Практическая работа №13 «Размеры и прокрутка элементов/окна. Координаты. Браузерные события. Делегирования событий»

Размеры и прокрутка элементов

Существует множество JavaScript-свойств, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. Они часто требуются, когда нам нужно передвигать или позиционировать элементы с помощью JavaScript.

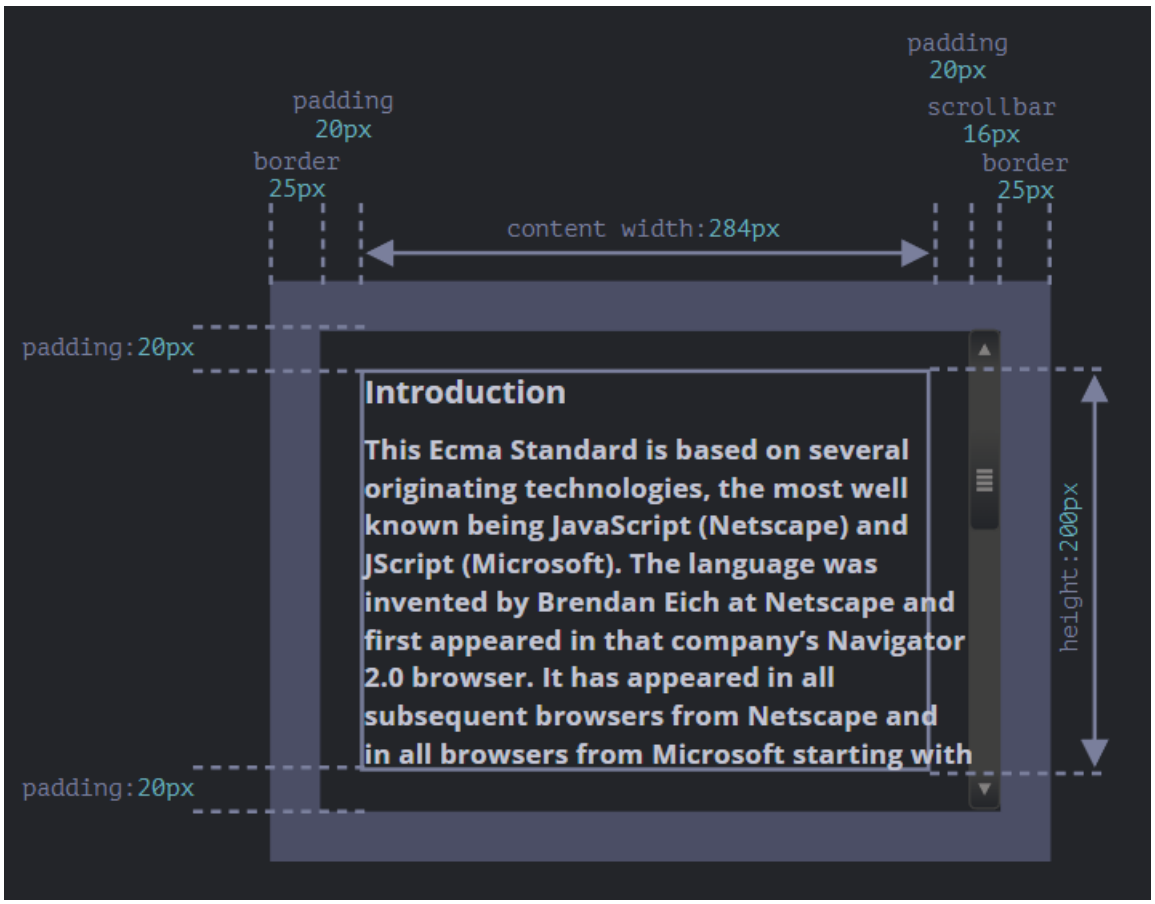
Простой пример

В качестве простого примера демонстрации свойств мы будем использовать следующий элемент:

```
<div id="example">
  ...Текст...
</div>
<style>
  #example {
    width: 300px;
    height: 200px;
    border: 25px solid #E8C48F;
    padding: 20px;
    overflow: auto;
  }
</style>
```

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств.

Результат выглядит так:


padding
20px
scrollbar
16px
padding
20px
border
25px
border
25px
content width:284px
padding:20px
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with
height:200px
padding:20px

Внимание, полоса прокрутки

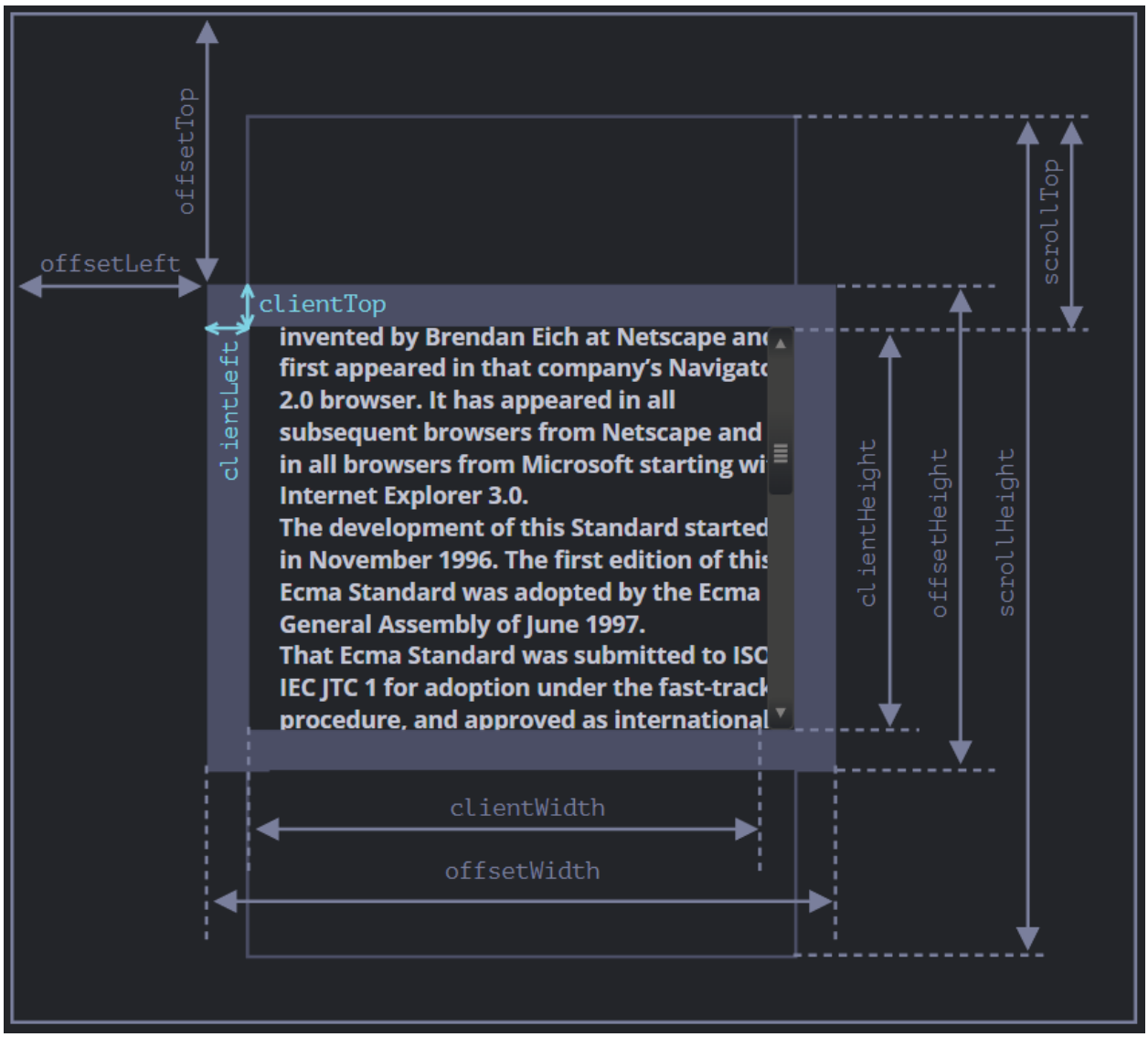
В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше). Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только 300 - 16 = 284px, и мы должны это учитывать.

Область padding-bottom (нижний внутренний отступ) может быть заполнена текстом

Нижние внутренние отступы padding-bottom изображены пустыми на наших иллюстрациях, но если элемент содержит много текста, то он будет перекрывать padding-bottom, это нормально.

