План практической работы № 6 для преподавателя

Оглавление

Введение

Эта практика посвящена Объектной модели документа, или коротко DOM (Document *Object Model*). DOM является просто специальным термином для "всего на Web-странице". *Объектная модель* включает каждую таблицу, изображение, ссылку, *поле* формы и т.д. на Web-странице. JavaScript позволяет манипулировать с любым элементом на странице в реальном времени. Можно скрывать или полностью удалять любой элемент, добавлять элементы, копировать их, изменять такие свойства, как цвет, ширина, *высота*, и т.д., а при некотором воображении можно даже реализовать функции перетаскивания, анимации и почти все остальное, что можно придумать.

Подробнее теоретический материал будет рассмотрен на лекции. Обратимся к практике.

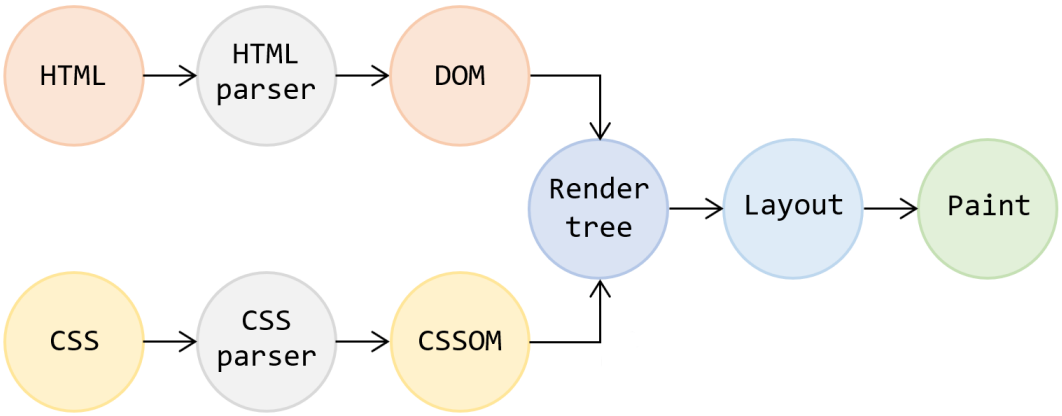
Документацию по DOM-моделям и использованию в JavaScript следует смотреть по ссылке: http://www.w3.org/standards/techs/dom#w3c_all

Как создаётся веб-страница?

Когда браузер загружает HTML-код страницы, он строит на основании него **объектную модель документа** (на английском *Document Object Model* или сокращённо *DOM*).

Рассмотрим **все основные этапы работ**, которые выполняет браузер для преобразования исходного кода HTML-документа в отображение

стилизованной и интерактивной картинки на экране. Этот процесс называется **Critical Rendering Path (CRP)**.



Хотя этот процесс состоит из большого количества шагов, их грубо можно представить в виде двух:

1. Анализирует HTML-документ, чтобы определить то, что в конечном итоге нужно отобразить на странице;
2. Выполняет отрисовку того что нужно отобразить.

Результатом первого этапа является **формирование дерева рендеринга** (*render tree*). Данное дерево содержит видимые элементы и текст, которые нужно отобразить на странице, и также связанные с ними стили. Это дерево дублирует структуру DOM, но включает как мы отметили выше только видимые элементы. В *render tree* каждый элемент содержит соответствующий ему объект DOM и рассчитанные для него стили. **Таким образом, *render tree* описывает визуальное представление DOM.**

Чтобы построить **дерево рендеринга**, браузеру нужны две вещи:

- **DOM**, который он формирует из полученного HTML-кода;
- **CSSOM (CSS Object Model)**, который он строит из загруженных и распознанных стилей.

На втором этапе браузер выполняет отрисовку *render tree*. Для этого он:

- рассчитывает положение и размеры каждого элемента в *render tree*, этот шаг называется **Layout**;
- выполняет рисование, этот шаг называется **Paint**.

