УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационной безопасности

Кафедра инфокоммуникационных технологий

ВЕБ-ТЕХНОЛОГИИ В ИНФОКОММУНИКАЦИЯХ

Разработка сценариев на языке JavaScript

Минск 2022

Макейчик Е.Г.

Содержание

Лабораторная работа №6 Разработка сценариев на языке JavaScript 3
Глобальный объект 3
Браузерное окружение, спецификации 3
DOM (Document Object Model) 4
BOM (Browser Object Model) 4
DOM-дерево 4
Навигация по DOM-элементам 7
DOM-коллекции 9
Соседи и родитель 9
Навигация только по элементам 10
Поиск: getElement*, querySelector* 11
Живые коллекции 12
Свойства узлов: тип, тег и содержимое 13
Браузерные события 15
Обработчики событий 16
Использование свойства DOM-объекта 17
addEventListener 18
Объект события 18
Объект-обработчик: handleEvent 19
Всплытие и погружение 19
Делегирование событий 21
Прием проектирования «поведение» 22
Действия браузера по умолчанию 23
Опция «passive» для обработчика 24
Планирование: setTimeout и setInterval 25
Задание к лабораторной работе №6 27

Глобальный объект

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

В браузере он называется window, в Node.js – global, в другой среде исполнения может называться иначе.

Ко всем свойствам глобального объекта можно обращаться напрямую.

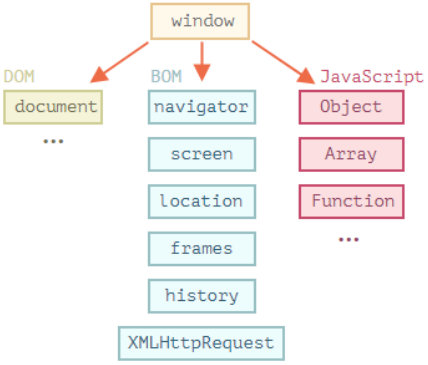
```
alert("Привет");
// это то же самое, что и
window.alert("Привет");
```

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта.

```
var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)
```

Браузерное окружение, спецификации

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами.



Из схемы видно, что имеется корневой объект window, который выступает в 2 ролях:

- Во-первых, это глобальный объект для JavaScript-кода.
- Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь window используется как глобальный объект.

```
function sayHi() {
  alert("Hello");
}

// глобальные функции доступны как методы глобального объекта:
window.sayHi();
```

А здесь используется window как объект окна браузера, чтобы узнать его высоту.

```
alert(window.innerHeight); // внутренняя высота окна браузера
```

Существует гораздо больше свойств и методов для управления окном браузера.

DOM (Document Object Model)

Document Object Model, сокращенно **DOM** – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект document – основная «входная точка». С его помощью можно что-то создавать или менять на странице.

Например:

```
// заменим цвет фона на красный,
document.body.style.background = "red";

// а через секунду вернём как было
setTimeout(() => document.body.style.background = "", 1000);
```

В примере использовали только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации https://dom.spec.whatwg.org.

BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, **BOM**) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

Объект navigator дает информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).

Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

Пример использования объекта location.

```
alert(location.href); // показывает текущий URL
if (confirm("Перейти на Wikipedia?")) {
  location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL
}
```

Функции alert/confirm/prompt тоже являются частью **BOM**: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

DOM-дерево

Основой HTML-документа являются теги.

В соответствии с **DOM**, каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

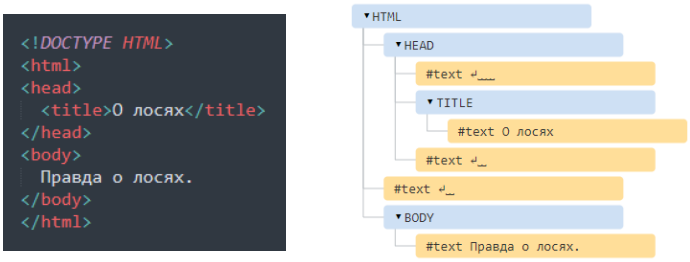
Все эти объекты доступны при помощи JavaScript, и их можно использовать их для изменения страницы.

Например, document.body – объект для тега <body>.

Если запустить этот код, то <body> станет красным на 3 секунды.

```
document.body.style.background = 'red'; // сделать фон красным

setTimeout(() => document.body.style.background = '', 3000); // вернуть назад
```

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит для кода:



Каждый узел этого дерева – это **объект**.

Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: `<html>` – это корневой узел, `<head>` и `<body>` его дочерние узлы и т.д.

Текст внутри элементов образует текстовые узлы, обозначенные как `#text`. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Например, в теге `<title>` есть текстовый узел `"О лосях"`.

Специальные символы в текстовых узлах:

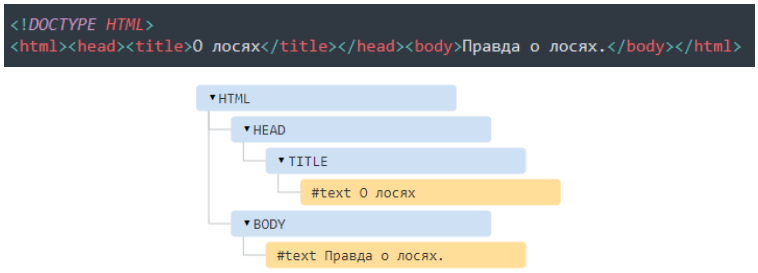
- перевод строки: `↵` (в JavaScript он обозначается как `\n`),
- пробел: `␣`.

