События в Javascript

События в DOM

События — это сигналы от браузера или другой среды исполнения JavaScript. Эти сигналы используются в DOM, чтобы уведомить JavaScript-код, что интересующие его действия произошли. События возникают в результате действий пользователя (заполнение и отправка формы, нажатие на кнопку, движение мыши, изменение размера окна) или потому, что изменилось:

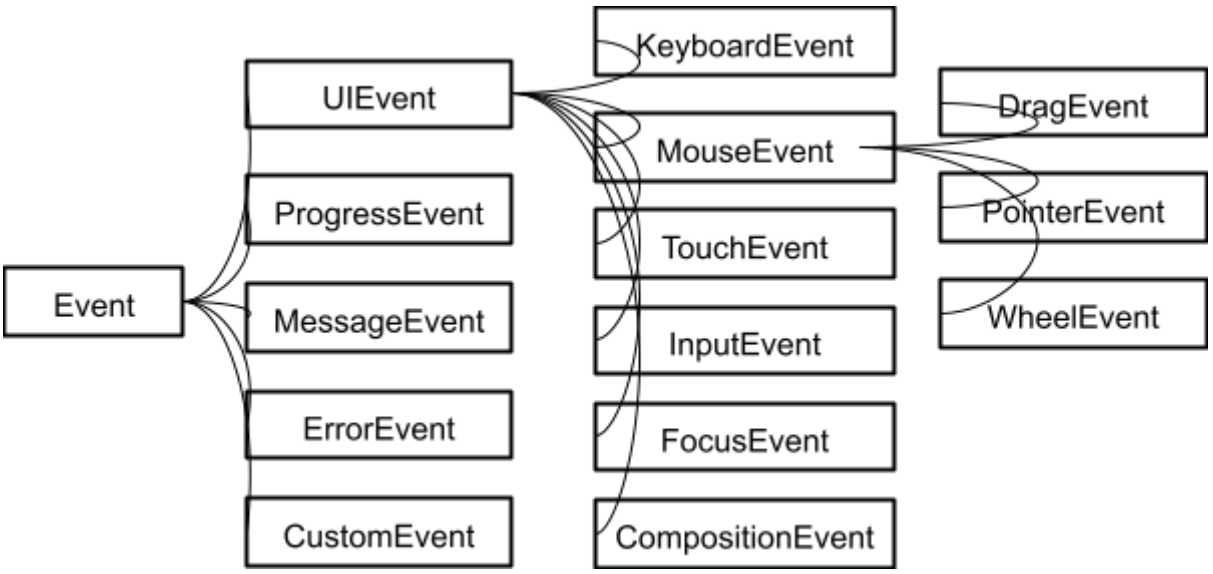
- состояние страницы (окончилась её загрузка, произошла ошибка);
- что-то в базовой среде (низкий заряд батареи, мультимедийные события из операционной системы);
- ещё что-то.

Список возможных событий в DOM очень длинен. Вот лишь некоторые примеры:

- **click** — нажатие кнопки мыши
- **touch** — касание
- **load** — загрузка
- **drag** — перетаскивание
- **change** — изменение
- **input** — ввод
- **error** — ошибка
- **resize** — изменение размера
- **contextmenu** — открытие меню
- **submit** — отправка формы

События срабатывают для любой части документа вследствие взаимодействия с ним пользователя или браузера. Такие сигналы не просто начинаются и заканчиваются в одном месте, они циркулируют по всему документу, проходя собственный жизненный цикл. Это и делает DOM-события столь гибкими и полезными. Разработчики должны понимать, как работают DOM-события, чтобы использовать их потенциал для построения удобных и функциональных интерфейсов.

Каждое событие представляет собой объект, который основан на интерфейсе Event и имеет дополнительные поля и/или функции, позволяющие получить дополнительную информацию о том, что произошло.