После **Paint** все нарисованные элементы находятся на одном слое. Для повышения производительности страницы браузер выполняет ещё один шаг, который называется **Composite**. В нем он группирует элементы по

композиционным слоям. Именно благодаря этому этапу мы можем создать на странице плавную анимацию элементов при использовании таких свойств как transform, opacity. Так как изменение этих свойств вызовет только одну задачу **Composite**.

Layout и **Paint** – это ресурсоемкие процессы, поэтому для хорошей отзывчивости вашей страницы или веб-приложения, необходимо свести к минимуму операции которые их вызывают.

Список свойств, изменение которых вызывают **Paint**:

- `color;`
- `background;`
- `visibility;`
- `border-style и другие.`

Список свойств, изменение которых вызывает **Layout**:

- `width и height;`
- `padding и margin;`
- `display;`
- `border;`
- `top, left, right и bottom;`
- `position;`
- `font-size и другие.`

Чтобы понимать какую «стоимость» имеет то или иное свойство (какие шаги за собой повлекут изменения), можно поискать соответствующее расширение для среды, в которой вы пишете.

Что же такое DOM?

DOM – это объектное представление исходного HTML-кода документа. Процесс формирования DOM происходит так: браузер получает HTML-код, парсит его и строит DOM.

Затем, как мы уже отмечали выше браузер использует DOM (а не исходный HTML) для строительства **дерева рендеринга**, потом выполняет **layout** и так далее.

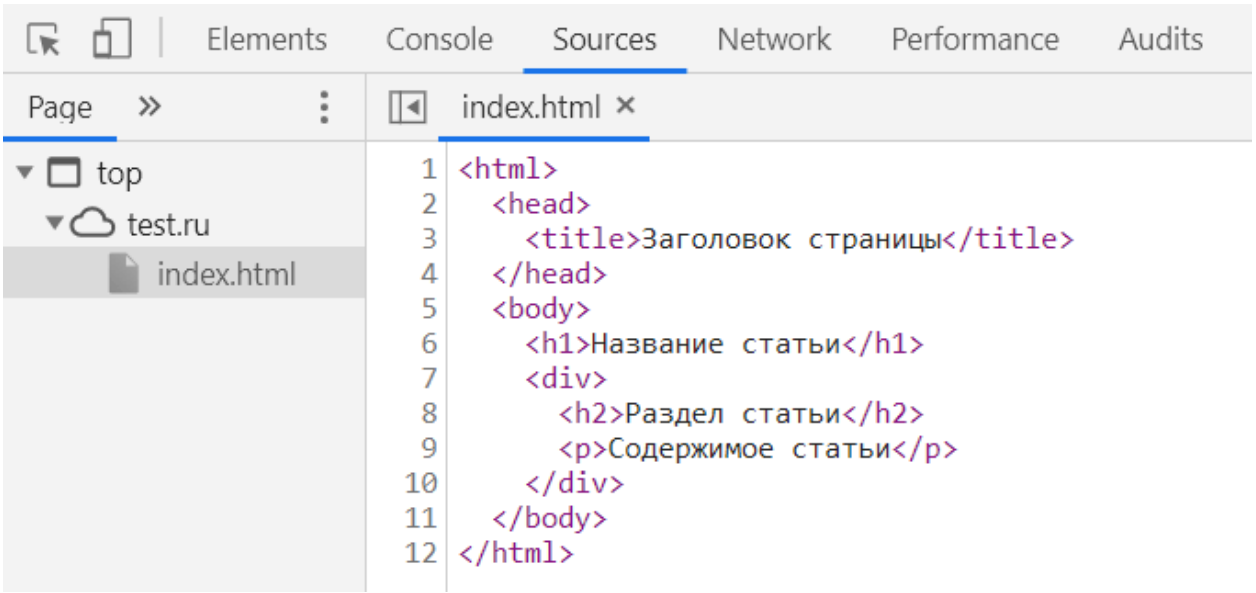
Почему не использовать в этом случае просто HTML? Потому что HTML – это текст, и с ним невозможно работать так, как есть. Для этого нужно его разобрать и создать на его основе объект, что и делает браузер. И этим объектом является DOM.