Метрики

Вот общая картина с геометрическими свойствами:



Значениями свойств являются числа, подразумевается, что они в пикселях.

offsetParent, offsetLeft/Top

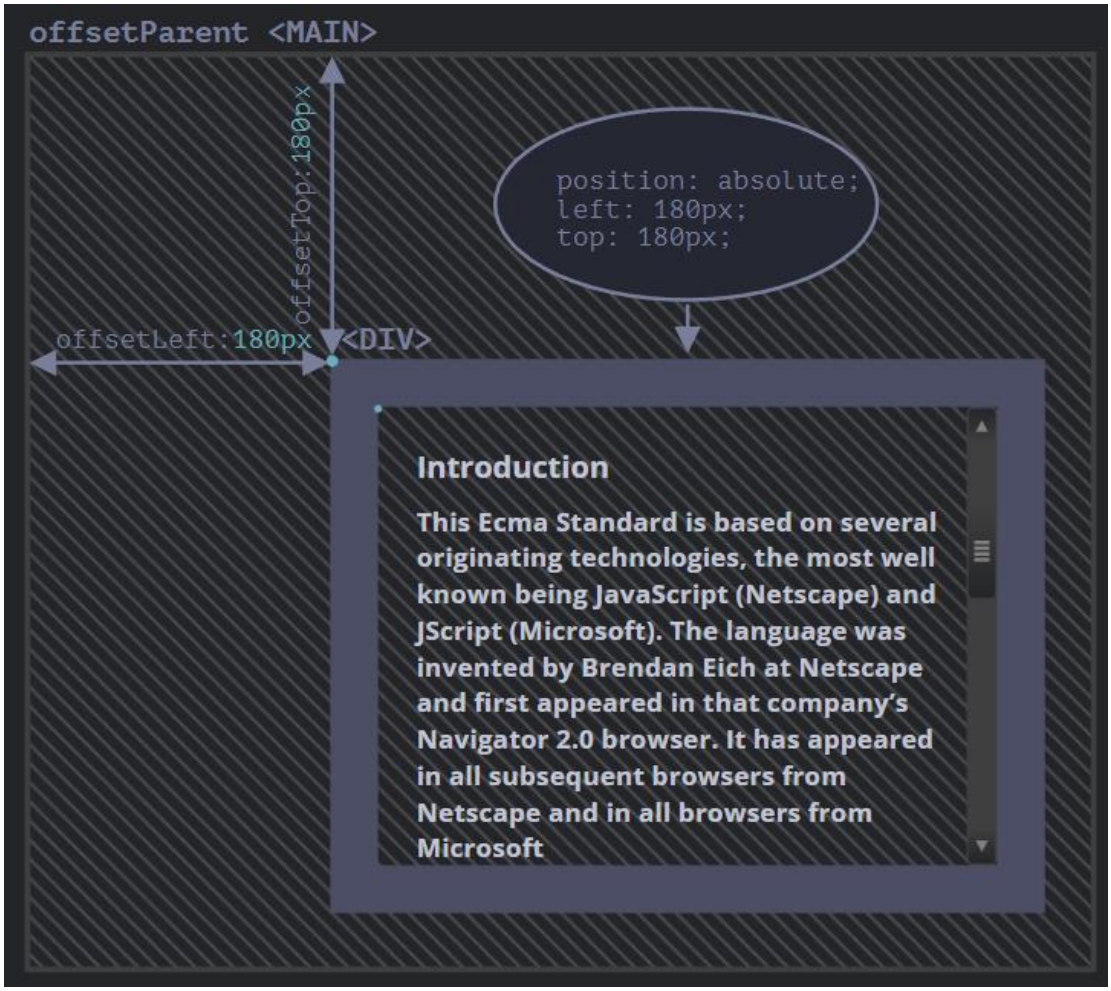
Эти свойства редко используются, но так как они являются «самыми внешними» метриками, мы начнём с них. В свойстве offsetParent находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге. То есть, ближайший предок, который удовлетворяет следующим условиям:

1. Является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky),
2. или <td>, <th>, <table>,

3. или <body>.

Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent. В примере ниже внутренний имеет элемент <main> в качестве offsetParent, а свойства offsetLeft/offsetTop являются сдвигами относительно верхнего левого угла (180):

```
<main style="position: relative" id="main">
  <article>
    <div id="example" style="position: absolute; left: 180px; top: 180px">...</div>
  </article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (обратите внимание: число, а не строка "180px")
  alert(example.offsetTop); // 180
</script>
```

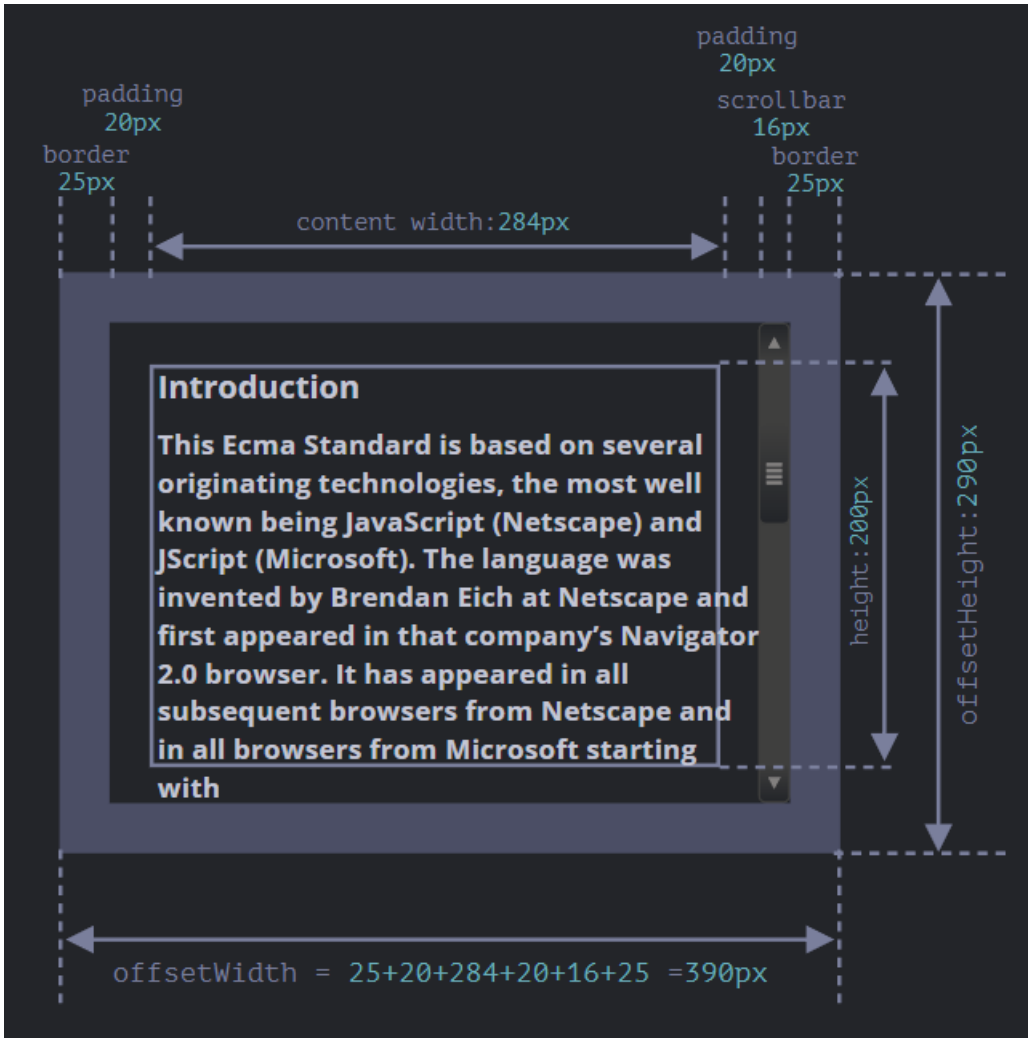


Существует несколько ситуаций, когда offsetParent равно null:

1. Для скрытых элементов (с CSS-свойством display:none или когда его нет в документе).
2. Для элементов <body> и <html>.
3. Для элементов с position:fixed.

offsetWidth/Height

Эти два свойства – самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки.



Для нашего элемента:

- offsetWidth = 390 – внешняя ширина блока, её можно получить сложением CSS-ширины (300px), внутренних отступов (2 * 20px) и рамок (2 * 25px).
- offsetHeight = 290 – внешняя высота блока.

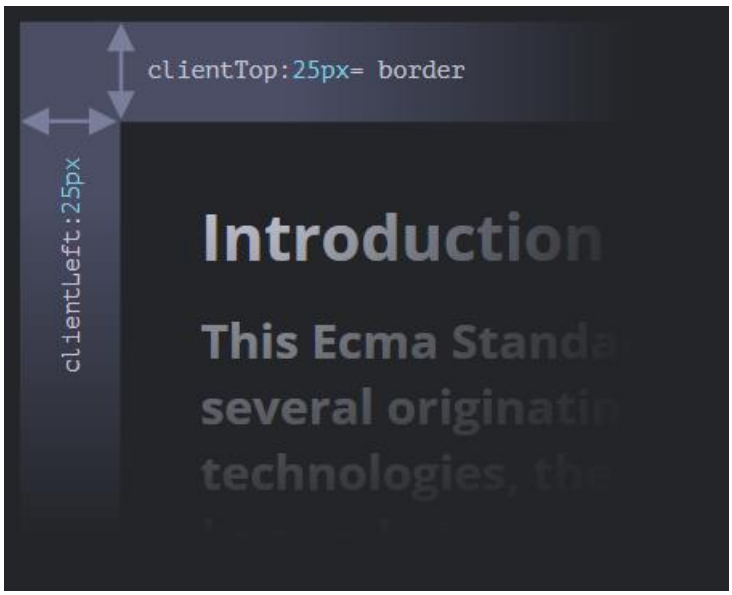
Метрики для не показываемых элементов равны нулю.

