

# Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery

top

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# Unit 4.2

## Document Object Model

### Содержание

Что такое Document Object Model?.....	3
Отличия DOM от BOM.....	6
Представление HTML-документа в виде дерева.....	9
Объекты модели DOM. Иерархия узлов .....	18
Свойства и методы модели DOM. Модель событий DOM.....	24
Изменение дерева DOM .....	46
Поиск элементов.....	67
Управление ссылками: объекты Link и Links .....	77
Управление выделением и текстовым диапазоном: объекты Range, Selection и TextRange.....	85
Обобщение сведений об объекте Document .....	103
Домашнее задание.....	110

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

# Что такое Document Object Model?

Вернемся еще раз к основной функции браузера — обрабатывать HTML код и создавать из него веб-страницу. Для того чтобы вникнуть в особенности работы браузера в целом и JavaScript в частности, нам нужно более детально разобраться в том, что из себя представляет эта самая веб-страница.

Изначально, язык HTML разрабатывался как разметочный язык для создания электронных книг, учебников, статей. Основой для работы браузера служили популярные в то время издательские (типографские) системы. Страница собиралась наподобие издательского макета, как страница газеты или журнала. Несмотря на то, что современный язык HTML значительно отличается от своих первых версий, в нем до сих пор существуют некоторые типографские термины, такие как строчно-блочные элементы, заголовки, базисная линия или интерлиньяж ([line-height](#)).

Сейчас от веб-страницы требуется не просто статическая разметка, как у печатной страницы, а полноценный интерактивный интерфейс, реагирующий на действия пользователя, события из сети или операционной системы, временные интервалы и т.п. В дизайне веб-страниц также появились собственные элементы, например, такие как панель навигации и боковая панель. В силу их популярности, для их создания были добавлены отдельные HTML теги, отсутствовавшие в первых версиях — `<nav>`

и `<aside>`, соответственно. На самом деле таких тегов множество, тут просто для примера приведены два из них.

Современную веб-страницу надо не просто «собрать и отобразить», а представить ее в виде программной конструкции, в которой элементы находятся в динамическом, «живом» состоянии — могут взаимодействовать между собой, изменять свои размеры, положение, цвет и другие атрибуты, появляться или скрываться, принимать и проверять данные и так далее. То есть разметочные «блоки» должны превратиться в программные «объекты», которые, с одной стороны, содержат все необходимые данные для их отображения и, с другой стороны, позволяют оперативно, «на лету» поменять эти данные, удалить их или добавить новые. В то же время и сам браузер вместо «сборщика» страницы должен стать «системой управления» контентом, дающей возможность манипулировать данными разных объектов и тут же вносить изменения в их отображение.

Упомянутые в предыдущем уроке тенденции различных производителей создать наилучший браузер (так называемые войны браузеров) приводят к тому, что каждый браузер стараются создать особенным, не таким как все. Сделать собственную систему управления контентом и различный набор ее возможностей. В результате страдает единообразие и правильность одинакового отображения страниц на различных браузерах. Для введения общих правил обработки HTML кода и построения из него страницы, *World Wide Web Consortium* (W3C) подготовил и опубликовал требования, известные нам под названием «объектная модель документа» (*Document Object Model*, DOM).

Что же такое DOM? Это набор требований к тому, как веб-страница должна быть представлена в виде управляемой информационной системы, каким образом элементы HTML должны превратиться в программные элементы (элементы DOM), при помощи каких команд ими можно управлять, на какие события они должны реагировать и так далее. DOM — это стандарт, требующий от различных браузеров соблюдать одни и те же правила обработки HTML-кода, что позволяет разработчикам писать универсальные инструкции не сильно беспокоясь о том, что данная часть кода не будет выполнена на других браузерах.

## Отличия DOM от BOM

Модели DOM и BOM, несмотря на похожие определения и аббревиатуры, имеют существенные отличия. Рассмотренная ранее объектная модель браузера (BOM) является программным «представителем» браузера, включая операционную систему, в которой он работает. В свою очередь, модель документа (DOM) «представляет» саму веб-страницу, открытую в браузере.

Если одна и та же страница будет несколько раз открыта в разных браузерах или даже в разных вкладках одного браузера, то для них:

- а) значения объектов BOM могут отличаться. И, скорее всего, будут отличаться — разные браузеры могут иметь различное название, различные версии. В различных вкладках может отличаться история ранее открываемых страниц,
- б) значения объектов DOM будут одинаковыми во всех браузерах, во всех вкладках. Поскольку в них отображается одна и та же страница, ее параметры просто не могут быть различными<sup>1</sup>.