Итак, **DOM – это объектная модель документа, которую браузер создаёт в памяти компьютера на основании HTML-кода**.

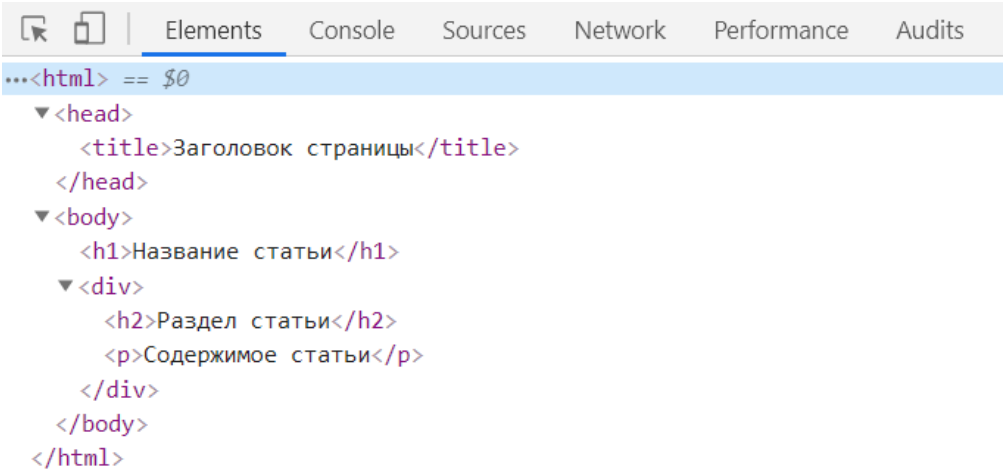
Но, браузер использует DOM не только для выполнения процесса **CRP**, но также предоставляет нам **программный доступ к нему**. Следовательно, с помощью JavaScript мы можем изменять DOM.

Благодаря тому, что JavaScript позволяет изменять DOM, мы можем создавать динамические и интерактивные веб-приложения и сайты. С помощью JavaScript мы можем менять всё что есть на странице. Сейчас практически нет сайтов, в которых не используется работа с DOM.

В браузере Chrome исходный HTML-код страницы, можно посмотреть во вкладке «Source» на панели «Инструменты веб-разработчика»:



На вкладке **Elements** мы видим что-то очень похожее на DOM:



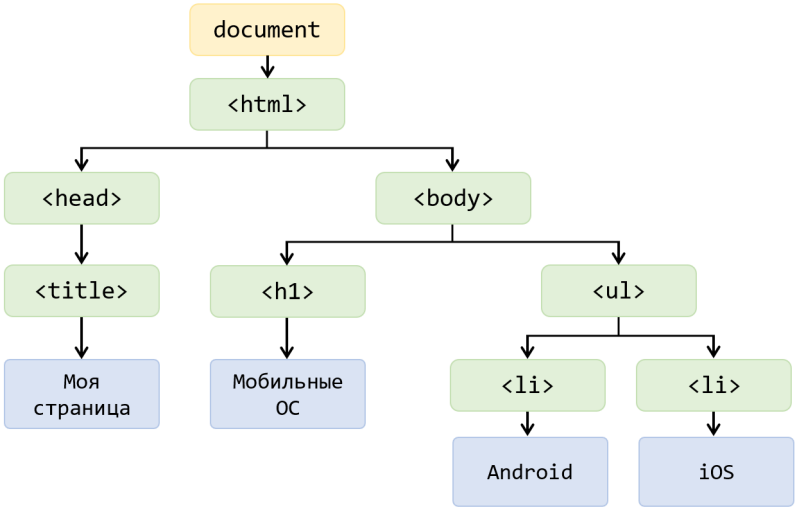
Однако, иструменты разработчика включают сюда дополнительную информацию, которой нет в DOM. Отличным примером этого являются псевдоэлементы в CSS. Псевдоэлементы, созданные с помощью

селекторов `::before` и `::after`, являются **частью CSSOM и дерева рендеринга**, и технически не являются частью DOM. Мы с ними не может взаимодействовать посредством JavaScript.