Координаты и размеры в JavaScript устанавливаются только для видимых элементов. Если элемент (или любой его родитель) имеет display:none или отсутствует в документе, то все его метрики равны нулю (или null, если это offsetParent). Например, свойство offsetParent равно null, а offsetWidth и offsetHeight равны 0, когда мы создали элемент, но ещё не вставили его в документ, или если у элемента (или у его родителя) display:none. Заметим, что функция isHidden также вернёт true для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые).

clientTop/Left

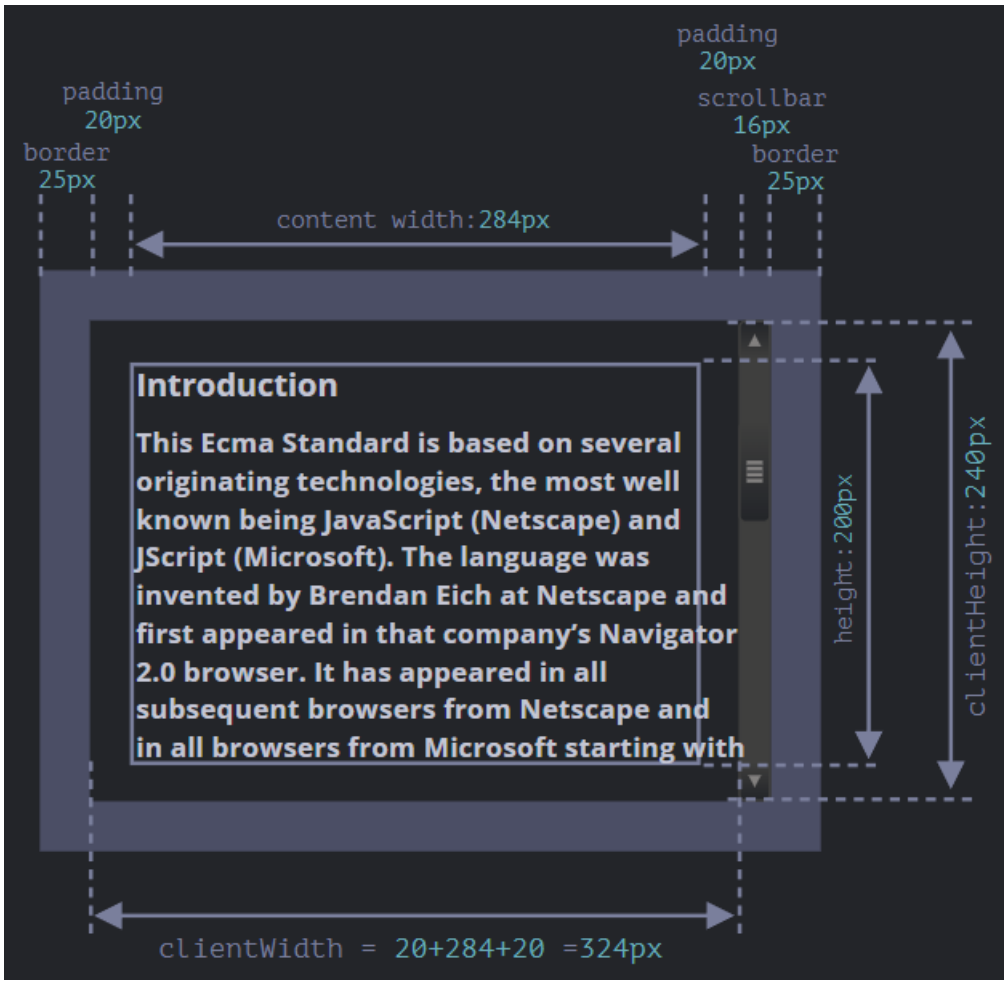
Внутри элемента у нас рамки (border). Для них есть свойства-метрики clientTop и clientLeft. В нашем примере:

- clientLeft = 25 – ширина левой рамки
- clientTop = 25 – ширина верхней рамки



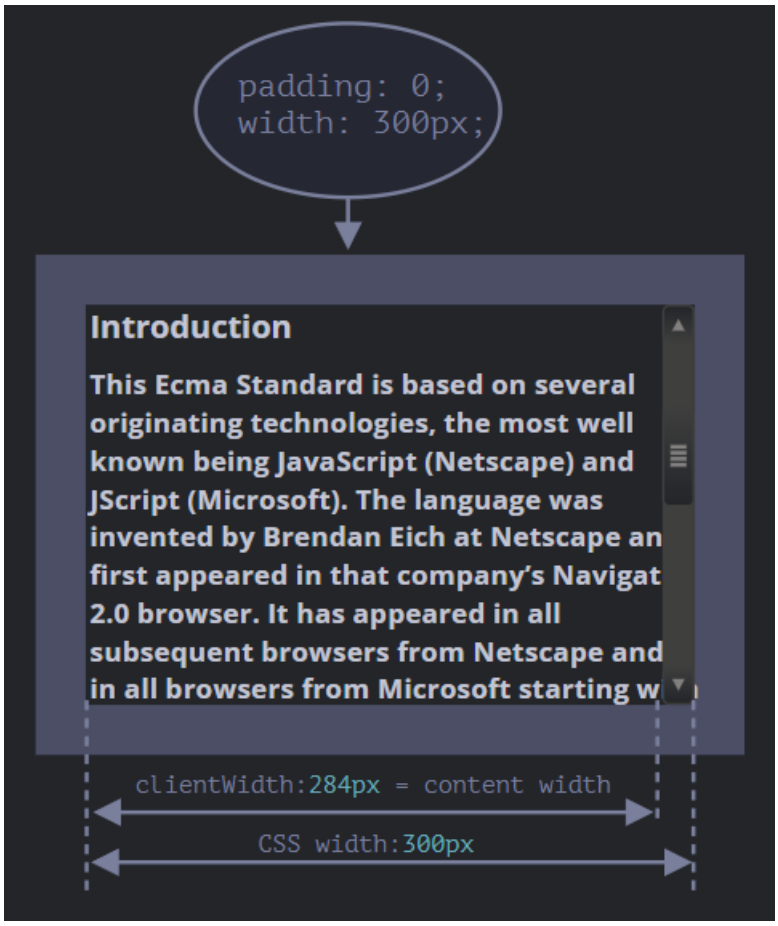
clientWidth/Height

Эти свойства – размер области внутри рамок элемента. Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки:



На рисунке выше посмотрим вначале на высоту clientHeight. Горизонтальной прокрутки нет, так что это в точности то, что внутри рамок: CSS-высота 200px плюс верхние и нижние внутренние отступы (2 * 20px), итого 240px. Теперь clientWidth – ширина содержимого здесь равна не 300px, а 284px, т.к. 16px отведено для полосы прокрутки. Таким образом: 284px плюс левый и правый отступы – всего 324px.