С точки зрения программной архитектуры, DOM является структурной частью BOM. Основным объектом BOM «[window](#)» является родительским для главного объ-

---

<sup>1</sup> На самом деле небольшие различия все же бывают. Желание сделать свой браузер «не как все» выражается в выходе за стандарты W3C в контексте реализации дополнительных функций, сверх стандартных. Также возможны отличия на техническом уровне, например, в том, как в браузерах представлены разрывы строк или пробелы между тегами. Одинаковыми являются объекты DOM, прямо предусмотренные стандартами.

екта DOM «[document](#)». Это отражает и простую логику: веб-страница открывается в браузере, она не существует параллельно с ним (на одном уровне) и не является более главным элементом. Иерархия именно такая, и программная и логическая, — страница является частью вкладки (окна) браузера.

Основное предназначение BOM — взаимодействовать с браузером и операционной системой, получать данные о размерах окна, состоянии аккумуляторных батарей, положении (геолокации) устройства, а также управлять историей просмотров, переходами между различными адресами страниц и т.д.

Задачи DOM практически никак не касаются самого браузера и нацелены на построение содержимого веб-страницы как связанной совокупности отдельных элементов (блоков, списков, рисунков и т.п.). Изменения, вносимые в DOM, должны сразу приводить к перестроению страницы, в результате чего пользователь их сразу увидит на экране своего устройства.

Есть и общие черты у моделей DOM и BOM. Ключевым для обеих является иерархическая объектная структура. Имеется в виду, что сами модели построены по принципам:

- а) каждый элемент модели является объектом (в программном понимании этого термина),
- б) любой объект может содержать в своем составе произвольное количество других (дочерних) объектов.

Основой каждой из моделей является один объект. В BOM — «[window](#)», в DOM — «[document](#)». В составе каждого из объектов присутствуют другие объекты, ко-

которые, в свою очередь, могут иметь внутреннюю структуру. В итоге схема модели представляет собой некоторое дерево или граф, где от каждого объекта (узла) отходят ветви к дочерним объектам, от них — к своим и т.д.

Деревья DOM и BOM близки по форме (по внешнему виду) — один корневой элемент и сложная ветвистая структура остальных элементов. Этим модели похожи друг на друга. Сами же элементы моделей DOM и BOM отличаются между собой: названием, функциями, наличием и количеством дочерних элементов. В этом заключается разница между ними.



# Представление HTML- документа в виде дерева

Для того чтобы исследовать DOM напомним основные понятия структурного отношения, необходимые для дальнейшей работы. Их основы мы приводили в уроке 1 (раздел «Понятие *Document Object Model*»). Более подробно отношения рассмотрим в следующем разделе урока.

Основным элементом модели является узел (*node*). Дочерние узлы хранятся в его коллекции «*childNodes*», родительский узел доступен через свойство «*parentNode*». Конкретный дочерний узел из коллекции может быть получен из выражения «*childNodes[n]*», где *n* — индекс узла (начало отсчета 0). Количество узлов в коллекции хранит в себе свойство «*childNodes.length*».

Дополнительно к основным определениям отметим, что корневой (самый первый) элемент документа находится в объекте «*documentElement*» главного объекта модели «*document*». То есть его полное имя «*document.documentElement*».

Для того чтобы наглядно увидеть описанные элементы, создадим небольшой HTML документ и исследуем его DOM структуру. Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое: (код также доступен в панели Sources — файл *js4\_4.html*).

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
```

```

<title>HTML DOM structure</title>
<style>
#d1 {
    border: 2px solid navy;
    height: 150px;
    margin: 20px;
    padding: 15px;
    width: 400px;
}
#d2 {
    border: 1px dashed navy;
    float: right;
    height: 140px;
    overflow: auto;
    padding: 5px;
    width: 200px;
}
</style>
</head>
<body>
    <div id="d1">
        <div id="d2"></div>
        <h1>Header</h1>
        <p>paragraph<br><span>Span</span></p>
        <button onclick="getStructure()">Get structure
        </button>
    </div>

    <script>
function getStructure() {
    var c=document.documentElement.childNodes;
    var msg="";
    for(let i=0; i<c.length; ++i) {
        let d = c[i];
        msg += (+i+1) + ". "+d.tagName+" (" +d.nodeName+" )
        <br>";
    }
}

```