По факту DOM создается только из исходного HTML-документа и не включает псевдоэлементы. Но в инспекторе элементов DevTools они имеются.

Как строится DOM?

Объектная структура DOM представляет собой дерево узлов (узел на английском называется `node`). При этом DOM-узлы образуются из всего, что есть в HTML: тегов, текстового контента, комментариев и т.д.



Корневым узлом DOM-дерева является объект `document`, он представляет сам этот документ. Далее в нём расположен узел `<html>`. Получить этот элемент в коде можно так:

```
const elHTML = document.documentElement;
```

В `<html>` находятся 2 узла-элемента: `<head>` и `<body>`. Получить их в коде можно так:

```
const elHead = document.head;
const elBody = document.body;
```

В `<head>` находится DOM-узел `<title>`:

```
const elTitle = document.title;
```

В `<title>` находится текстовый узел. Теперь перейдём к `<body>`. В нём находятся 2 элемента `<h1>` и `<ul>`, и так далее.

При этом, как вы уже поняли, узлы в зависимости от того, чем они образованы делятся на разные типы. В DOM выделяют:

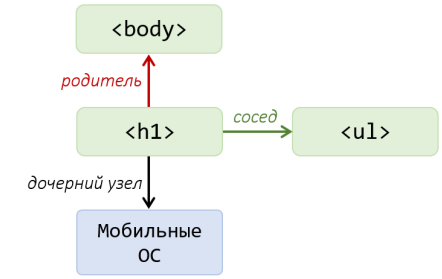
- узел, представляющий собой весь документ; этим узлом является объект `document`; он выступает входной точкой в DOM;
- узлы, образованные тегами, их называют узлами-элементами или просто элементами;
- текстовые узлы, они образуются текстом внутри элементов;
- узлы-комментарии и так далее.

Каждый узел в дереве DOM является объектом. Но при этом формируют структуру DOM только узлы-элементы. Текстовые узлы, например, содержат в себе только текст. Они не могут содержать внутри себя другие узлы. Поэтому вся работа с DOM в основном связана с узлами-элементами.

Чтобы перемещаться по узлам DOM-дерева нужно знать какие они имеют отношения. Зная их можно будет выбирать правильные свойства и методы. Связи между узлами, определяются их вложенностью. Каждый узел в DOM может иметь следующие виды отношений:

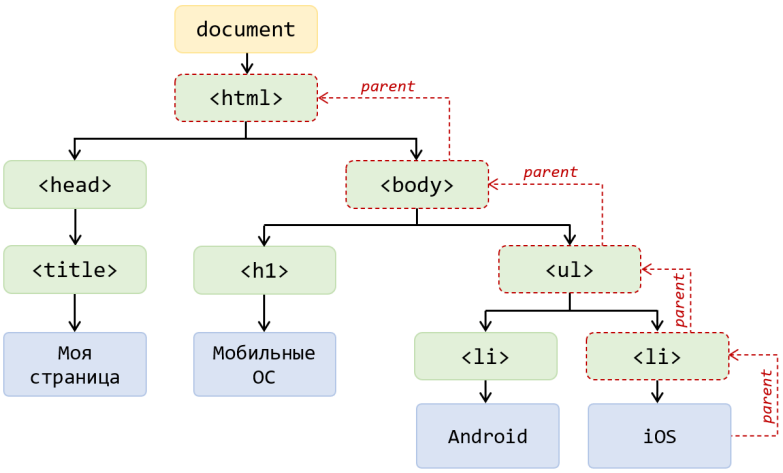
- **родитель** – это узел, в котором он непосредственно расположен; при этом родитель у узла может быть только один; также узел может не иметь родителя, в данном примере им является `document`;
- **дети или дочерние узлы** – это все узлы, которые расположены непосредственно в нём; например, узел `<ul>` имеет 2 детей;
- **соседи или сиблинги** – это узлы, которые имеют такого же родителя что и этот узел;
- **предки** – это его родитель, родитель его родителя и так далее;
- **потомки** – это все узлы, которые расположены в нем, то есть это его дети, а также дети его детей и так далее.