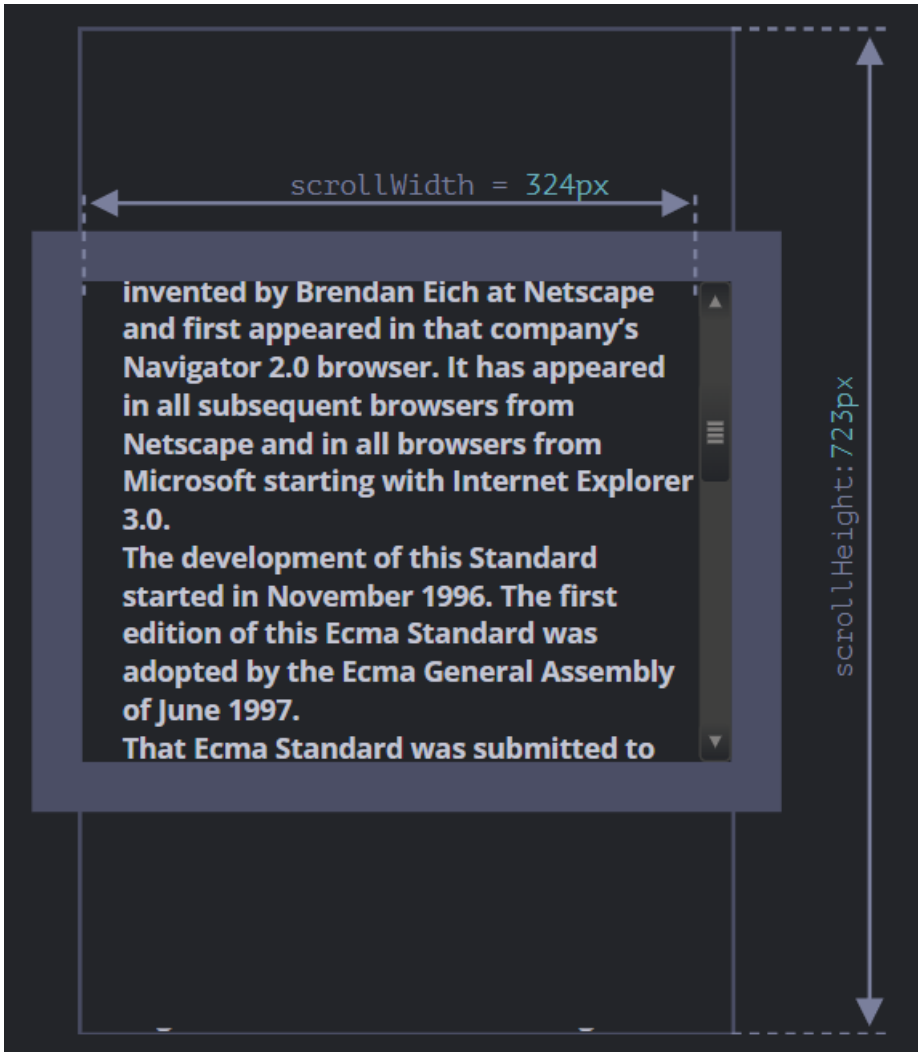
Если нет внутренних отступов padding, то clientWidth/Height в точности равны размеру области содержимого внутри рамок за вычетом полосы прокрутки (если она есть).



Поэтому в тех случаях, когда мы точно знаем, что отступов нет, можно использовать clientWidth/clientHeight для получения размеров внутренней области содержимого.

scrollWidth/Height

Эти свойства – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента.



На рисунке выше:

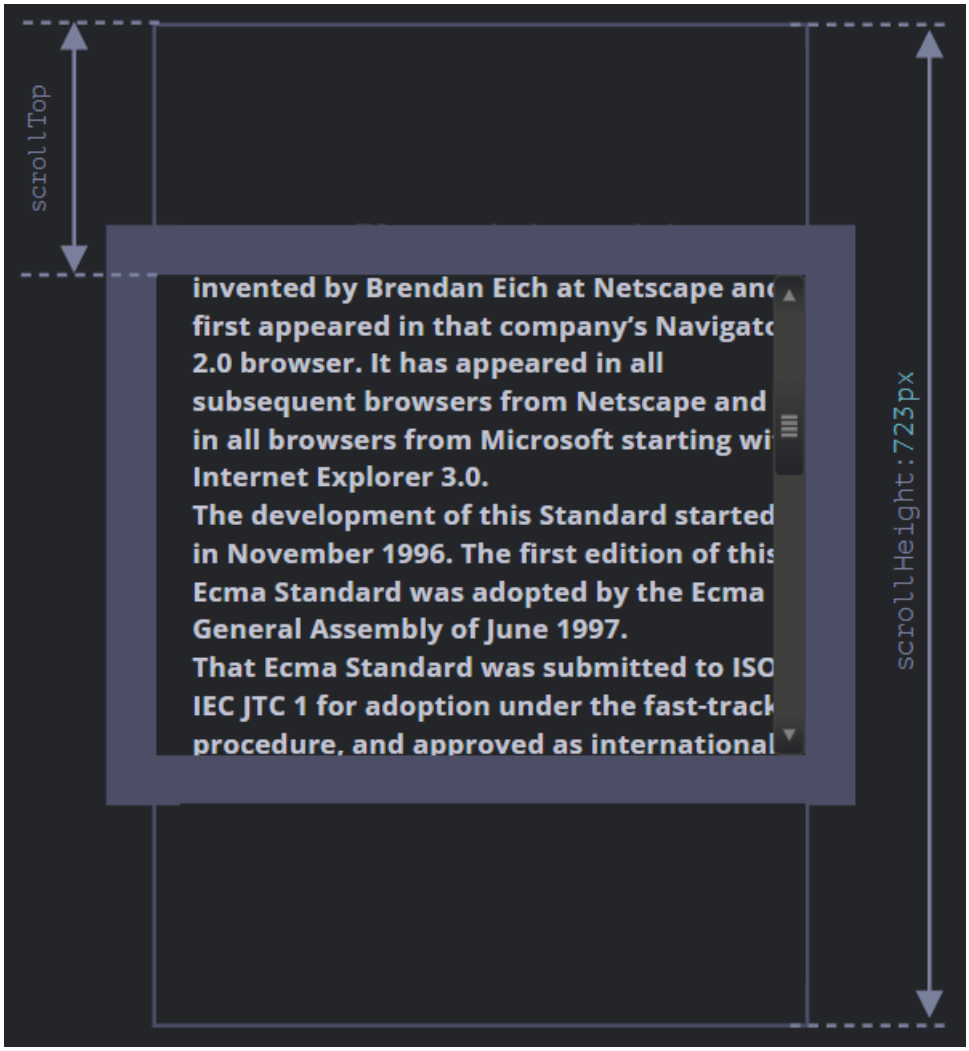
- scrollHeight = 723 – полная внутренняя высота, включая прокрученную область.
- scrollWidth = 324 – полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна clientWidth.

Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту. Таким кодом:

```
// распахнуть элемент на всю высоту
element.style.height = `${element.scrollHeight}px`;
```

scrollLeft/scrollTop

Свойства scrollLeft/scrollTop – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху. Следующая иллюстрация показывает значения scrollHeight и scrollTop для блока с вертикальной прокруткой.



Другими словами, свойство scrollTop – это «сколько уже прокручено вверх».

Свойства scrollLeft/scrollTop можно изменять

В отличие от большинства свойств, которые доступны только для чтения, значения scrollLeft/scrollTop можно изменять, и браузер выполнит прокрутку

элемента. При клике на следующий элемент будет выполняться код elem.scrollTop += 10. Поэтому он будет прокручиваться на 10px вниз.

Не стоит брать width/height из CSS

Мы рассмотрели метрики, которые есть у DOM-элементов, и которые можно использовать для получения различных высот, ширин и прочих расстояний. CSS-высоту и ширину можно извлечь, используя getComputedStyle.

Так почему бы не получать, к примеру, ширину элемента при помощи getComputedStyle. На то есть две причины:

1. Во-первых, CSS-свойства width/height зависят от другого свойства – box-sizing, которое определяет, «что такое», собственно, эти CSS-ширина и высота. Получается, что изменение box-sizing, к примеру, для более удобной вёрстки, сломает такой JavaScript.
2. Во-вторых, CSS свойства width/height могут быть равны auto, например, для инлайнового элемента:

```
<span id="elem">Привет!</span>

<script>
alert( getComputedStyle(elem).width ); // auto
</script>
```

Конечно, с точки зрения CSS width:auto – совершенно нормально, но нам-то в JavaScript нужен конкретный размер в px, который мы могли бы использовать для вычислений. Получается, что в данном случае ширина из CSS вообще бесполезна. Есть и ещё одна причина: полоса прокрутки. Бывает, без полосы прокрутки код работает прекрасно, но стоит ей появиться, как начинают проявляться баги. Так происходит потому, что полоса прокрутки «отъедает» место от области внутреннего содержимого в некоторых браузерах. Таким образом, реальная ширина содержимого *меньше* CSS-ширины. Как раз это и учитывают свойства clientWidth/clientHeight. Но с

getComputedStyle(elem).width ситуация иная. Некоторые браузеры (например, Chrome) возвращают реальную внутреннюю ширину с вычетом ширины полосы прокрутки, а некоторые (например, Firefox) – именно CSS-свойство (игнорируя полосу прокрутки). Эти кроссбраузерные отличия – ещё один повод не использовать getComputedStyle, а использовать свойства-метрики.

У элементов есть следующие геометрические свойства (метрики):

- offsetParent – ближайший CSS-позиционированный родитель или ближайший td, th, table, body.
- offsetLeft/offsetTop – позиция в пикселях верхнего левого угла относительно offsetParent.
- offsetWidth/offsetHeight – «внешняя» ширина/высота элемента, включая рамки.
- clientLeft/clientTop – расстояние от верхнего левого внешнего угла до внутренного. Для операционных систем с ориентацией слева-направо эти свойства равны ширинам левой/верхней рамки. Если язык ОС таков, что ориентация справа налево, так что вертикальная полоса прокрутки находится не справа, а слева, то clientLeft включает в своё значение её ширину.
- clientWidth/clientHeight – ширина/высота содержимого вместе с внутренними отступами padding, но без полосы прокрутки.
- scrollWidth/scrollHeight – ширины/высота содержимого, аналогично clientWidth/Height, но учитывают прокрученную, невидимую область элемента.
- scrollLeft/scrollTop – ширина/высота прокрученной сверху части элемента, считается от верхнего левого угла.