```
        window.d2.innerHTML = msg;
    }
</script>

</body>
</html>
```

Основу страницы составляет блок «`<div id="d1">`» в который вложены:

- блок `<div id="d2">`;
- заголовок `<h1>`;
- абзац `<p>`, в который, в свою очередь, вложен блок `<span>`;
- кнопка `<button>`, запускающая функцию «`getStructure()`».

В скриптовой части документа описывается функция «`getStructure()`». Сначала в функции определяется коллекция дочерних элементов корневого элемента «`c = document.documentElement.childNodes`» и сохраняется в переменной «`c`». для формирования сообщения задается переменная «`msg`».

Затем циклом осуществляется обход коллекции, на каждой итерации которого

- определяется очередной элемент коллекции `d = c[i]`;
- определяется имя тега данного элемента (`d.tagName`) и имя узла (`d.nodeName`);
- указанные значения, а также номер итерации добавляются к сообщению «`msg`».

По окончании работы цикла сообщение выводится в блок «`d2`».

В заголовочной части указаны стили для блоков для того чтобы сделать их более заметными.

Сохраните файл, откройте его при помощи браузера. В результате страница должна иметь вид, подобный приведенному на следующем рисунке



Рисунок 1

Нажмите на кнопку «[Get structure](#)». Во внутреннем блоке должно появиться сообщение:

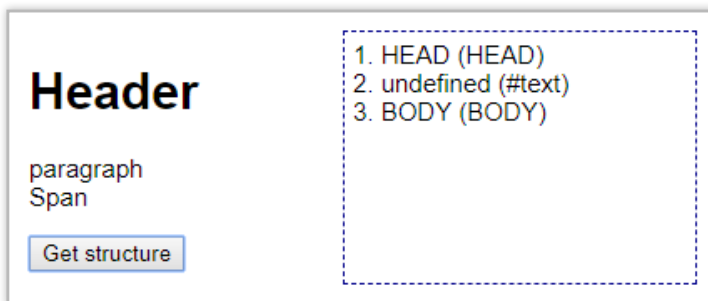


Рисунок 2

Подведем первые итоги. Корневой элемент документа или, проще говоря, сам документ состоит из трех дочерних элементов. Первый элемент отвечает за заголовочную часть (**HEAD**), последний — за тело документа (**BODY**).

Между ними появляется дополнительный элемент, не имеющий имени тега (**undefined**) и представляющий собой текстовый узел (**#text**).

Этот анонимный элемент представляет собой промежуток между объявлениями заголовка и тела. Обратите внимания, что закрывающий тег `</head>` и открывающий `<body>` находятся на разных строках. То есть между ними есть разрыв строки. Именно этот разрыв является дополнительным элементом DOM текстового типа.

Уберите разрыв строки между тегами, чтобы определения шли друг за другом (без пробелов)

```
</head><body>
```

Сохраните файл, обновите страницу в браузере и снова нажмите на кнопку «[Get structure](#)». Сообщение должно измениться на следующее

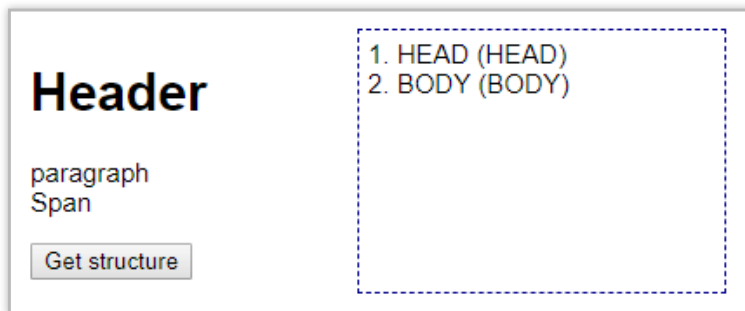


Рисунок 3

Как видим, безымянный текстовый элемент исчез из структуры.

Продолжим исследовать структуру документа. Добавим в цикл обхода коллекции еще один внутренний

цикл: (код с изменениями доступен в папке Sources — файл *js4\_5.html*)

```
function getStructure() {
    var c=document.documentElement.childNodes;
    var msg="";
    for(let i=0; i<c.length; ++i) {
        let d = c[i];
        msg += (+i+1) + ". "+d.tagName+" (" +
            d.nodeName+")<br>";
        if(d.hasChildNodes()) {
            let e=d.childNodes;
            for(let j=0; j<e.length; ++j) {
                let f = e[j];
                msg += "  &nbsp;"+(+j+1)+" "+
                    f.tagName+" (" +f.nodeName+
                    ")<br>";
            }
        }
    }
    window.d2.innerHTML = msg;
}
```

После формирования строки сообщения добавим проверку, является ли данный узел составным объектом (имеет ли он дочерние элементы)