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева **DOM**. Так, в примере выше в теге `<head>` есть несколько пробелов перед `<title>`, которые образуют текстовый узел `#text` (он содержит в себе только перенос строки и несколько пробелов).

Существует всего два исключения из этого правила:

1. Пробелы и перевод строки перед тегом `<head>` игнорируются.
2. Если что-либо записано после закрывающего тега `</body>`, браузер автоматически перемещает эту запись в конец `body`, поскольку спецификация HTML требует, чтобы все содержимое было внутри `<body>`. Поэтому после закрывающего тега `</body>` не может быть никаких пробелов.

В остальных случаях– если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева **DOM**, и если их удалить, то в **DOM** их тоже не будет.

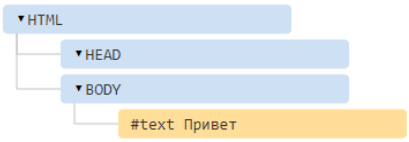


Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении **DOM**.

Например, в начале документа всегда должен быть тег `<html>`. Даже если его нет в документе – он будет в дереве **DOM**, браузер его создаст. То же самое касается и тега `<body>`.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернет его в теги <html> и <body>, добавит необходимый тег <head>, и **DOM** будет выглядеть так.

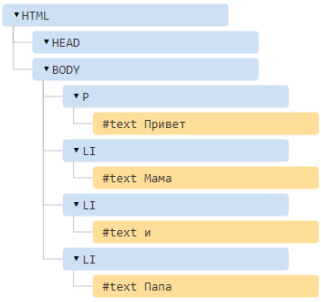


При генерации **DOM** браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

Есть такой документ с незакрытыми тегами.

```
<p>Привет
<li>Мама
<li>и
<li>Папа
```

DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали.

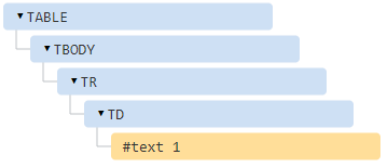


Особый случай – работа с таблицами. По стандарту **DOM** у них должен быть <tbody>, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет <tbody> в **DOM** самостоятельно.

Для такого HTML:

```
<table id="table"><tr><td>1</td></tr></table>
```

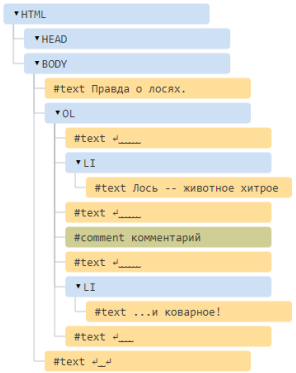
DOM-структура будет такой:



Другие типы узлов

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий:

```
!DOCTYPE HTML>
<html>
<body>
  Правда о лосях.
  <ol>
    <li>Лось -- животное хитрое</li>
    <!-- комментарий -->
    <li>...и коварное!</li>
  </ol>
</body>
</html>
```



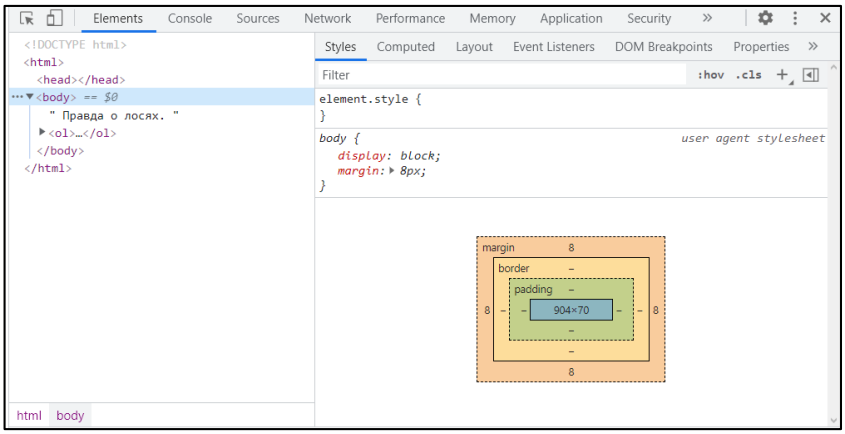
Существует 12 типов узлов. В основном работа ведется с 4-мя из них:

- document – «входная точка» в DOM.
- узлы-элементы – HTML-теги, основные строительные блоки.
- текстовые узлы – содержат текст.
- комментарии – в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

Чтобы посмотреть структуру **DOM** в реальном времени, можно использовать Live DOM Viewer.

Другой способ исследовать **DOM** – это использовать инструменты разработчика браузера.

Откройте страницу index.html, включите инструменты разработчика и перейдите на вкладку Elements.