Все свойства доступны только для чтения, кроме scrollLeft/scrollTop, изменение которых заставляет браузер прокручивать элемент.

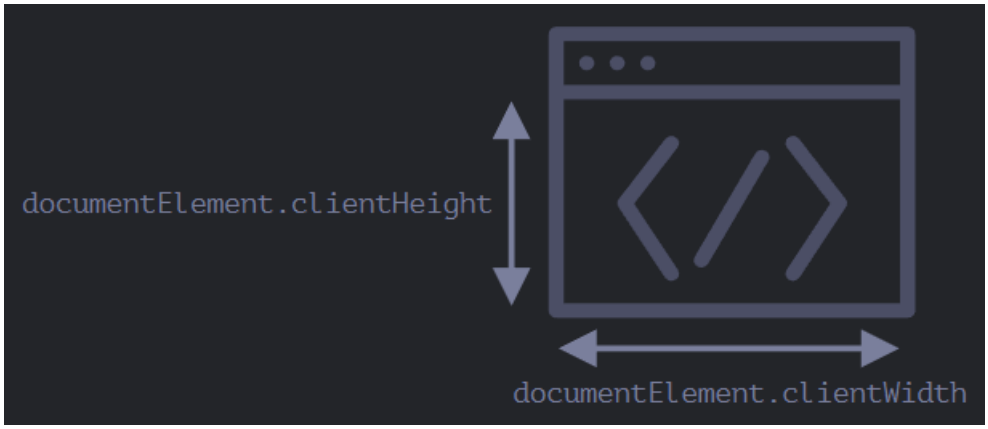
Размеры и прокрутка окна

Как узнать ширину и высоту окна браузера? Как получить полную ширину и высоту документа, включая прокрученную часть? Как прокрутить страницу с помощью JavaScript?

Для большинства таких запросов мы можем использовать корневой элемент документа document.documentElement, который соответствует тегу <html>. Однако есть дополнительные методы и особенности, которые необходимо учитывать.

Ширина/высота окна

Чтобы получить ширину/высоту окна, можно взять свойства clientWidth/clientHeight из document.documentElement:



Ширина/высота документа

Теоретически, т.к. корневым элементом документа является documentElement, и он включает в себя всё содержимое, мы можем получить полный размер документа как documentElement.scrollWidth/scrollHeight. Но именно на этом элементе, для страницы в целом, эти свойства работают не так, как предполагается. В Chrome/Safari/Opera, если нет прокрутки, то documentElement.scrollHeight может быть даже меньше, чем documentElement.clientHeight! С точки зрения элемента это невозможная

ситуация. Чтобы надёжно получить полную высоту документа, нам следует взять максимальное из этих свойств:

```
let scrollHeight = Math.max(
  document.body.scrollHeight, document.documentElement.scrollHeight,
  document.body.offsetHeight, document.documentElement.offsetHeight,
  document.body.clientHeight, document.documentElement.clientHeight
);
alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);
```

Получение текущей прокрутки

Текущую прокрутку можно прочитать из свойств window.pageXOffset/pageYOffset:

```
alert('Текущая прокрутка сверху: ' + window.pageYOffset);
alert('Текущая прокрутка слева: ' + window.pageXOffset);
```

Эти свойства доступны только для чтения.

Прокрутка: scrollTo, scrollBy, scrollIntoView

Важно:

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен. Например, если мы попытаемся прокрутить страницу из скрипта, подключенного в <head>, это не сработает.

Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft. Мы можем сделать то же самое для страницы в целом, используя document.documentElement.scrollTop/Left (кроме основанных на старом WebKit (Safari), где, как сказано выше, document.body.scrollTop/Left). Есть и другие способы, в которых подобных несовместимостей нет: специальные методы window.scrollBy(x,y) и window.scrollTo(pageX,pageY).

- Метод scrollBy(x,y) прокручивает страницу *относительно её текущего положения*. Например, scrollBy(0,10) прокручивает страницу на 10px вниз.
- Метод scrollTo(pageX,pageY) прокручивает страницу *на абсолютные координаты* (pageX,pageY). То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить scrollLeft/scrollTop. Для прокрутки в самое начало мы можем использовать scrollTo(0,0).

Эти методы одинаково работают для всех браузеров.

scrollIntoView

Для полноты картины давайте рассмотрим ещё один метод: elem.scrollIntoView(top). Вызов elem.scrollIntoView(top) прокручивает страницу, чтобы elem оказался вверху. У него есть один аргумент:

- если top=true (по умолчанию), то страница будет прокручена, чтобы elem появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.
- если top=false, то страница будет прокручена, чтобы elem появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Запретить прокрутку

Иногда нам нужно сделать документ «непрокручиваемым». Например, при показе большого диалогового окна над документом – чтобы посетитель мог прокручивать это окно, но не документ.

Чтобы запретить прокрутку страницы, достаточно установить document.body.style.overflow = "hidden". Аналогичным образом мы можем «заморозить» прокрутку для других элементов, а не только

для document.body. Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место.

Это выглядит немного странно, но это можно обойти, если сравнить clientWidth до и после остановки, и если clientWidth увеличится (значит полоса прокрутки исчезла), то добавить padding в document.body вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

Размеры:

- Ширина/высота видимой части документа (ширина/высота области содержимого): document.documentElement.clientWidth/Height
- Ширина/высота всего документа со всей прокручиваемой областью страницы:
- let scrollHeight = Math.max(
- document.body.scrollHeight, document.documentElement.scrollHeight,
- document.body.offsetHeight, document.documentElement.offsetHeight,
- document.body.clientHeight, document.documentElement.clientHeight

  );

Прокрутка:

- Прокрутку окна можно получить так: window.pageYOffset/pageXOffset.
- Изменить текущую прокрутку:
    - window.scrollTo(pageX,pageY) – абсолютные координаты,
    - window.scrollBy(x,y) – прокрутка относительно текущего места,

- elem.scrollIntoView(top) – прокрутить страницу так, чтобы сделать elem видимым (выровнять относительно верхней/нижней части окна).

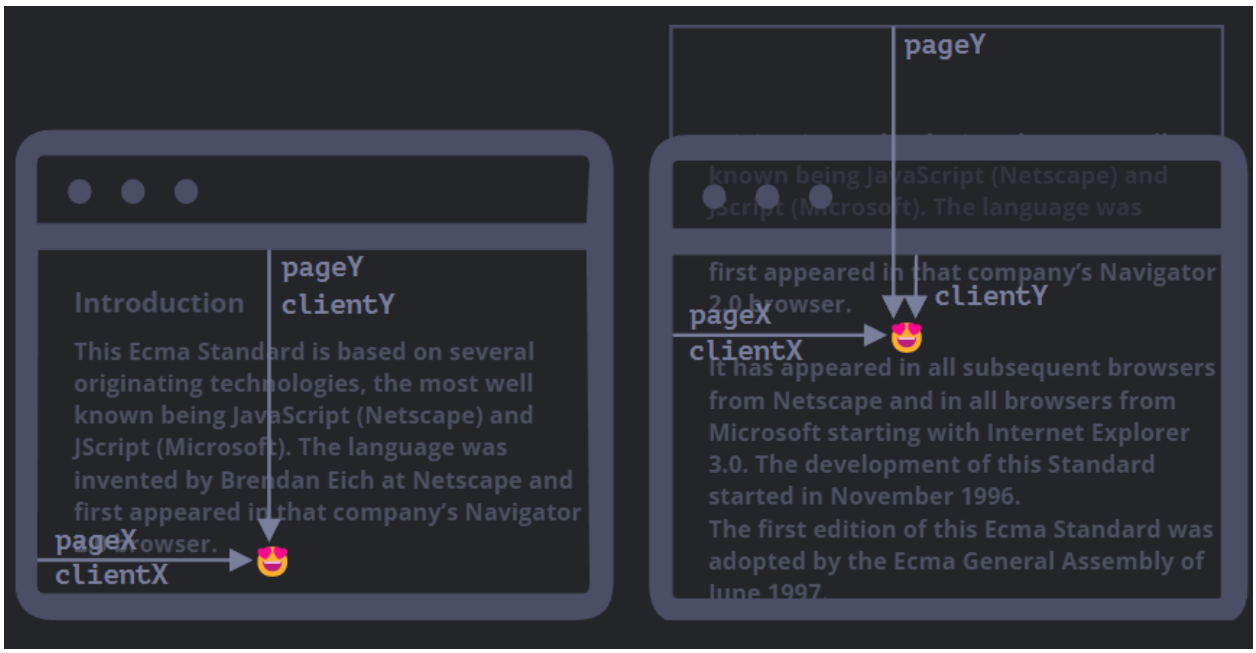
Координаты

Чтобы передвигать элементы по экрану, нам следует познакомиться с системами координат. Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

1. **Относительно окна браузера** – как position:fixed, отсчёт идёт от верхнего левого угла окна.
   - мы будем обозначать эти координаты как clientX/clientY.
2. **Относительно документа** – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа.
   - мы будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На приведённой картинке взята точка в документе и показаны её координаты до прокрутки (слева) и после (справа):



При прокрутке документа:

- pageY – координата точки относительно документа осталась без изменений, так как отсчёт по-прежнему ведётся от верхней границы документа (сейчас она прокручена наверх).
- clientY – координата точки относительно окна изменилась (стрелка на рисунке стала короче), так как точка стала ближе к верхней границе окна.