Чтобы JavaScript-код узнал о наступлении того или иного события, нам надо на это событие подписаться. Подписка на событие, которую мы рассмотрим ниже, заключается в добавлении слушателя (обработчика) события. Слушатель — функция или тело функции в виде строки. Рассмотрим добавление слушателей некоторых пользовательских и браузерных событий на примере:

```
<button onclick="counter++;console.log(counter)">Increment counter</button>

<script>
  let counter = 5
  window.onload = () => {
    console.log('Страница со всеми ресурсами загружена полностью.')
  }
  document.addEventListener('DOMContentLoaded', () => {
    console.log('Построение DOM-дерева завершено.')
  })
  const mouseEventListener = (event) => {
    console.log(event.type, event.clientX, event.clientY)
  }
  document.addEventListener('click', mouseEventListener)
  document.addEventListener('dblclick', {handleEvent: mouseEventListener})
  document.addEventListener('contextmenu', mouseEventListener, true)
  document.addEventListener('mouseenter', mouseEventListener, true)
  document.addEventListener('mouseleave', mouseEventListener, true)

  const throttle = (func, wait = 0) => {
    let ticking = false
```

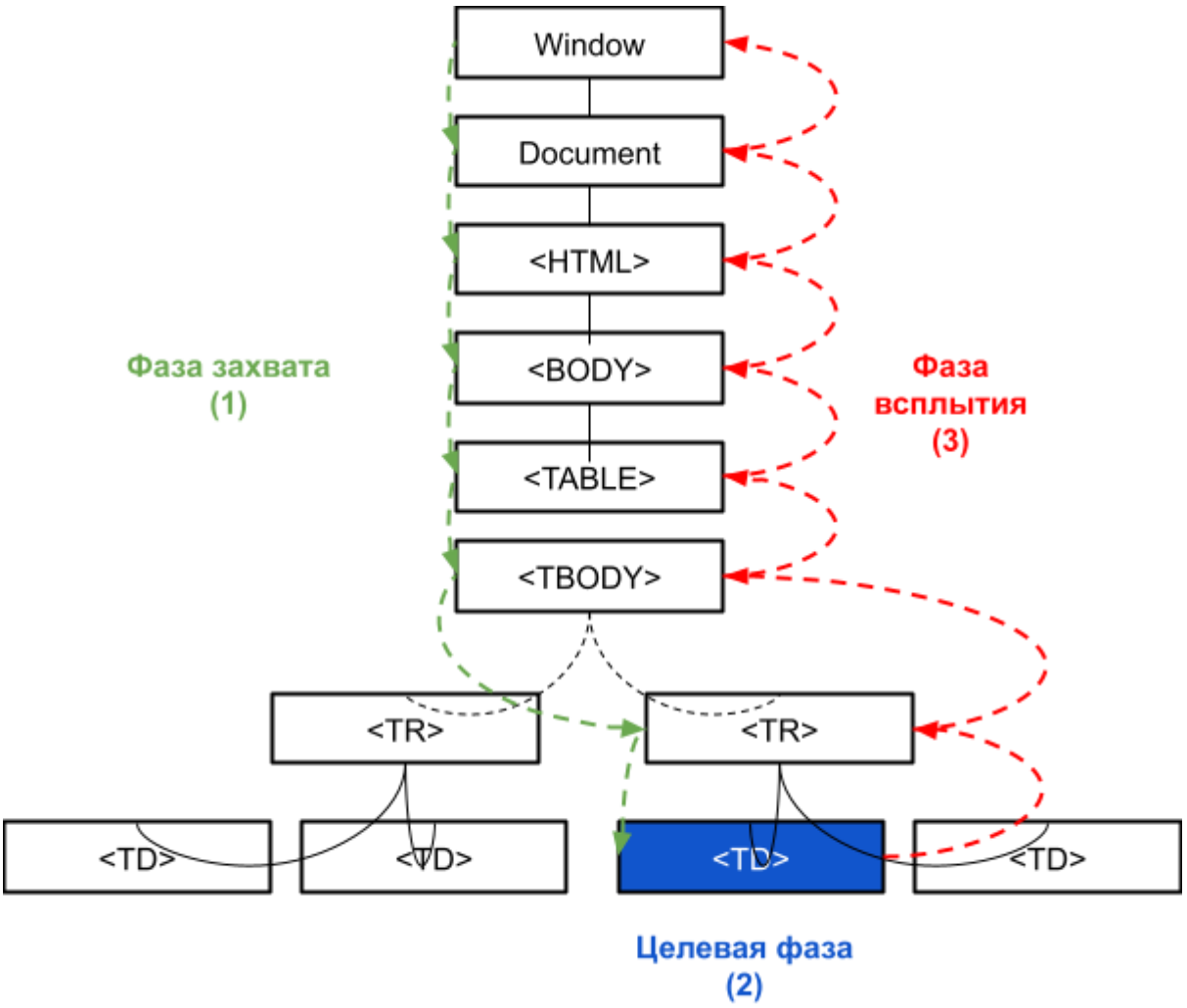
```
    return (...args) => {
      if (!ticking) {
        window.setTimeout(() => {
          func(...args)
          ticking = false
        }, wait)

        ticking = true
      }
    }
  }
  document.addEventListener('mousemove', throttle(mouseEventListener, 300),
true)
</script>
```