Взаимодействие с консолью

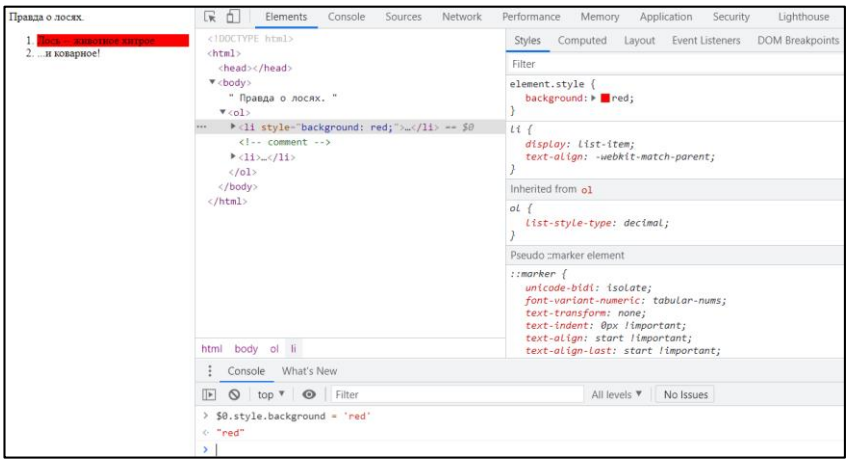
При работе с **DOM** часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Перемещение между вкладками Elements и Console выполняется пошагово:

Шаг 1. На вкладке Elements выберите первый элемент <li>.

Шаг 2. Нажмите Esc – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как $0; предыдущий, выбранный до него, как $1 и т.д.

Теперь можно запускать на них команды. Например $0.style.background = 'red' сделает выбранный элемент красным.

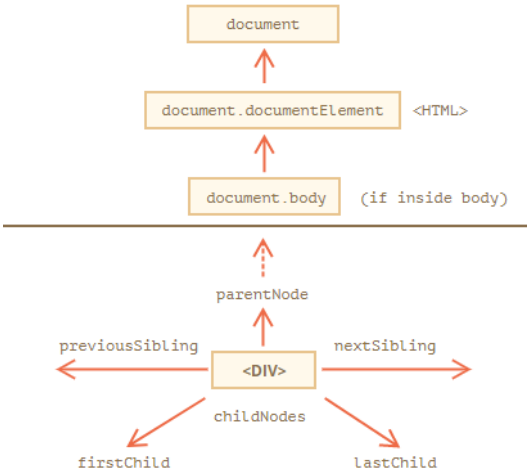


Навигация по DOM-элементам

DOM позволяет делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий **DOM-объект**.

Все операции с **DOM** начинаются с объекта document. Это главная «точка входа» в **DOM**. Из него можно получить доступ к любому узлу.

Так выглядят основные ссылки, по которым можно переходить между узлами **DOM**:



documentElement и body

Самые верхние элементы дерева доступны как свойства объекта document:

<html> = document.documentElement

Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

<body> = document.body

Другой часто используемый DOM-узел – узел тега <body>: document.body.

<head> = document.head

Тег <head> доступен как document.head.

childNodes, firstChild, lastChild

Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.

Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

В примере ниже детьми тега <body> являются теги и <ul> (и несколько пустых текстовых узлов):

```
<html>
<body>
  <div>Начало</div>

  <ul>
    <li>
      <b>Информация</b>
    </li>
  </ul>
</body>
</html>
```

Потомки <body> – это и прямые дети , <ul> и вложенные в них: <li> (потомок <ul>) и <b> (потомок <li>) – в общем, все элементы поддерева.

Коллекция childNodes содержит список всех детей, включая текстовые узлы.

Пример ниже последовательно выведет детей document.body:

```
<html>
<body>
  <div>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT
    }
  </script>
  ...какой-то HTML-код...
</body>
</html>
```

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу.

Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

DOM-коллекции

childNodes похож на массив. На самом деле это не массив, а коллекция – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

Для перебора коллекции можно использовать for…of:

```
for (let node of document.body.childNodes) {
alert(node); // покажет все узлы из коллекции
}
```

Коллекции перебираются циклом for…of. Некоторые начинающие разработчики пытаются использовать для этого цикл for…in.

Так нельзя. Цикл for…in перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно не нужны:

```
<body>
<script>
  // выводит 0, 1, length, item, values и другие свойства.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

Соседи и родитель

Соседи – это узлы, у которых один и тот же родитель.

Например, здесь <head> и <body> соседи:

```
<html>
  <head>...</head><body>...</body>
</html>
```

Можно сказать, что <body> – «следующий» или «правый» сосед <head>. Также можно сказать, что <head> «предыдущий» или «левый» сосед <body>.

Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling.

Родитель доступен через parentNode.

Например:

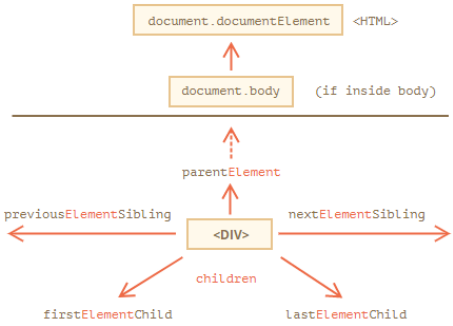
```
// родителем <body> является <html>
alert( document.body.parentNode === document.documentElement ); // выведет true

// после <head> идёт <body>
alert( document.head.nextSibling ); // HTMLBodyElement

// перед <body> находится <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Навигация только по элементам

Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в childNodes находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть.

Но для большинства задач текстовые узлы и узлы-комментарии не нужны.

Рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово Element:

- children – коллекция детей, которые являются элементами.
- firstElementChild, lastElementChild – первый и последний дочерний элемент.
- previousElementSibling, nextElementSibling – соседи-элементы.
- parentElement – родитель-элемент.

Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа.

Элемент <table>, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

- table.rows – коллекция строк <tr> таблицы.
- table.caption/tHead/tFoot – ссылки на элементы таблицы <caption>, <thead>, <tfoot>.
- table.tBodies – коллекция элементов таблицы <tbody> (по спецификации их может быть больше одного).

<thead>, <tfoot>, <tbody> предоставляют свойство rows:

- tbody.rows – коллекция строк <tr> секции.

<tr>:

- tr.cells – коллекция <td> и <th> ячеек, находящихся внутри строки <tr>.
- tr.sectionRowIndex – номер строки <tr> в текущей секции <thead>/<tbody>/<tfoot>.
- tr.rowIndex – номер строки <tr> в таблице (включая все строки таблицы).

<td> и <th>:

- td.cellIndex – номер ячейки в строке <tr>.

```
<table id="table">
  <tr>
    <td>один</td><td>два</td>
  </tr>
  <tr>
    <td>три</td><td>четыре</td>
  </tr>
</table>

<script>
  // выводит содержимое первой строки, второй ячейки
  alert( table.rows[0].cells[1].innerHTML ) // "два"
</script>
```

Поиск: getElement*, querySelector*

Свойства навигации по DOM хороши, когда элементы расположены рядом. Если нужно получить произвольный элемент страницы используются дополнительные методы поиска.

document.getElementById

Если у элемента есть атрибут id, то можно получить его вызовом document.getElementById(id), где бы он ни находился.

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // получить элемент
  let elem = document.getElementById('elem');

  // сделать его фон красным
  elem.style.background = 'red';
</script>
```

querySelectorAll

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

Следующий запрос получает все элементы <li>, которые являются последними потомками в <ul>:

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

querySelector

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдет все элементы, а потом возьмет первый, в то время как elem.querySelector найдет только первый и остановится.

matches

Предыдущие методы искали по DOM.

Метод elem.matches(css) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false.

Этот метод удобен, когда нужно перебрать элементы (например, в массиве или в чем-то подобном) и попытаться выбрать те из них, которые интересуют.

Например:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```

closest

Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины.

Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

Другими словами, метод closest поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо null, если такой элемент не найден.

Например:

```
<h1>Содержание</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Глава 1</li>
    <li class="chapter">Глава 2</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</script>
```

Живые коллекции

Все методы "getElementsBy*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении.

В приведенном ниже примере есть два скрипта.

- Первый создает ссылку на коллекцию . На этот момент ее длина равна 1.
- Второй скрипт запускается после того, как браузер встречает еще один , теперь ее длина – 2.

```
<div>First div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 2
</script>
```

Напротив, querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов.

Если использовать его в примере выше, то оба скрипта вернут длину коллекции, равную 1:

```
<div>First div</div>

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 1
</script>
```

Длина статической коллекции не изменилась после появления нового div в документе.

Свойства узлов: тип, тег и содержимое

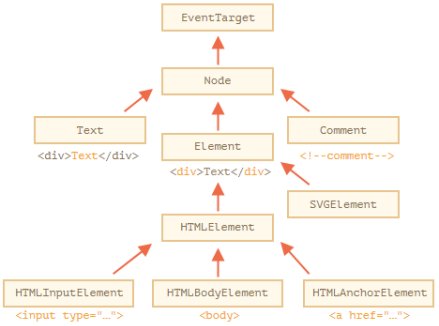
Классы DOM-узлов

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу <a>, есть свойства, связанные со ссылками, а у соответствующего тегу <input> – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию.

Каждый DOM-узел принадлежит соответствующему встроенному классу.

Корнем иерархии является EventTarget, от него наследует Node и остальные DOM-узлы.

На рисунке ниже изображены основные классы:



Существуют следующие классы:

- EventTarget – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события».
- Node – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определенные классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
- Element – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не

только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.

- HTMLElement – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
  - HTMLInputElement – класс для тега <input>,
  - HTMLBodyElement – класс для тега <body>,
  - HTMLAnchorElement – класс для тега <a>,
  - …и т.д., каждому тегу соответствует свой класс, который предоставляет определенные свойства и методы.

Таким образом, полный набор свойств и методов данного узла собирается в результате наследования.

Рассмотрим DOM-объект для тега <input>. Он принадлежит классу HTMLInputElement.

Он получает свойства и методы из (в порядке наследования):

- HTMLInputElement – этот класс предоставляет специфичные для элементов формы свойства,
- HTMLElement – предоставляет общие для HTML-элементов методы (и геттеры/сеттеры),
- Element – предоставляет типовые методы элемента,
- Node – предоставляет общие свойства DOM-узлов,
- EventTarget – обеспечивает поддержку событий (поговорим о них дальше),
- …и, наконец, он наследует от Object, поэтому доступны также методы «обычного объекта», такие как hasOwnProperty.

Для того, чтобы узнать имя класса DOM-узла, вспомним, что обычно у объекта есть свойство constructor. Оно ссылается на конструктор класса, и в свойстве constructor.name содержится его имя:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

Можно просто привести его к строке:

```
alert( document.body ); // [object HTMLBodyElement]
```

Проверить наследование можно также при помощи instanceof:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true]
```

innerHTML: содержимое элемента

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки. Это один из самых мощных способов менять содержимое на странице.