Координаты относительно окна: getBoundingClientRect

Метод elem.getBoundingClientRect() возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса DOMRect.

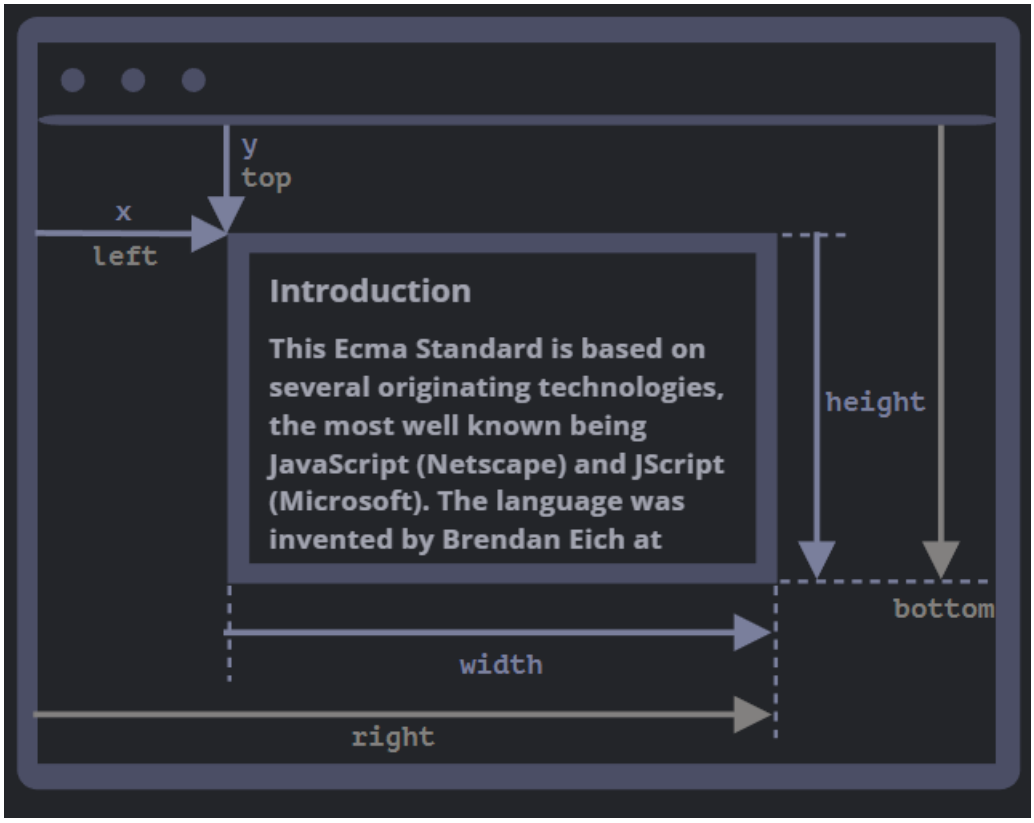
Основные свойства объекта типа DOMRect:

- x/y – X/Y-координаты начала прямоугольника относительно окна,
- width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

- top/bottom – Y-координата верхней/нижней границы прямоугольника,
- left/right – X-координата левой/правой границы прямоугольника.

Вот картинка с результатами вызова elem.getBoundingClientRect():



Как вы видите, x/y и width/height уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе:

- left = x
- top = y
- right = x + width
- bottom = y + height

Заметим:

- Координаты могут считаться с десятичной частью, например 10.5. Это нормально, ведь браузер использует дроби в своих внутренних вычислениях. Мы не обязаны округлять значения при установке style.left/top.

- Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент elem ушёл вверх за пределы окна, то вызов elem.getBoundingClientRect().top вернёт отрицательное значение.

elementFromPoint(x, y)

Вызов document.elementFromPoint(x, y) возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y). Синтаксис:

```
let elem = document.elementFromPoint(x, y);
```

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента,

```
let centerX = document.documentElement.clientWidth / 2;
let centerY = document.documentElement.clientHeight / 2;

let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";
alert(elem.tagName);

```

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

Для координат за пределами окна метод elementFromPoint возвращает null

Метод document.elementFromPoint(x,y) работает, только если координаты (x,y) относятся к видимой части содержимого окна. Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается null.

Координаты относительно документа

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна. В CSS координаты относительно окна браузера соответствуют свойству position:fixed, а координаты относительно документа –

свойству position:absolute на самом верхнем уровне вложенности. Мы можем воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но мы можем написать его сами. Две системы координат связаны следующими формулами:

- pageY = clientY + высота вертикально прокрученной части документа.
- pageX = clientX + ширина горизонтально прокрученной части документа.

Функция getCoords(elem) берёт координаты в контексте окна с помощью elem.getBoundingClientRect() и добавляет к ним значение соответствующей прокрутки:

```
// получаем координаты элемента в контексте документа
function getCoords(elem) {
  let box = elem.getBoundingClientRect();

  return {
    top: box.top + window.pageYOffset,
    right: box.right + window.pageXOffset,
    bottom: box.bottom + window.pageYOffset,
    left: box.left + window.pageXOffset
  };
}
```

Если бы в примере выше мы использовали её вместе с position:absolute, то при прокрутке сообщение оставалось бы рядом с элементом. Модифицированная функция createMessageUnder:

```
function createMessageUnder(elem, html) {
  let message = document.createElement('div');
  message.style.cssText = "position:absolute; color: red";

  let coords = getCoords(elem);

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;

  return message;
}
```

Любая точка на странице имеет координаты:

1. Относительно окна браузера – elem.getBoundingClientRect().
2. Относительно документа – elem.getBoundingClientRect() плюс текущая прокрутка страницы.

Координаты в контексте окна подходят для использования с position:fixed, а координаты относительно документа – для использования с position:absolute.

Каждая из систем координат имеет свои преимущества и недостатки. Иногда будет лучше применить одну, а иногда – другую, как это и происходит с позиционированием в CSS, где мы выбираем между absolute и fixed.

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.

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.

Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>. Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick. Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать. Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Следующий пример по клику запускает функцию countRabbits():

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK… Но, как правило, атрибуты пишут в нижнем регистре: onclick.

Частые ошибки

Если вы только начинаете работать с событиями, обратите внимание на следующие моменты.

Функция должна быть присвоена как sayThanks, а не sayThanks().

```
// правильно
button.onclick = sayThanks;

// неправильно
button.onclick = sayThanks();

```

Если добавить скобки, то sayThanks() – это уже вызов функции, результат которого (равный undefined, так как функция ничего не возвращает) будет присвоен onclick. Так что это не будет работать.

Используйте именно функции, а не строки.

Назначение обработчика строкой elem.onclick = "alert(1)" также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не используйте setAttribute для обработчиков.

Такой вызов работать не будет:

```
// при нажатии на body будут ошибки,
// атрибуты всегда строки, и функция станет строкой
document.body.setAttribute('onclick', function() { alert(1) });

```

Регистр DOM-свойства имеет значение.

Используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие. Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение. Мы хотим назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // заменит предыдущий обработчик

```

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка. Синтаксис добавления обработчика:

```
element.addEventListener(event, handler, [options]);
```

event

Имя события, например "click".

handler

Ссылка на функцию-обработчик.

options

Дополнительный объект со свойствами:

- once: если true, тогда обработчик будет автоматически удалён после выполнения.
- capture: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}.
- passive: если true, то указывает, что обработчик никогда не вызовет preventDefault().

Для удаления обработчика следует использовать removeEventListener:

```
element.removeEventListener(event, handler, [options]);
```

Удаление требует именно ту же функцию

Для удаления нужно передать именно ту функцию-обработчик, которая была назначена. Обратим внимание – если функцию обработчик не сохранить где-либо, мы не сможем её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener. Метод addEventListener позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

```
<input id="elem" type="button" value="Нажми меня"/>

<script>
  function handler1() {
    alert('Спасибо!');
  };

  function handler2() {
    alert('Спасибо ещё раз!');
  }

  elem.onclick = () => alert("Привет");
  elem.addEventListener("click", handler1); // Спасибо!
  elem.addEventListener("click", handler2); // Спасибо ещё раз!
</script>
```

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство, и через addEventListener. Однако, во избежание путаницы, рекомендуется выбрать один способ.

Обработчики некоторых событий можно назначать только через addEventListener

Существуют события, которые нельзя назначить через DOM-свойство, но можно через addEventListener. Например, таково событие DOMContentLoaded, которое срабатывает, когда завершена загрузка и построение DOM документа.