Например, узел-элемент `<h1>` имеет в качестве родителя `<body>`. Ребенок у него один – это текстовый узел «Мобильные ОС». Сосед у него тоже только один – это `<ul>`



.

Теперь рассмотрим, каких предков имеет текстовый узел «iOS». У него они следующие: `<li>`, `<ul>`, `<body>` и `<html>`.



Зачем нужно знать, как строится DOM-дерево? Во-первых, это понимание той среды, в которой вы хотите что-то изменять. Во-вторых, большинство действий при работе с DOM сводится к поиску нужных элементов. Но, не зная, как устроено DOM-дерево и отношения между узлами, найти что-то в нём будет достаточно затруднительно.

Типы и имена DOM-узлов

Узнать тип узла в DOM можно с помощью свойства `nodeType`:

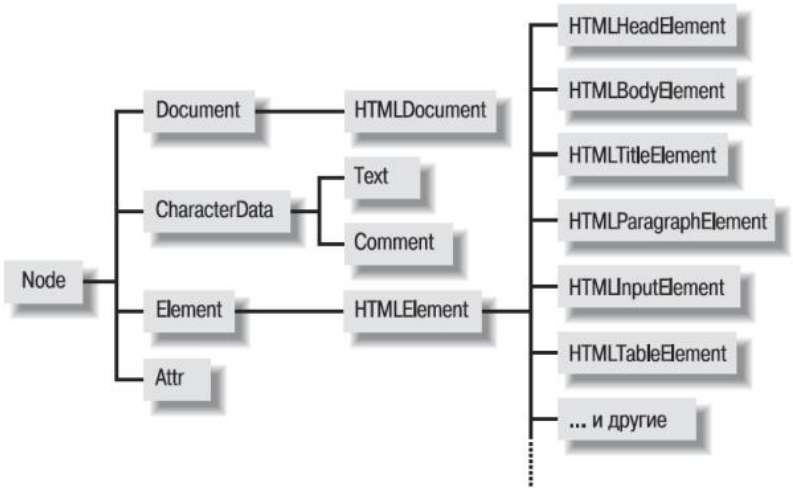
```
console.log(document.nodeType); // 9

console.log(document.body.nodeType); // 1
```

Это свойство возвращает число от 1 до 12, обозначающее тип узла.

Основные значения:

```
1 – элемент (Node.ELEMENT_NODE);
2 – атрибут (Node.ATTRIBUTE_NODE);
3 – текстовый узел (Node.TEXT_NODE);
8 – комментарий (Node.COMMENT_NODE);
9 – document (Node.DOCUMENT_NODE);
10 – узел, содержащий тип документа
(Node.DOCUMENT_TYPE_NODE);
11 – узел, представляющий фрагмент
документа DocumentFragment (Node.DOCUMENT_FRAGMENT_NODE).
```

Иерархия DOM-узлов:



Теперь изучим свойство `nodeName`. С его помощью мы можем узнать имя узла или тег, если узел является элементом:

```
console.log(document.body.nodeName); // "BODY"
console.log(document.doctype.nodeName) // "html"
console.log(document.nodeName); // #document"
```

В коде JavaScript программы предоставляется возможность поиска элементов по имени. И это работает для любого тега.

```
var tables = document.getElementsByTagName("table");
alert("Количество таблиц в документе: " + tables.length);
```

Если же нужен конкретный элемент, то либо необходимо знать его порядковый номер в массиве найденных элементов, либо иметь возможность идентифицировать по классу, либо идентификатору. В следующем примере будет найден 4-й параграф

```
var myParagraph = document.getElementsByTagName("p")[3];
```

