**Порядок обработки событий**

События могут возникать не только по очереди, но и «пачкой» по много сразу. Возможно и такое, что во время обработки одного события возникают другие, например пока выполнялся код для onclick – посетитель нажал кнопку на клавиатуре (событие keydown).

Здесь мы разберём, как браузер обычно работает с одновременно возникающими событиями и какие есть исключения из общего правила.

**[Главный поток](http://learn.javascript.ru/events-and-timing-depth" \l "glavnyy-potok)**

В каждом окне выполняется только один *главный* поток, который занимается выполнением JavaScript, отрисовкой и работой с DOM.

Он выполняет команды последовательно, может делать только одно дело одновременно и блокируется при выводе модальных окон, таких как alert.

**Дополнительные потоки тоже есть**

Есть и другие, служебные потоки, например, для сетевых коммуникаций.

Поэтому скачивание файлов может продолжаться пока главный поток ждёт реакции на alert. Но управлять служебными потоками мы не можем.

**Web Workers**

Существует спецификация [Web Workers](http://www.w3.org/TR/workers/), которая позволяет запускать дополнительные JavaScript-процессы(workers).

Они могут обмениваться сообщениями с главным процессом, но у них свои переменные, и работают они также сами по себе.

Такие дополнительные процессы не имеют доступа к DOM, поэтому они полезны, преимущественно, при вычислениях, чтобы загрузить несколько ядер/процессоров одновременно.

**[Очередь событий](http://learn.javascript.ru/events-and-timing-depth" \l "ochered-sobytiy)**

Произошло одновременно несколько событий или во время работы одного случилось другое – как главному потоку обработать это?

Если главный поток прямо сейчас занят, то он не может срочно выйти из середины одной функции и прыгнуть в другую. А потом третью. Отладка при этом могла бы превратиться в кошмар, потому что пришлось бы разбираться с совместным состоянием нескольких функций сразу.

Поэтому используется альтернативный подход.

**Когда происходит событие, оно попадает в очередь.**

Внутри браузера непрерывно работает «главный внутренний цикл», который следит за состоянием очереди и обрабатывает события, запускает соответствующие обработчики и т.п.

**Иногда события добавляются в очередь сразу пачкой.**

Например, при клике на элементе генерируется несколько событий:

1. Сначала mousedown – нажата кнопка мыши.
2. Затем mouseup – кнопка мыши отпущена и, так как это было над одним элементом, то дополнительно генерируется click (два события сразу).

В действии:

<textarea rows="8" cols="40" id="area">Кликни меня

</textarea>

<script>

area.onmousedown = function(e) { this.value += "mousedown\n"; this.scrollTop = this.scrollHeight; };

area.onmouseup = function(e) { this.value += "mouseup\n"; this.scrollTop = this.scrollHeight; };

area.onclick = function(e) { this.value += "click\n"; this.scrollTop = this.scrollHeight; };

</script>

Таким образом, при нажатии кнопки мыши в очередь попадёт событие mousedown, а при отпускании – сразу два события: mouseup и click. Браузер обработает их строго одно за другим: mousedown → mouseup → click.

При этом каждое событие из очереди обрабатывается полностью отдельно от других.

**[Вложенные (синхронные) события](http://learn.javascript.ru/events-and-timing-depth" \l "vlozhennye-sinhronnye-sobytiya)**

Обычно возникающие события «становятся в очередь».

Но в тех случаях, когда событие инициируется не посетителем, а кодом, то оно, как правило, обрабатывается синхронно, то есть прямо сейчас.

Рассмотрим в качестве примера событие onfocus.

**[Пример: событие onfocus](http://learn.javascript.ru/events-and-timing-depth" \l "primer-sobytie-onfocus)**

Когда посетитель фокусируется на элементе, возникает событие onfocus. Обычно оно происходит, когда посетитель кликает на поле ввода, например:

<p>При фокусе на поле оно изменит значение.</p>

<input type="text" onfocus="this.value = 'Фокус!'" value="Кликни меня">

Но ту же фокусировку можно вызвать и явно, вызовом метода elem.focus():

<input type="text" id="elem" onfocus="this.value = 'Фокус!'">

<script>

// сфокусируется на input и вызовет обработчик onfocus

elem.focus();

</script>

В главе [Фокусировка: focus/blur](http://learn.javascript.ru/focus-blur) мы познакомимся с этим событием подробнее, а пока – нажмите на кнопку в примере ниже.

При этом обработчик onclick вызовет метод focus() на текстовом поле text. Код обработчика onfocus, который при этом запустится, сработает синхронно, прямо сейчас, до завершения onclick.

<input type="button" id="button" value="Нажми меня">

<input type="text" id="text" size="60">

<script>

button.onclick = function() {

text.value += ' ->в onclick ';

text.focus(); // вызов инициирует событие onfocus

text.value += ' из onclick-> ';

};

text.onfocus = function() {

text.value += ' !focus! ';

};

</script>

При клике на кнопке в примере выше будет видно, что управление вошло в onclick, затем перешло в onfocus, затем вышло из onclick.

**Исключение в IE**

Так ведут себя все браузеры, кроме IE.

В нём событие onfocus – всегда асинхронное, так что будет сначала полностью обработан клик, а потом – фокус. В остальных – фокус вызовется посередине клика. Попробуйте кликнуть в IE и в другом браузере, чтобы увидеть разницу.

**[Делаем события асинхронными через setTimeout(…,0)](http://learn.javascript.ru/events-and-timing-depth" \l "delaem-sobytiya-asinhronnymi-cherez-settimeout-0)**

А что, если мы хотим, чтобы *сначала* закончилась обработка onclick, а потом уже произошла обработка onfocusи связанные с ней действия?

Можно добиться и этого.

Один вариант – просто переместить строку text.focus() вниз кода обработчика onclick.

Если это неудобно, можно запланировать text.focus() чуть позже через setTimeout(..., 0), вот так

<input type="button" id="button" value="Нажми меня">

<input type="text" id="text" size="60">

<script>

button.onclick = function() {

text.value += ' ->в onclick ';

setTimeout(function() {

text.focus(); // сработает после onclick

}, 0);

text.value += ' из onclick-> ';

};

text.onfocus = function() {

text.value += ' !focus! ';

};

</script>

Такой вызов обеспечит фокусировку через минимальный «тик» таймера, по стандарту равный 4 мс. Обычно такая задержка не играет роли, а необходимую асинхронность мы получили.

**[Итого](http://learn.javascript.ru/events-and-timing-depth" \l "itogo)**

* JavaScript выполняется в едином потоке. Современные браузеры позволяют порождать подпроцессы [Web Workers](http://www.w3.org/TR/workers/), они выполняются параллельно и могут отправлять/принимать сообщения, но не имеют доступа к DOM.
* Обычно события становятся в очередь и обрабатываются в порядке поступления, асинхронно, независимо друг от друга.
* Синхронными являются вложенные события, инициированные из кода.
* Чтобы сделать событие гарантированно асинхронным, используется вызов через setTimeout(func, 0).

Отложенный вызов через setTimeout(func, 0) используется не только в событиях, а вообще – всегда, когда мы хотим, чтобы некая функция func сработала после того, как текущий скрипт завершится.