего документа:

```
window.addEventListener("copy", function (event) {
  console.log("Fired!");
  event.preventDefault();
});
```

Пример события вставки **paste.** Можно также слушать у **window**

```
window.addEventListener('paste', function(event){
   console.log("Fired!11");
});
```

Пример события вырезать **cut:**

```
window.addEventListener("cut", function (event) {
  console.log("Fired!11");
});
```

Прокрутка

Событие прокрутки **scroll** позволяет реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые при этом можно сделать. Например: Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь; Подгрузить данные, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

```
window.addEventListener('scroll', function() {
  document.getElementById('showScroll').innerHTML = pageYOffset + 'px';
});
```

Событие scroll работает как на window, так и на других элементах, на которых включена прокрутка.

Предотвращение прокрутки

Нельзя предотвратить прокрутку, используя event.preventDefault() в обработчике onscroll, потому что он срабатывает после того, как прокрутка уже произошла. Но можно предотвратить прокрутку, используя event.preventDefault() на событии, которое **вызывает прокрутку**, например, на событии **keydown** для клавиш pageUp и pageDown. Если поставить на них обработчики, в которых вызвать event.preventDefault(), то прокрутка не начнётся.

Способов инициировать прокрутку много, поэтому более надёжный способ – использовать CSS, свойство overflow.

События указателя

События указателя (**Pointer events**) – это современный способ обработки ввода с помощью различных указывающих устройств, таких как мышь, перо/стилус, сенсорный экран и так далее. К настоящему времени спецификация **Pointer Events Level 2** (https://www.w3.org/TR/pointerevents2/) поддерживается всеми основными браузерами, а **Pointer Events Level 3** (https://w3c.github.io/pointerevents/) находится в разработке и почти полностью совместима с Pointer Events Level 2. Если вы не разрабатываете под старые браузеры, такие как Internet Explorer 10, Safari 12, или более ранние версии, больше нет необходимости использовать события мыши или касаний – можно переходить сразу на **события указателя**.

Схема именований событий указателя похожа на события мыши:

Событие указателя	**Аналогичное событие мыши**
pointerdown	mousedown
pointerup	mouseup
pointermove	mousemove
pointerover	mouseover
pointerout	mouseout
pointerenter	mouseenter
pointerleave	mouseleave

Событие указателя	Аналогичное событие мыши
pointercancel	-
gotpointercapture	-
lostpointercapture	-

Свойства событий указателя

События указателя содержат те же свойства, что и события мыши, например **clientX/Y, target** и т.п., и несколько дополнительных:

- **pointerId** – уникальный идентификатор указателя, вызвавшего событие. Идентификатор генерируется браузером. Это свойство позволяет обрабатывать несколько указателей, например, сенсорный экран со стилусом и мульти-тач (увидим примеры ниже).
- **pointerType** – тип указывающего устройства. Должен быть строкой с одним из значений: «mouse», «pen» или «touch». Мы можем использовать это свойство, чтобы определять разное поведение для разных типов указателей.
- **isPrimary** – равно true для основного указателя (первый палец в мульти-тач).

Некоторые устройства измеряют область контакта и степень надавливания, например пальца на сенсорном экране, для этого есть дополнительные свойства. Эти свойства большинством устройств не поддерживаются, поэтому редко используются.

Мульти-тач

Одной из функций, которую абсолютно не поддерживают события мыши, является **мульти-тач**: возможность касаться сразу нескольких мест на телефоне или планшете или выполнять специальные жесты. События указателя позволяют обрабатывать мульти-тач с помощью свойств **pointerId** и **isPrimary**.

Вот что происходит, когда пользователь касается сенсорного экрана в одном месте, а затем в другом:

1. При касании первым пальцем: происходит событие pointerdown со свойством isPrimary=true и некоторым pointerId.
2. При касании вторым и последующими пальцами (при остающемся первом): происходит событие pointerdown со свойством isPrimary=false и уникальным pointerId для каждого касания.

Обратите внимание: pointerId присваивается не на всё устройство, а для каждого касающегося пальца. Если коснуться экрана 5 пальцами одновременно, получим 5 событий pointerdown, каждое со своими координатами и индивидуальным pointerId.
События, связанные с первым пальцем, всегда содержат свойство **isPrimary=true**.
Мы можем отслеживать несколько касающихся экрана пальцев, используя их pointerId. Когда пользователь перемещает, а затем убирает палец, получаем события pointermove и pointerup с тем же pointerId, что и при событии pointerdown.

Событие: pointercancel

Событие **pointercancel** происходит, когда текущее действие с указателем по какой-то причине прерывается, и события указателя больше не генерируются. К таким причинам можно отнести:

- Указывающее устройство было физически выключено.
- Изменилась ориентация устройства (перевернули планшет).
- Браузер решил сам обработать действие, считая его жестом мыши, масштабированием и т.п.
-

Предотвращайте действие браузера по умолчанию, чтобы избежать pointercancel. Нужно сделать две вещи:
Предотвратить запуск встроенного drag’n’drop

1. Мы можем сделать это, задав **ball.ondragstart = () => false**, как описано в статье Drag'n'Drop с событиями мыши. Это работает для событий мыши.
2. Для устройств с сенсорным экраном существуют другие действия браузера, связанные с касаниями, кроме drag’n’drop. Чтобы с ними не возникало проблем: Мы можем предотвратить их, добавив в CSS свойство **#ball { touch-action: none }.** Затем наш код начнёт корректно работать на устройствах с сенсорным экраном

Захват указателя

Захват указателя – особая возможность событий указателя. Основной метод:

elem.setPointerCapture(pointerId) – привязывает события с данным pointerId к elem. После такого вызова все события указателя с таким pointerId будут иметь elem в качестве целевого элемента (как будто произошли над elem), вне зависимости от того, где в документе они произошли. Другими словами, elem.setPointerCapture(pointerId) меняет target всех дальнейших событий с данным pointerId на elem.

Эта привязка отменяется:

- автоматически, при возникновении события pointerup или pointercancel,
- автоматически, если elem удаляется из документа,
- при вызове **elem.releasePointerCapture(pointerId).**

Захват указателя используется для упрощения операций с переносом (drag’n’drop) элементов.

Существует два связанных с захватом события:

- **gotpointercapture** срабатывает, когда элемент использует setPointerCapture для включения захвата.
- **lostpointercapture** срабатывает при освобождении от захвата: явно с помощью releasePointerCapture или автоматически, когда происходит событие pointerup/pointercancel.