```
document.onDOMContentLoaded = function() {
  alert("DOM построен"); // не будет работать
};
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM построен"); // а вот так сработает
});
```

Так что addEventListener более универсален. Хотя заметим, что таких событий меньшинство, это скорее исключение, чем правило.

Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт *объект события*, записывает в него детали и передаёт его в качестве аргумента функции-обработчику. Пример ниже демонстрирует получение координат мыши из объекта события:

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

<script>
  elem.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert("Координаты: " + event.clientX + ":" + event.clientY);
  };
</script>

```

Некоторые свойства объекта event:

event.type

Тип события, в данном случае "click".

event.currentTarget

Элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.

event.clientX / event.clientY

Координаты курсора в момент клика относительно окна, для событий мыши.

Есть также и ряд других свойств, в зависимости от типа событий.

Объект события доступен и в HTML

При назначении обработчика в HTML, тоже можно использовать объект event, вот так:

```
<input type="button" onclick="alert(event.type)" value="Тип события">
```

Это возможно потому, что когда браузер из атрибута создаёт функцию-обработчик, то она выглядит так: function(event) { alert(event.type) }. То есть, её первый аргумент называется "event", а тело взято из атрибута.

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. 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

Этот обработчик для сработает, если вы кликните по любому из вложенных тегов, будь то или <code>:

```
<div onclick="alert('Обработчик!')">
  <em>Если вы кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>
</div>
```

Всплытие

Принцип всплытия очень простой.

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

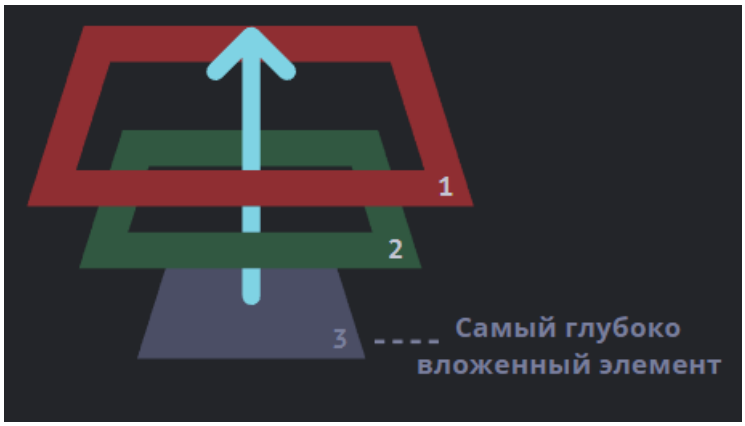
Например, есть 3 вложенных элемента FORM > DIV > P с обработчиком на каждом:

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

Клик по внутреннему <p> вызовет обработчик onclick:

1. Сначала на самом <p>.

2. Потом на внешнем .
3. Затем на внешнем <form>.
4. И так далее вверх по цепочке до самого document.



Поэтому если кликнуть на <p>, то мы увидим три оповещения: p → div → form.

Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей.

Почти **все события всплывают.**

Ключевое слово в этой фразе – «почти».

Например, событие focus не всплывает. В дальнейшем мы увидим и другие примеры. Однако, стоит понимать, что это скорее исключение, чем правило, всё-таки большинство событий всплывают.

Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути. Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие. Для этого нужно вызвать метод event.stopPropagation().

Например, здесь при клике на кнопку <button> обработчик body.onclick не сработает:

```
<body onclick="alert(`сюда всплытие не дойдёт`)">
  <button onclick="event.stopPropagation()">Кликни меня</button>
</body>

```

event.stopImmediatePropagation()

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. То есть, event.stopPropagation() препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны. Для того, чтобы полностью остановить обработку, существует метод event.stopImmediatePropagation(). Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости!

Всплытие – это удобно. Не прекращайте его без явной нужды, очевидной и архитектурно прозрачной. Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами.

Например:

1. Мы делаем вложенное меню. Каждое подменю обрабатывает клики на своих элементах и делает для них stopPropagation, чтобы не срабатывало внешнее меню.
2. Позже мы решили отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще у нас кликают люди. Некоторые системы аналитики так делают. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.

3. Наша аналитика не будет работать над областью, где клики прекращаются stopPropagation. Увы, получилась «мёртвая зона».

Зачастую нет никакой необходимости прекращать всплытие. Задача, которая, казалось бы, требует этого, может быть решена иначе. Например, с помощью создания своего уникального события, о том, как это делать, мы поговорим позже. Также мы можем записывать какую-то служебную информацию в объект event в одном обработчике, а читать в другом, таким образом мы можем сообщить обработчикам на родительских элементах информацию о том, что событие уже было как-то обработано.

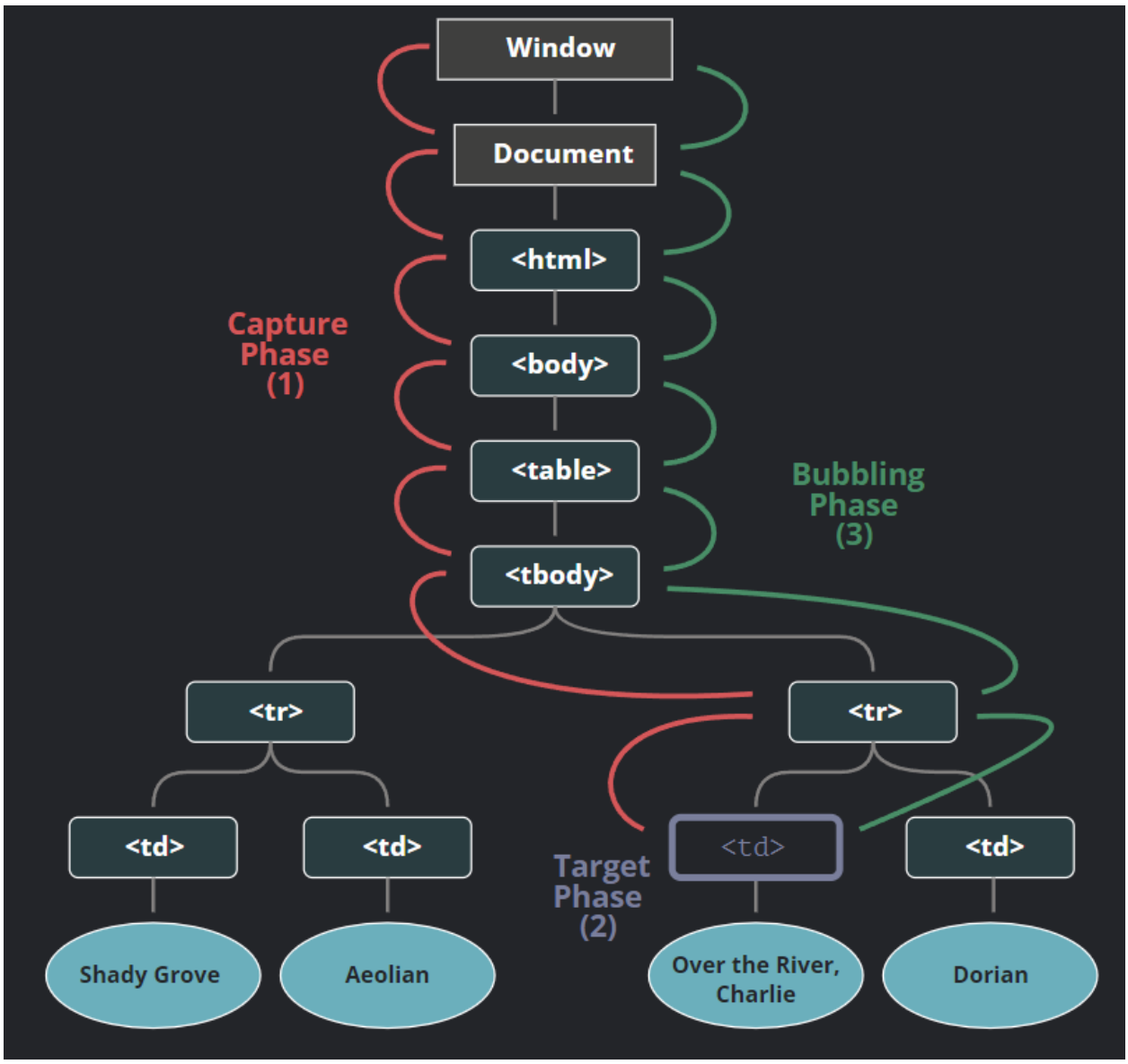
Погружение

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт DOM Events описывает 3 фазы прохода события:

1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке <td>, расположенной внутри таблицы:



То есть при клике на <td> событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт наверх (всплытие), вызывая по пути обработчики.