В этом примере будет найден параграф, имеющий уникальный идентификатор specialParagraph. Обратите внимание на то, что **одинаковых идентификаторов в одном документе быть не может.**

```
… var myParagraph = document.getElementById("specialParagraph");
```

Часто бывает полезно вставить элементы не как объекты, а как HTML текст. Для этого существует свойство innerHTML. Однако помните, что использование этого свойства полностью удалит все дочерние элементы, если они были у владельца innerHTML. Пример заполнения заголовка в таблице у созданного элемента table:

```
var table = document.createElement("table"); // Создать элемент
<table>
table.border = 1; // Установить атрибут
// Добавить в таблицу заголовок Имя|Тип|Значение
table.innerHTML =
"<tr><th>Имя</th><th>Тип</th><th>Значение</th></tr>";
```

Обработка событий

Теперь, имея общее представление о компоновке страницы, можно начинать ее модификацию. Начнем с создания простого эффекта изменения изображения:

```
<img src="button_off.png"
onmouseover="this.src='button_over.png';"
onmousedown="this.src='button_down.png';"
onmouseout ="this.src='button_off.png';"
onmouseup  ="this.src='button_over.png';">
```

В этом коде присутствуют 4 события изображения: onmouseover, onmousedown, onmouseout и onmouseup. Каждое из этих событий имеет присоединенный простой фрагмент кода JavaScript, который изменяет атрибут src изображения. Создавая три разных изображения, можно легко и быстро создать изображение с тремя сменяющими друг друга состояниями.

Значение this, передаваемое обработчикам события мыши, содержит указатель на элемент, вызывающий этот обработчик.

Кроме изменения изображения, по сути, по событию мыши или любому другому событию на экране можно вызывать вашу функцию. Тут вы ограничен только фантазией. Подробнее о событиях в DOM читать тут: https://learn.javascript.ru/introduction-browser-events

Добавление и удаление элементов

Одной из задач, которая становится все более распространенной в современных приложениях JavaScript, является возможность добавления или удаления элементов страницы. Предположим, что имеется форма, которая позволяет послать кому-нибудь ссылку. Обычно используется одно поле ввода для адреса e-mail и второе - для имени получателя. Если требуется послать ссылку нескольким адресатам, то либо придется посылать форму несколько раз, либо можно было бы разместить на странице более одного набора полей имя/e-mail. Но в этом случае мы все еще ограничены заданным числом контактов. Если имеется пространство для 5 контактов и необходимо послать ссылку 20 людям, то форму придется заполнять 4 раза.

JavaScript позволяет обойти эту проблему, делая возможным динамическое дополнение и удаление содержимого страницы:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">

<head>
  <meta charset="utf-8">
  <title></title>
  <script type="text/javascript">
    let inputs = 0;

    function addContact() {
      let table = document.getElementById("contacts");
      let tr = document.createElement("TR");
      let td1 = document.createElement("TD");
      let td2 = document.createElement("TD");
      let td3 = document.createElement("TD");
      let new_name = document.createElement("p");
      let new_email = document.createElement("p");
      inputs++;
      if (inputs > 0) {
        let img = document.createElement('IMG');
        img.setAttribute('src', 'delete.gif');
        img.setAttribute('width', '10%');
        img.onclick = function() {
          removeContact(tr);
        }
        td3.appendChild(img);
      }
      new_name.textContent = "Name" + inputs;
      new_email.textContent = "Email" + inputs;
      table.appendChild(tr);
      tr.appendChild(td1);
      tr.appendChild(td2);
      tr.appendChild(td3);
      td1.appendChild(new_name);
      td2.appendChild(new_email);
    }

    function removeContact(tr) {
      tr.parentNode.removeChild(tr);
    }
  </script>
</head>

<body>
  <table>
    <tbody id="contacts">
      <tr>
        <td colspan="3">
          <button onclick="javascript:addContact();">Добавить
контакт</button>
```

```
        </td>
      </tr>
      <tr>
        <td>Name </td>
        <td>Email</td>
        <td></td>
      </tr>
    </tbody>
  </table>
</body>