Пример ниже показывает содержимое document.body, а затем полностью заменяет его:

```
<body>
  <p>Параграф</p>
  <div>DIV</div>

  <script>
    alert( document.body.innerHTML ); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
  </script>

</body>
```

outerHTML: HTML элемента целиком

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент.

```
<div id="elem">Привет <b>Мир</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>
</script>
```

textContent: просто текст

Свойство textContent предоставляет доступ к тексту внутри элемента за вычетом всех <тегов>.

Например:

```
<div id="news">
  <h1>Срочно в номер!</h1>
  <p>Марсиане атаковали человечество!</p>
</div>

<script>
  // Срочно в номер! Марсиане атаковали человечество!
  alert(news.textContent);
</script>
```

Свойство hidden

Атрибут и DOM-свойство hidden указывает на то, виден ли элемент или нет.

Можно использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

```
<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Технически, hidden работает так же, как style="display:none". Но его применение проще.

Мигающий элемент:

```
<div id="elem">Мигающий элемент</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

Другие свойства

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

- value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).
- href – адрес ссылки href для <a href="..."> (HTMLAnchorElement).
- id – значение атрибута id для всех элементов (HTMLElement).
- …и др.

Например:

```
<input type="text" id="elem" value="значение">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // значение
</script>
```

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).

Список самых часто используемых DOM-событий

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 и других событий.

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.

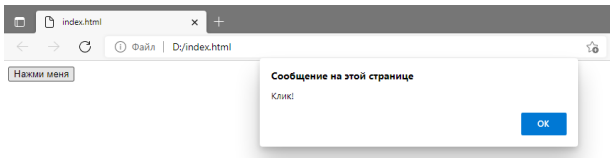
Использование атрибута HTML

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.

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

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

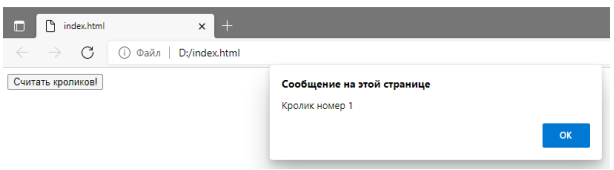


Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если поставить двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать ее 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!">
```

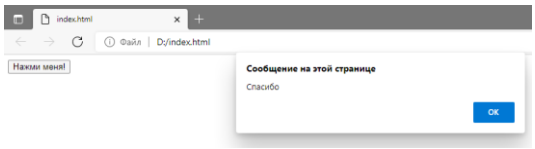


Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

К примеру, elem.onclick:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```



Если обработчик задан через атрибут, то браузер читает HTML-разметку, создает новую функцию из содержимого атрибута и записывает в свойство.

Этот способ, по сути, аналогичен предыдущему.

Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Эти два примера кода работают одинаково:

1. HTML:

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

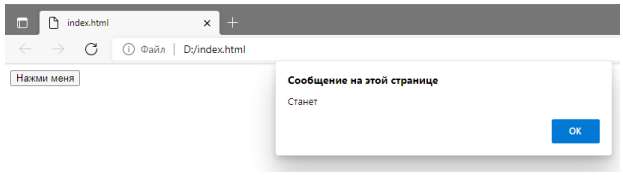
2. HTML + JS:

```
<input type="button" id="button" value="Кнопка">
<script>
  button.onclick = function() {
    alert('Клик!');
  };
</script>
```

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя.

В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

```
<input type="button" id="elem" onclick="alert('Было')" value="Нажми меня">
<script>
  elem.onclick = function() { // перезапишет существующий обработчик
    alert('Станет'); // выведется только это
  };
</script>
```



Обработчиком можно назначить и уже существующую функцию.

```
function sayThanks() {
  alert('Спасибо!');
}

elem.onclick = sayThanks;
```

Убрать обработчик можно назначением elem.onclick = null.

Доступ к элементу через this

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже button выводит свое содержимое, используя this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

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 ее подсвеченной, а другая – выдавать сообщение.

Необходимо 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 предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener.

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 удален после выполнения.
- capture: фаза, на которой должен сработать обработчик. 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]);
```

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>
```

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 создает объект события, записывает в него детали и передает 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 можно получить элемент из event.currentTarget.
- event.clientX / event.clientY – координаты курсора в момент клика относительно окна, для событий мыши.

Объект-обработчик: handleEvent

Можно назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent.

К примеру:

```
<button id="elem">Нажми меня</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " на " + event.currentTarget);
    }
  });
</script>
```

Если addEventListener получает объект в качестве обработчика, он вызывает object.handleEvent(event), когда происходит событие.

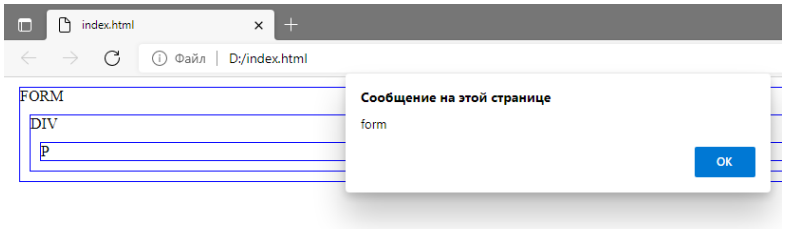
Всплытие и погружение

Всплытие

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

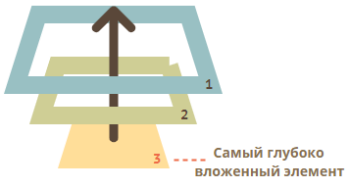
Например, есть 3 вложенных элемента FORM > DIV > P с обработчиком на каждом:

```
<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 увидим три оповещения: p → div → form.