Обработчики, добавленные через on<event>-свойство или через HTML-атрибуты, или через addEventListener(event, handler) с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах.

Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент capture вот так:

```
elem.addEventListener(..., {capture: true})
// или просто "true", как сокращение для {capture: true}
elem.addEventListener(..., true)
```

Существуют два варианта значений опции capture:

- Если аргумент false (по умолчанию), то событие будет поймано при всплытии.
- Если аргумент true, то событие будет перехвачено при погружении.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение.

Давайте посмотрим и всплытие, и погружение в действии:

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form>FORM
  <div>DIV
    <p>P</p>
  </div>
</form>

<script>
  for(let elem of document.querySelectorAll('*')) {
    elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);
    elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));
  }
</script>
```

Здесь обработчики навешиваются на *каждый* элемент в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события. Если вы кликните по <p>, то последовательность следующая:

1. HTML → BODY → FORM → DIV (фаза погружения, первый обработчик)
2. P (фаза цели, срабатывают обработчики, установленные и на погружение, и на всплытие, так что выведется два раза)
3. DIV → FORM → BODY → HTML (фаза всплытия, второй обработчик)

Существует свойство event.eventPhase, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, мы обычно и так знаем об этом в обработчике.

Чтобы убрать обработчик removeEventListener, нужна та же фаза

Если мы добавили обработчик вот так addEventListener(..., true), то мы должны передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

На каждой фазе разные обработчики на одном элементе срабатывают в порядке назначения

Если у нас несколько обработчиков одного события, назначенных addEventListener на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

```
elem.addEventListener("click", e => alert(1)); // всегда сработает перед следующим
elem.addEventListener("click", e => alert(2));
```

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.

Делегирование событий

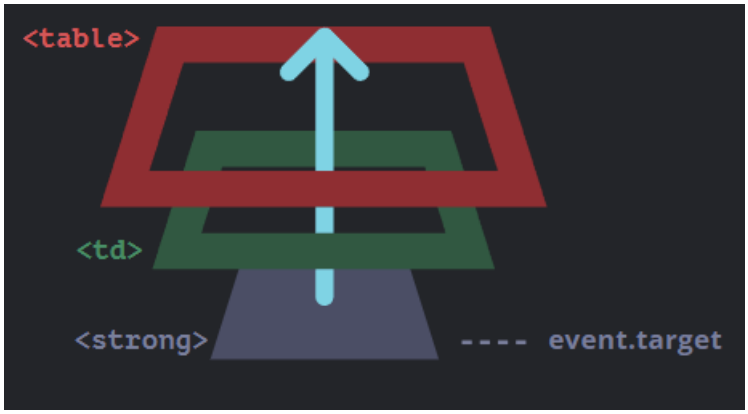
Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – *делегирование*. Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка. Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

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



Внутри обработчика table.onclick мы должны по event.target разобраться, 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>.

Применение делегирования: действия в разметке

Есть и другие применения делегирования. Например, нам нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами save, load, search. Мы можем добавить один обработчик для всего меню и атрибуты data-action для каждой кнопки в соответствии с методами, которые они вызывают:

```
1  <button data-action="save">Нажмите, чтобы Сохранить</button>
```

Обработчик считывает содержимое атрибута и выполняет метод. Взгляните на рабочий пример:

```html
<div id="menu">
  <button data-action="save">Сохранить</button>
  <button data-action="load">Загрузить</button>
  <button data-action="search">Поиск</button>
</div>

<script>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('сохраняю');
    }

    load() {
      alert('загружаю');
    }

    search() {
      alert('ищу');
    }

    onClick(event) {
      let action = event.target.dataset.action;
      if (action) {
        this[action]();
      }
    }
  }

  new Menu(menu);
</script>
```

Обратите внимание, что метод this.onClick в строке, отмеченной звёздочкой (*), привязывается к контексту текущего объекта this. Это важно, т.к. иначе this внутри него будет ссылаться на DOM-элемент (elem), а не на объект Menu, и this[action] будет не тем, что нам нужно.

Так что же даёт нам здесь делегирование?

- Не нужно писать код, чтобы присвоить обработчик каждой кнопке. Достаточно просто создать один метод и поместить его в разметку.

- Структура HTML становится по-настоящему гибкой. Мы можем добавлять/удалять кнопки в любое время.

Мы также можем использовать классы .action-save, .action-load, но подход с использованием атрибутов data-action является более семантичным. Их можно использовать и для стилизации в правилах CSS.

Приём проектирования «поведения»

Делегирование событий можно использовать для добавления элементам «поведения» (behavior), *декларативно* задавая хитрые обработчики установкой специальных HTML-атрибутов и классов.

Приём проектирования «поведение» состоит из двух частей:

1. Элементу ставится пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования ставится обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут, производит соответствующее действие.

Поведение: «Счётчик»

Например, здесь HTML-атрибут data-counter добавляет кнопкам поведение: «увеличить значение при клике»:

```
Счётчик: <input type="button" value="1" data-counter>
Ещё счётчик: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {

    if (event.target.dataset.counter != undefined) { // если есть атрибут...
      event.target.value++;
    }

  });
</script>
```

Если нажать на кнопку – значение увеличится. Конечно, нам важны не счётчики, а общий подход, который здесь продемонстрирован. Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования мы фактически добавили новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Всегда используйте метод addEventListener для обработчиков на уровне документа

Когда мы устанавливаем обработчик событий на объект document, мы всегда должны использовать метод addEventListener, а не document.on<событие>, т.к. в случае последнего могут возникать конфликты: новые обработчики будут перезаписывать уже существующие. Для реального проекта совершенно нормально иметь много обработчиков на элементе document, установленных из разных частей кода.

Поведение: «Переключатель» (Toggler)

Ещё один пример поведения. Сделаем так, что при клике на элемент с атрибутом data-toggle-id будет скрываться/показываться элемент с заданным id:

```html
<button data-toggle-id="subscribe-mail">
  Показать форму подписки
</button>

<form id="subscribe-mail" hidden>
  Ваша почта: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>
```

Ещё раз подчеркнём, что мы сделали. Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, даже не надо знать JavaScript, можно просто написать атрибут data-toggle-id. Это бывает очень удобно – не нужно писать JavaScript-код для каждого элемента, который должен так себя вести. Просто используем поведение. Обработчики на уровне документа сделают это возможным для элемента в любом месте страницы. Мы можем комбинировать несколько вариантов поведения на одном элементе. Шаблон «поведение» может служить альтернативой для фрагментов JS-кода в вёрстке.

Делегирование событий – это здорово! Пожалуй, это один из самых полезных приёмов для работы с DOM. Он часто используется, если есть много элементов, обработка которых очень схожа, но не только для этого.

Алгоритм:

1. Вешаем обработчик на контейнер.
2. В обработчике проверяем исходный элемент event.target.
3. Если событие произошло внутри нужного нам элемента, то обрабатываем его.

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.

Конечно, у делегирования событий есть свои ограничения:

- Во-первых, событие должно всплывать. Некоторые события этого не делают. Также, низкоуровневые обработчики не должны вызывать event.stopPropagation().
- Во-вторых, делегирование создаёт дополнительную нагрузку на браузер, ведь обработчик запускается, когда событие происходит в любом месте контейнера, не обязательно на элементах, которые нам интересны. Но обычно эта нагрузка настолько пустяковая, что её даже не стоит принимать во внимание.

Задачи к практической работе №13

1. Добавьте на сайт элемент, в котором по центру будет позиционироваться картинка. И элемент, и картинка должны позиционироваться в центре экрана. Картинка должна позиционироваться за счёт JavaScript, а не CSS. Код должен работать с любым размером картинки (10, 20, 30 пикселей) и любым размером поля без привязки к исходным значениям. Сделайте вывод координат того места куда кликает пользователь.

2. Для уведомлений, созданных ранее, при помощи JavaScript (с применением делегирования событий. Должен быть лишь один обработчик на элементе-контейнере для всего) для каждого сообщения добавьте в верхний правый угол кнопку закрытия.

3. Создать эффект параллакса на веб-странице, который будет реагировать на прокрутку страницы. Для достижения этой цели используйте браузерные события, обработчики, прокрутку элементов и окна браузера, а также координаты элементов.

- Для улучшения визуального эффекта, можно использовать фоновое изображение с высоким разрешением.
- Экспериментируйте с коэффициентом параллакса (например, измените делитель) для достижения наилучшего визуального восприятия.

4. Реализовать динамическое обновление содержимого страницы при прокрутке, используя браузерные события, обработчики событий, а также знания о размерах и прокрутке элементов страницы.

- Используйте window.addEventListener для прослушивания события прокрутки страницы.
- Внутри обработчика события определите текущее положение страницы или прокрутки (window.scrollY).