Представленный код выводит в консоль сообщение в момент построения DOM-дерева и во время окончания полной загрузки страницы. Он также выводит тип события и позицию курсора мыши для некоторых пользовательских событий типа MouseEvent. Далее мы подробно разберём используемые в примере методы. Поскольку событие mousemove срабатывает слишком часто, мы написали дополнительный код, чтобы сократить число вызовов обработчика, используя паттерн throttling.

Архитектура DOM-событий

Кратко разберём механизм отправки событий и то, как события распространяются по дереву DOM:



Объекты событий отправляются в цель события. Но сначала, перед отправкой, требуется определить путь распространения объекта события.

Путь распространения — это упорядоченный список текущих целей события, через которые оно проходит. Путь распространения отражает иерархическую древовидную структуру документа. Последний элемент в списке — это цель события, предыдущие элементы в списке называются предками цели (ancestors), а напрямую предшествующий элемент — родителем цели (parent).

После определения пути распространения объект события проходит через одну или несколько фаз события. Есть три фазы события: фаза захвата, целевая фаза и фаза всплытия. Фаза будет пропущена, если она не поддерживается или распространение объекта события остановилось. Например, если для bubbles-атрибута установлено значение false, фаза всплытия пропускается. А если метод **stopPropagation()** был вызван до отправки, пропускаются все фазы.

Фаза захвата (capturing phase) — объект события распространяется через его предков: от окна к родителю цели.

Целевая фаза (target phase) — объект события прибывает в цель события. Эта фаза также известна как фаза попадания в цель. Если тип события указывает, что событие не всплывает, объект события остановится после завершения этой фазы.

Фаза всплытия (bubbling phase) — объект события распространяется через предков цели в обратном порядке, начиная с родителя цели и заканчивая окном.

Важно! *Не все события в DOM всплывают. Например, события* ***focus***, ***blur***, ***load***, ***unload***, ***change***, ***reset***, ***scroll***, ***mouseenter***, ***mouseleave*** *не всплывают. Чтобы узнать, всплывает ли событие, используется логическое поле* ***Event.bubbles.***

Сравнение разных целей события

Есть несколько различных целей для рассмотрения:

Свойство	**Определено в интерфейсе**	**Описание**
event.target	DOM Event	Исходный DOM-элемент, на котором произошло событие
event.currentTarget	DOM Event	Текущий DOM-элемент, чьи подписчики в настоящее время обрабатываются. По мере того как происходит захват и всплытие событий, это значение изменяется
event.relatedTarget	DOM MouseEvent	Определяет вторичную цель события

Для некоторых пар событий мыши есть дополнительный, связанный с исходным, элемент, на котором находился курсор мыши перед событием.

Тип события	**event.target**	**event.relatedTarget**
mouseover, mouseenter, dragenter	Элемент, в который входит курсор	Элемент, из которого выходит курсор
mouseout, mouseleave, dragleave	Элемент, из которого выходит курсор	Элемент, в который входит курсор

Добавление слушателя события

Есть несколько способов добавить событию слушателей. Мы уже знакомы с атрибутом **onclick**, — давайте рассмотрим ещё некоторые способы.

Использование свойства DOM-объекта

Чтобы назначить обработчик, используется свойство DOM-элемента on<событие>.

Например, **button.onclick**:

```
<button>Кликни меня!</button>

<script>
  const button = document.querySelector('button')
  button.onclick = (event) => {
    console.log(event.target === button)
  }
</script>
```

Этот метод заменяет текущих слушателей события click, если они есть. То же самое работает для других событий и ассоциируемых с ними обработчиков, таких как **blur** (**onblur**), **keypress** (**onkeypress**) и так далее. Чтобы удалить обработчик, используется назначение **button.onclick = null**.