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

event.target

Всегда можно узнать, на каком конкретно элементе произошло событие.

Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target.

Отличия от this (=event.currentTarget):

- event.target – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
- this – это «текущий» элемент, до которого дошло всплытие, на нем сейчас выполняется обработчик.

Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывет до элемента <form>, на котором сработает обработчик.

При этом внутри обработчика form.onclick:

- this (=event.currentTarget) всегда будет элемент <form>, так как обработчик сработал на ней.
- event.target будет содержать ссылку на конкретный элемент внутри формы, на котором произошел клик.

Смотреть пример обработчика события

Прекращение всплытия

Всплытие идет 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 своем пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод event.stopPropagation().

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

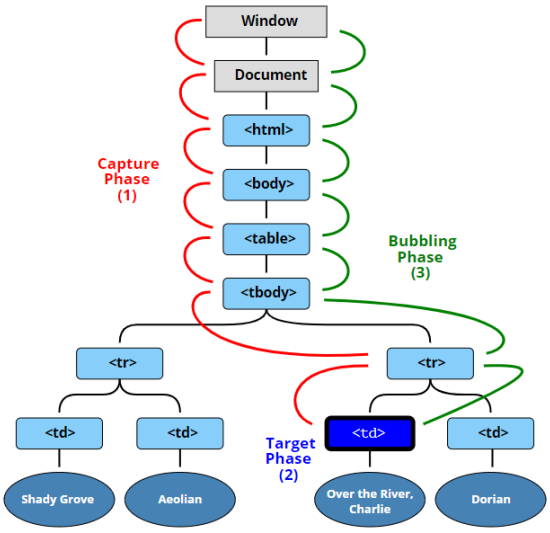
Погружение

Существует еще одна фаза из жизненного цикла события – **погружение** (иногда ее 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 идет сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Изображение в 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 идет наверх (всплытие), вызывая по пути обработчики.

Делегирование событий

Всплытие и перехват событий позволяет реализовать один из самых важных приемов разработки – делегирование.

Идея в том, что если есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, ставится один обработчик на их общего предка.

Из него можно получить целевой элемент `event.target`, понять на каком именно потомке произошло событие и обработать его.

Смотреть пример

```
<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 больше.

Необходимо реализовать подсветку ячейки `<td>` при клике.

Вместо того, чтобы назначать обработчик `onclick` для каждой ячейки `<td>` (их может быть множество) – повесим «единый» обработчик на элемент `<table>`.

Он будет использовать `event.target`, чтобы получить элемент, на котором произошло событие, и подсветить его.

```
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
};
```

В этом коде нет разницы, сколько ячеек в таблице. Можно добавлять, удалять <td> из таблицы динамически в любое время, и подсветка будет стабильно работать.

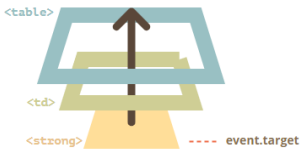
Однако, у текущей версии кода есть недостаток.

Клик может быть не на теге <td>, а внутри него.

Если взглянуть на HTML-код таблицы внимательно, видно, что ячейка <td> содержит вложенные теги, например :

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

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



Внутри обработчика table.onclick нужно по event.target разобраться, был клик внутри <td> или нет.

Улучшенная версия кода:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};
```

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае нужен <td>, находящийся выше по дереву от исходного элемента.
2. Если event.target не содержится внутри элемента <td>, то вызов вернет null, и ничего не произойдет.
3. Если таблицы вложенные, event.target может содержать элемент <td>, находящийся вне текущей таблицы. В таких случаях нужно проверить, действительно ли это <td> таблицы.
4. Если это так, то элемент будет подсвечен.

Прием проектирования «поведение»

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

Прием 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: «Счетчик»

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>
```

Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования здесь фактически добавлен новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Поведение: «Переключатель» (Toggler)

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

```
<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>
```

Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, можно просто написать атрибут data-toggle-id.

Не нужно писать JavaScript-код для каждого элемента, который должен так себя вести. Просто используется поведение. Обработчики на уровне документа сделают это возможным для элемента в любом месте страницы.

Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

- Клик по ссылке инициирует переход на новый URL.
- Нажатие на кнопку «отправить» в форме – отсылку ее на сервер.
- Зажатие кнопки мыши над текстом и ее движение в таком состоянии – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера не нужно, его можно отменить.

Отмена действия браузера

Есть два способа отменить действие браузера:

- Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
- Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

В следующем примере при клике по ссылке переход не произойдет:

```
<a href="/" onclick="return false">Нажми здесь</a>
или
<a href="/" onclick="event.preventDefault()">здесь</a>
```

Рассмотрим меню для сайта, например:

```html
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
```


```css
.menu li {
  display: inline-block;
  margin: 0;
}

.menu > li a {
  display: inline-block;
  margin: 0 2px;
  outline: none;
  text-align: center;
  text-decoration: none;
  font: 14px/100% sans-serif;
  padding: .5em 2em .55em;
  text-shadow: 0 1px 1px rgba(0, 0, 0, .3);
  border-radius: .5em;
  box-shadow: 0 1px 2px rgba(0, 0, 0, .2);
  color: #d9eef7;
  border: solid 1px #0076a3;
  background: #0095cd;
}

.menu > li:hover a {
  text-decoration: none;
  background: #007ead;
}
```

В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами <a>. В этом подходе есть некоторые преимущества, например:

- Некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если использовать <button> или , то данное сочетание работать не будет.
- Поисковые движки переходят по ссылкам <a href="..."> при индексации.

Поэтому в разметке используются <a>. Но необходимо обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить.