</html>
```

Давайте исследуем код:

Прежде всего здесь имеется новая функция: document.createElement. Функция createElement создает задаваемый аргументом элемент. Можно видеть, что в строках сценария создается несколько элементов. В действительности создается новая строка TR, которая вставляется в таблицу.

В результате новая строка TR будет выглядеть следующим образом:

```
<tr>
   <td>
<p>Name</p>
   </td>
   <td>
      <p>Email</p>
   </td>
   <td>
      <img src="delete.gif" width="10%">
   </td>
</tr>
```

Другими словами, мы создали 7 элементов: 1 TR, 3 TD, 2 P и 1 IMG. Тег IMG будет использоваться как изображение кнопки " Удалить". Так как пользователь должен всегда видеть по крайней мере 1 строку ввода, то первую строку удалить невозможно. Поэтому в 12 строке сценария проверяется, что создается первая строка. Если строка не первая, то добавляется изображение.

После создания всех этих элементов остается в действительности поместить их в документ. Каждый элемент на странице имеет встроенную функцию appendChild, которую можно использовать для добавления к этому элементу потомка. Когда добавляется потомок, то он добавляется как последний элемент, поэтому если таблица уже имеет в качестве потомков 10 тегов TR и добавляется еще один, то он будет добавлен как 11-ый тег TR. Мы начинаем с получения ссылки на таблицу (строка 4). Затем мы

добавляем TR к этой таблице (строка 24) и добавляем затем 3 TD (строки 25-27).

Вот и все! Теперь у нас есть новый элемент TR, и он находится на странице. Осталось пояснить еще пару моментов. Чтобы форма была обработана правильно, все поля ввода должны иметь различные имена. Поэтому мы задаем имя двух полей ввода на основе счетчика (21-22), а затем увеличиваем счетчик (20). Это делается с помощью еще одной новой функции setAttribute, которая имеет два параметра: имя атрибута и значение атрибута. Для нее существует дополнительная функция getAttribute, которая имеет только один аргумент: имя атрибута, значение которого надо получить.

element.setAttribute("name", "elementName")

по сути то же самое, что

element.name="elementName"

Однако задание атрибута непосредственно, как в предыдущем примере, может иногда вызывать некоторые проблемы для различных браузеров или для некоторых специфических атрибутов. Поэтому хотя любой метод обычно будет работать, предпочтительным является первый метод, использующий setAttribute.

Необходимо также позаботиться о кнопке удаления. Мы уже знаем, что кнопка удаления для первой строки полей не создается, но необходимо заставить ее работать для всех остальных. Это делается в строках кода 15-16. Здесь к изображению добавлена функция onclick, которая вызывает функцию removeContact, передавая элемент TR в качестве единственного аргумента.

Взглянув на функцию removeContact, можно видеть, что сначала происходит обращение tr.parentNode к функции parentNode, которая является еще одной функцией для работы с DOM. Она просто возвращает порождающий элемент для текущего элемента. Если посмотреть на изображенное ранее *дерево документа*, то видно, что parentNode вернет элемент непосредственно над элементом, на котором он вызван. Поэтому tr.parentNode возвращает ссылку на элемент TABLE над TR. Затем вызывается функция removeChild на этом элементе TABLE, которая просто удаляет у предка указанного потомка.

Взглянув еще раз на строку 34, можно теперь увидеть, что она просто говорит: " Удалить элемент TR у его предка " или еще проще " Удалить элемент TR ".

Элементы потомки

Ко всем потомкам элементам можно обратиться с помощью атрибута childNodes, который возвращает массив, содержащий все узлы потомки текущего элемента. Можно также использовать атрибуты *firstChild* и lastChild на любом элементе, чтобы получить ссылки на первый или на последний элемент.

Чтобы увидеть, как это работает, давайте напишем сценарий для раскраски чередующихся строк TR в таблице.

Нужно не забывать, что DOM-дерево содержит решительно всё, что есть в HTML. Поэтому, если в вашем HTML-коде есть отступы, они тоже отобразятся как составляющие дерева.

Например, если таблица задана вот так:

```
<table>
  <tbody id='tab'>
    <tr>
      <td>First row</td>
    </tr>
    <tr>
      <td>Second row</td>
    </tr>
    <tr>
      <td>Third row</td>
    </tr>
  </tbody>
</table>
```

То список узлов-потомков потомки тега tbody будут выглядеть вот так:

```
> tab.childNodes
◂ ▶ NodeList(7) [text, tr, text, tr, text, tr, text]
```

Отступы тоже являются потомками. Тогда, если перебирать потомки, нам подойдёт не любой sibling, а только определённый, со значением tr.

```
function setColors(tbody, color1, color2){
        var colors  = [color1, color2];
        let k = 0;
        for(var i=0; i<tbody.childNodes.length; i++){
          if (tbody.childNodes[i].tagName == 'TR') {
          tbody.childNodes[i].style.backgroundColor = colors[k %
2];
          k ++;
```

```
    }
  }
}
```

Это не очень удобно. Проще сразу достать список всех элементов tr на странице и пройтись по ним:

```
function setColors(tbody, color1, color2){
  var colors  = [color1, color2];
  var trs     = tbody.getElementsByTagName('TR');

  for(var i=0; i<trs.length; i++){
    trs[i].style.backgroundColor = colors[i % 2];
  }
}
```

Работа с текстом

Работа с текстом немного отличается от работы с другими элементами DOM. Первое: каждый фрагмент текста на странице помещен в невидимый узел #TEXT. Поэтому следующий код HTML

```
<div id="ourTest">this is <a href="link.html">a link</a> and an
image: <img src="img.jpg"></div>
```

имеет четыре корневых элемента: текстовый узел со значением " this is ", элемент A, еще один текстовый узел со значением " and an image: " и, наконец, элемент IMG. Элемент A имеет конечный текстовый узел в качестве потомка со значением " a link ". Когда необходимо изменить текст, то прежде всего необходимо получить этот "невидимый" узел. Если мы хотим изменить текст " and an image: ", то необходимо написать:

```
document.getElementById('ourTest').childNodes[2].nodeValue =
'our new text';
```

document.getElementById('ourTest') дает нам тег div. childNodes[2] дает узел текста " and an image: " и наконец nodeValue изменяет значение этого узла текста.

А если требуется добавить к этому еще текст, но не в конце, а перед " a link "?

```
var newText = document.createTextNode('our new text');
var ourDiv  = document.getElementById('ourTest');
ourDiv.insertBefore(newText, ourDiv.childNodes[1]);
```

Первая строка показывает, как создать текст с помощью document.createTextNode. Это аналогично функции использованной ранее функции document.createElement. Третья строка содержит еще одну новую функцию insertBefore, которая аналогична appendChild, за исключением того, что имеет два аргумента: добавляемый элемент и существующий элемент, перед которым надо сделать вставку. Так как мы хотим добавить новый текст перед элементом A и знаем, что элемент A является вторым элементом в div, то мы используем ourDiv.childNodes[1] в качестве второго аргумента для insertBefore.

Важный момент: в этом примере скрипт должен быть вызван только ПОСЛЕ загрузки HTML, в противном случае у вас возникнет ошибка `Cannot read properties of null (reading 'insertBefore')` так как ещё не будет прочитано содержимое страницы.

По большей части это все манипуляции с DOM. Если требуется создать, например, поле с изменяемым размером, то для изменения ширины и высоты поля будут использоваться те же функции мыши и функции getAttribute и setAttribute. Очень похожим образом, если изменять верхнюю и левую позицию стиля элемента, то можно перемещать элементы по странице, либо в ответ на ввод мыши (перетаскивание), либо по таймеру (анимация).

Теперь можно включить свое воображение и экспериментировать, так как почти нет ничего такого, чего нельзя сделать со страницей HTML, когда вы знаете, как обращаться с DOM.