Эти методы имеют широкую поддержку и не требуют специального кросс-браузерного кода, однако они архаичны, неудобны и функционально ограничены.

Использование метода addEventListener

Метод **addEventListener** — это современный способ добавления слушателей событий. В отличие от описанных выше способов он:

- срабатывает на любом DOM-элементе, а не только на HTML-элементах;
- добавляет несколько обработчиков для одного события;
- предоставляет точный контроль фазы срабатывания (вызова) обработчика (захват или всплытие).

Ввиду исторических причин метод **addEventListener** поддерживает два варианта синтаксиса.

```
target.addEventListener(type, listener[, options])
target.addEventListener(type, listener[, useCapture])
```

В более старых версиях спецификации DOM третьим параметром **addEventListener** было логическое значение, указывающее, следует ли подписываться на событие на этапе захвата. Со временем стало ясно, что требуется больше вариантов. Вместо того чтобы добавлять в функцию дополнительные параметры, усложняя ситуацию при использовании необязательных значений, разработчики заменили третий параметр на объект, который содержит свойства, определяющие значения параметров для настройки обработчика событий.

Для удаления обработчика используется метод **removeEventListener** с аналогичным синтаксисом:

```
<button>Кликни меня!</button>

<script>
  const button = document.querySelector('button')
  const onceListener = () => {
    console.log('Будет вызван только один раз с включённой опцией "once"')
  }
  const onceListenerManual = (event) => {
    console.log('Будет вызван только один раз и удалён вручную через вызов
removeEventListener')
    event.target.removeEventListener('click', onceListenerManual)
  }
  button.addEventListener('click', onceListener, {once: true})
  button.addEventListener('click', onceListenerManual)
</script>
```

Как мы видим, для добавления обработчика, который автоматически удаляется после первого срабатывания, удобнее использовать опцию **once**.

Несколько одинаковых обработчиков события

Если на одном EventTarget зарегистрировано несколько одинаковых EventListener с одинаковыми параметрами, дублирующиеся обработчики игнорируются. Пример ниже содержит два одинаковых обработчика для фазы захвата, один из которых будет проигнорирован:

```
<button>Кликни меня!</button>

<script>
  const button = document.querySelector('button')
  const listener = (event) => {
    switch (event.eventPhase) {
      case Event.CAPTURING_PHASE: {
        console.log('Будет вызван во время фазы захвата')
        break
      }
      case Event.BUBBLING_PHASE: {
        console.log('Будет вызван во время фазы всплытия')
        break
      }
      case Event.AT_TARGET: {
        console.log('Будет вызван в фазе цели')
        break
      }
    }
  }

  document.addEventListener('click', listener) // Фаза всплытия (bubbling phase)
  document.addEventListener('click', listener, true) // Фаза захвата (capturing
```

```
phase)
  document.addEventListener('click', listener, {capture: true}) // Фаза захвата
(capturing phase)
  button.addEventListener('click', listener) // Фаза цели (target phase)
</script>
```

Делегирование событий

Часто возникает задача назначить один обработчик события на множество однотипных нод, например, DOM-элементов списка <li>. В этом случае вместо получения ссылок на все <li> в списке, перебора их в цикле и назначения каждому слушателя, разумно назначить только одного слушателя на родительский элемент <ul>. Эта техника называется делегированием событий и широко применяется во фронтенд-разработке.

При делегировании используется свойство event.target для доступа к целевому элементу события. Свойство **event.currentTarget** будет указывать на тот элемент, на который мы делегировали обработчик:

```
<ul>
  <li>Первый</li>
  <li>Второй</li>
  <li>Третий</li>
</ul>

<script>
  const list = document.querySelector('ul')
  const listener = (e) => {
   console.log(e.target, e.currentTarget)
  }

  list.addEventListener('click', listener)

  ['fourth', 'fifth'].forEach((text) => {
    const listItem = document.createElement('li')
    listItem.append(text)
    list.append(listItem)
  })
</script>
```

Ещё один бонус делегирования в том, что динамически добавленные после назначения обработчика элементы списка также будут реагировать на клик, и никаких дополнительных вызовов **addEventListener** для них не потребуется.