```js
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert( href ); // может быть подгрузка с сервера, генерация интерфейса и т.п.

  return false; // отменить действие браузера (переход по ссылке)
};
```

Если убрать return false, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из href.

Использование здесь делегирования событий делает меню очень гибким. Можно добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

Опция «passive» для обработчика

Необязательная опция passive: true для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault().

Есть некоторые события, как touchmove на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но можно отменить это действие, используя preventDefault() в обработчике.

Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если preventDefault не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция passive: true сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает ее немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие.

Для некоторых браузеров (Firefox, Chrome) опция passive по умолчанию включена в true для таких событий, как touchstart и touchmove.

Планирование: setTimeout и setInterval

Можно вызывать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

- setTimeout позволяет вызвать функцию один раз через определенный интервал времени.
- setInterval позволяет вызывать функцию регулярно, повторяя вызов через определенный интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout()

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Параметры:

- func|code
  Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.
- delay
  Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.
- arg1, arg2…
  Аргументы, передаваемые в функцию (не поддерживается в IE9-).

Данный код вызывает sayHi() спустя одну секунду:

```
function sayHi() {
  alert('Привет');
}

setTimeout(sayHi, 1000);
```

Используя аргументы:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Привет", "Карина"); // Привет, Карина
```

Если первый аргумент является строкой, то JavaScript создаст из нее функцию.

Использовать строки не рекомендуется. Вместо этого лучше использовать функции.

```
setTimeout(() => alert('Привет'), 1000);
```

setInterval()

Метод setInterval имеет такой же синтаксис как setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId).

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

```
// повторить с интервалом 2 секунды
let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Задание к лабораторной работе №6

Для выполнения лабораторной работы необходимо установить и настроить редактор кода.

Задание к лабораторной работе №6 состоит из задач разного уровня сложности. Выполнение задач 1-5 оценивается максимально в 5 баллов, задач 1-10 в 10 баллов.

Задача 1

Условие: Необходимо создать скрипт который выделит желтым цветом все ячейки в таблице по диагонали.

1. Нужно получить из таблицы `<table>` все диагональные `<td>` и выделить их, используя код:

```
//  в переменной td находится DOM-элемент для тега <td>
td.style.backgroundColor = 'yellow';
```

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5
C1	C2	C3	C4	C5
D1	D2	D3	D4	D5
E1	E2	E3	E4	E5

Задача 2

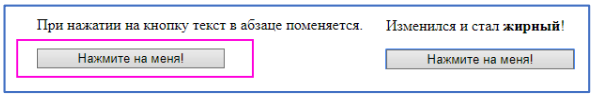
Условие: Дан HTML код. Необходимо поменять содержимое элементов с классом `pr` на их порядковый номер в коде.

1. Создать кнопку, по нажатию на которую будет вызываться функция `func` которая использует метод `querySelectorAll`.

```
<h2 class="pr">Заголовок с классом pr.</h2>
<p class="pr">Абзац с классом zzz.</p>
<p class="pr">Абзац с классом zzz.</p>
<p>Просто абзац, не поменяется.</p>
```

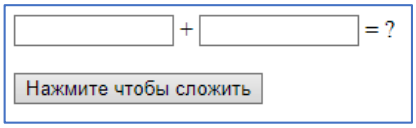
Задача 3

Условие: Необходимо повторить страницу по образцу. Пояснение.



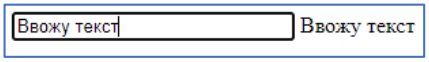
Задача 4

Условие: Необходимо повторить страницу по образцу. Пояснение.



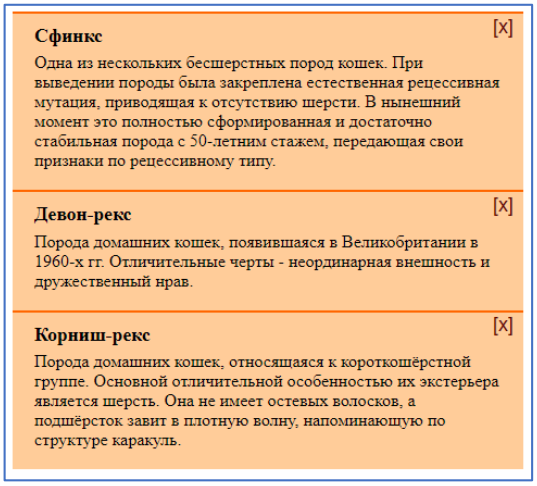
Задача 5

Условие: Необходимо повторить страницу по образцу. Пояснение.



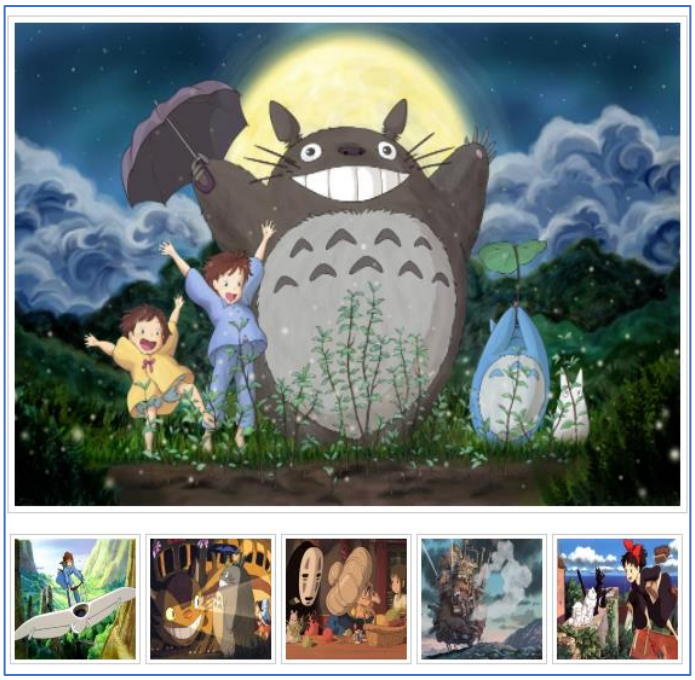
Задача 6

Условие: Дан список сообщений с кнопками для удаления [x]. Необходимо сделать кнопки рабочими. Используйте делегирование. Пояснение. Скачать исходные файлы.



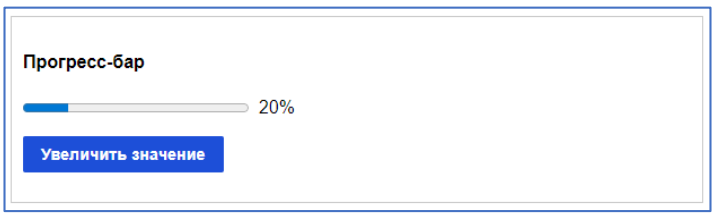
Задача 7

Условие: Необходимо создать галерею изображений, в которой основное изображение изменяется при клике на уменьшенный вариант. Используйте делегирование. Пояснение. Скачать исходные файлы.



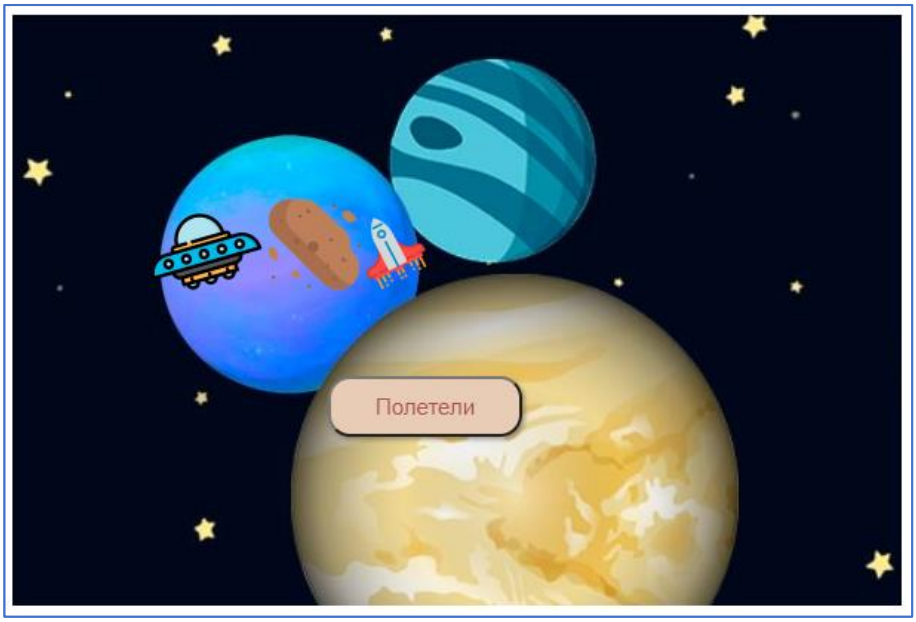
Задача 8

Условие: Необходимо создать страницу, в которой значение атрибута `value` элемента `<progress>` будет изменяется при клике на кнопку. Пояснение.



Задача 9

Условие: Необходимо создать страницу, в которой при клике не кнопку будет запускаться карусель элементов. Пояснение. Скачать исходные файлы.



Задача 10

Условие: Необходимо создать страницу с кнопкой «Отправить в работу», которая при нажатии превращается в яхту и плывет. При загрузке страницы кнопка отрисуется так, как будет написано в стилях. Потом, при нажатии, кнопка трансформируется в яхту с помощью скрипта и уплывает за границу экрана. Пояснение.

1. HTML:
   - Создать HTML, подключить CSS-стили и скрипт.
   - Сделать общий контейнер, куда сложить всю будущую анимацию и кнопку.
   - В контейнер поместить все детали яхты, они должны быть невидимые.
2. CSS:
   - Отрисовать все элементы яхты, они должны быть спрятаны за нарисованной кнопкой.
   - Добавить свойство трансформации к парусам, мачте и самой кнопке, чтобы превратить все в яхту.
3. JS:
   - Получить доступ ко всем элементам страницы по их id.
   - Добавить обработчик нажатия на кнопку.
   - При нажатии последовательно поднять мачту, паруса, а затем уплыть.