```
if(d.hasChildNodes()) {
```

Если это так, то получаем его дочерние элементы в переменную «e»:

```
let e=d.childNodes
```

И далее полностью аналогично обходим циклом эту коллекцию. В новом цикле используем другое имя для

переменной цикла «j», а также добавляем пробел в начале строки сообщения

```
msg += " &nbsp;"; " + ...
```

для того чтобы наглядно создавался эффект вложенности.

Сохраните изменения, откройте или обновите страницу в браузере. Внешний вид страницы не меняется, т.к. мы не вносили изменений в визуальную часть. Нажмите на кнопку «[Get structure](#)». Во внутреннем блоке должно появиться сообщение:

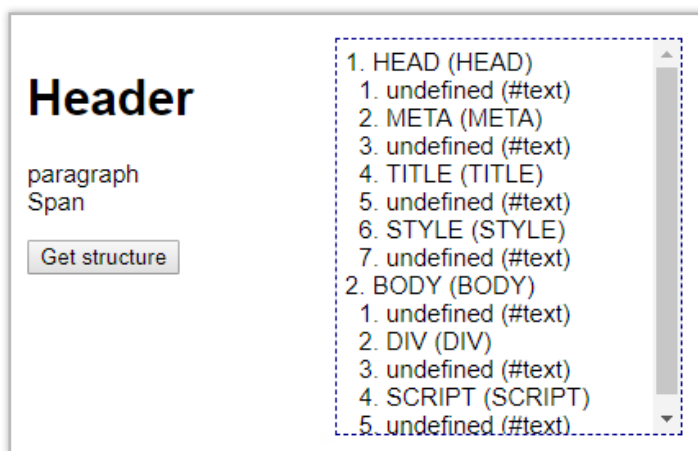


Рисунок 4

По аналогии с предыдущим результатом, делаем выводы о том, что текстовые блоки появляются и в структуре других дочерних блоков. Зная их природу, внесите изменения в HTML код, убрав пробелы и разрывы строк между закрывающими и открывающими тегами. Добейтесь того, чтобы в структуре документа остались только

основные структурные части: (код с изменениями доступен в папке *Sources* — файл *js4\_6.html*)

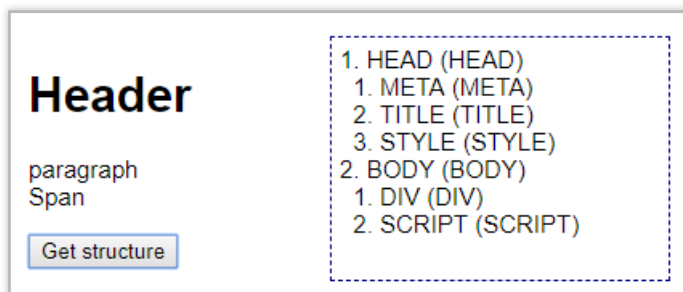


Рисунок 5

Продолжим движение вглубь DOM структуры. Добавим третий цикл, раскрывающий третий уровень иерархии объектов. Постарайтесь самостоятельно его составить (код с изменениями доступен в папке *Sources* — файл *js4\_7.html*). Итогом раскрытия третьего уровня структуры должен быть результат на подобие приведенного на следующем рисунке:

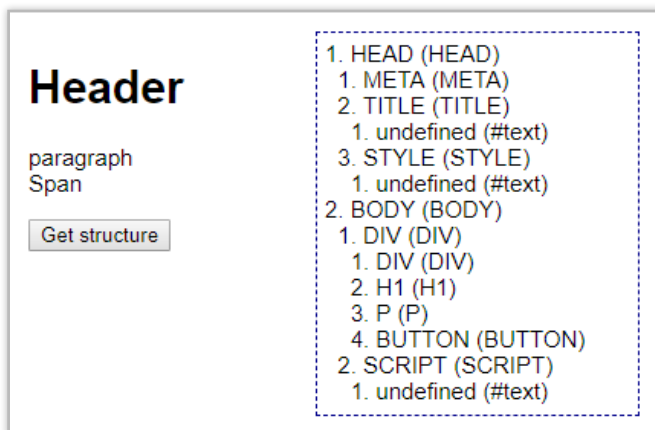


Рисунок 6



Как видно, начинает проявляться внутренняя структура блока «**div**», описанного в теле документа. Анонимные текстовые объекты в узлах «**TITLE**», «**STYLE**» и «**SCRIPT**» — это сами текстовые определения узлов: заголовок документа «**HTML DOM structure**», указанный между тегами `<title>` и `</title>`; тексты стилевых определений и скриптов. Об этом свидетельствует тот факт, что текстовые блоки идут по одному, а не по несколько. Дополнительно, можно убедиться в этом открыв консоль разработчика и добравшись до нужного элемента по иерархии **DOM**:

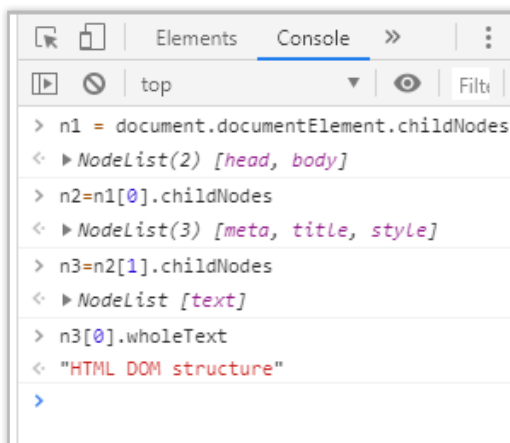


Рисунок 7

Продолжите углубление в структуру документа, раскройте состав параграфа, в котором есть дочерние элементы.

# Объекты модели DOM.

## Иерархия узлов

Вернемся еще раз к примеру, рассмотренному ранее в первом уроке при знакомстве с принципами DOM. Повторим его и разберем более детально.

Фрагмент документа представляет собой маркированный (нумерованный) список состоящий из трех пунктов. Второй пункт также имеет собственную структуру. HTML код списка выглядит следующим образом:

```
<ul>
  <li>first element</li>
  <li> second element
    <span>child Node 0</span>
    <a>child node 1</a>
    <p> child  node 2 </p>
  </li>
  <li>third element</li>
</ul>
```

Согласно принципам DOM этот список будет иметь следующую структуру<sup>1</sup> (рис. 8).

В отношениях между узлами модели можно выделить три уровня:

- Родительский узел (**parentNode**) позволяет двигаться по «дереву» структуры вверх

<sup>1</sup> На самом деле, как мы уже убедились в предыдущих упражнениях, структура приведенного HTML кода будет значительно объемнее, включая в себя анонимные текстовые блоки, отвечающие за форматирование исходного кода. Для того чтобы получить структуру как на рисунке, из кода нужно удалить лишние пробелы, табуляции и разрывы строк.

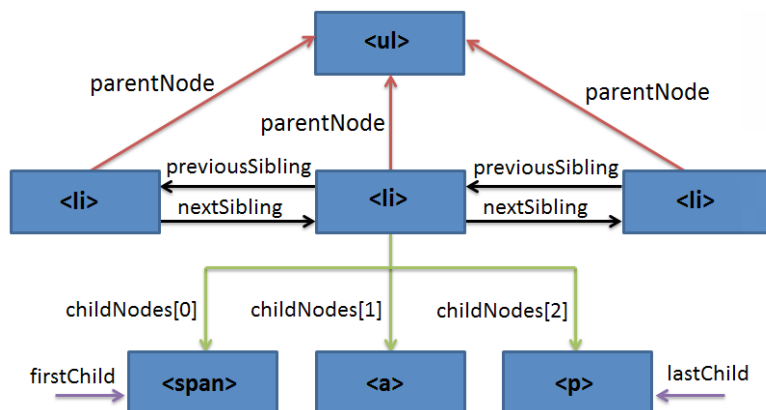


Рисунок 8

- Соседние узлы (**siblings**) отвечают за элементы одного уровня иерархии. Переход между такими узлами обеспечивается свойствами «**previousSibling**» и «**nextSibling**»
- Дочерние узлы представляют движение по дереву вниз. Для каждого узла существует коллекция дочерних узлов (**childNodes**). Обход этой коллекции возможен двумя способами: 1) при помощи индексов коллекции, наподобие **childNodes[1]**, либо 2) при помощи указателей первого и последнего дочернего элемента (**firstChild** и **lastChild**), а также переходов к последующему или предыдущему соседнему узлу (**previousSibling** и **nextSibling**)

Для практического исследования отношения между узлами создайте новый html-файл и скопируйте в него определение списка, приведенное выше. Удалите из него все лишние пробелы и разрывы строк (код с необходимыми изменениями доступен в папке Sources — файл *js4\_8.html*).

Сохраните файл и откройте его в браузере, откройте консоль разработчика. Мы уже знаем, что корневым элементом документа выступает объект «`document.documentElement`». Его дочерними элементами являются заголовок (**HEAD**) и тело (**BODY**) документа. Нас интересует тело, поэтому введем в консоль следующую инструкцию:

```
body=document.documentElement.childNodes[1]
```

Этим мы создадим переменную «`body`», в которой будет сохранен узел, отвечающий за тело. Убедитесь, что после ввода инструкции в консоли появится ответ в виде тега `<body>`. Запросим состав коллекции дочерних элементов сохраненной переменной. Напишем в консоли инструкцию

```
body.childNodes
```

В качестве ответа в консоли появится коллекция, состоящая из одного элемента «`ul`»:

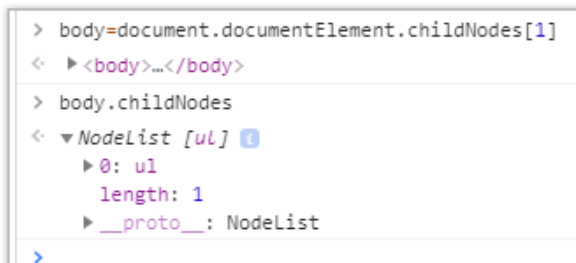


Рисунок 9

Если у Вас коллекция содержит дополнительные текстовые элементы, значит не все лишние пробелы или разрывы строк были удалены из html-кода.

Продолжим движение дальше. Убедившись, что в коллекции «`body.childNodes`» список имеет индекс «0», извлечем его из коллекции и сохраним в отдельной переменной. Введите в консоль инструкцию

```
ul=body.childNodes[0]
```

Аналогично предыдущим командам, запросим состав его коллекции дочерних элементов инструкцией

```
ul.childNodes
```

В результате мы должны получить коллекцию из трех элементов, отвечающих за три пункта списка:

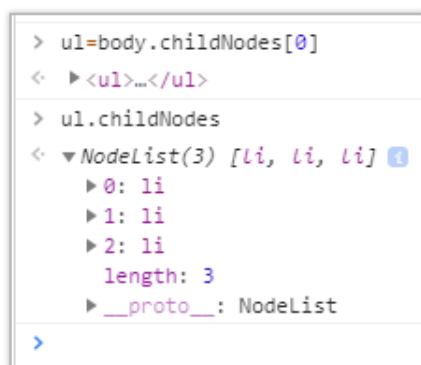


Рисунок 10

Если в коллекции есть дополнительные текстовые элементы, то значит не все лишние пробелы были удалены из кода.

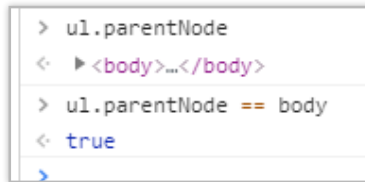
Теперь мы можем проверить отношения с родительским узлом. Наберите в консоли запрос

```
ul.parentNode
```

В качестве ответа увидим тег `<body>`. Поскольку тело документа у нас сохранено в отдельной переменной, мы можем выполнить проверку на равенство:

```
ul.parentNode == body
```

Убедитесь, что данное равенство выполняется (ответ в консоли «`true`»)



```
> ul.parentNode
< > <body>...</body>
> ul.parentNode == body
< true
>
```

Рисунок 11

Для проверки отношений соседского типа, перейдем к элементу, который имеет таких соседей — к пункту списка `<li>`. В качестве альтернативы индексам воспользуемся свойством «`firstChild`» списка. Введите в консоль инструкцию

```
l1 = ul.firstChild
```

Ответом будет элемент `<li>first element</li>`. Запросим у него соседний элемент:

```
l1.nextSibling
```

В ответ получим второй элемент списка. Можем раскрыть подробности ответа и убедиться в этом. Запрос следующего соседа можно выполнить в каскадном стиле:

```
l1.nextSibling.nextSibling
```

Убедитесь, что в качестве ответа мы получим третий элемент списка.

```
> ul.firstChild
< <li>first element</li>

> l1 = ul.firstChild
< <li>first element</li>

> l1.nextSibling
< ▼<li>
    " second element "
    <span>child Node 0</span>
    <a>child node 1</a>
    <p> child node 2 </p>
  </li>

> l1.nextSibling.nextSibling
< <li>third element</li>

>
```

Рисунок 12

Попробуйте самостоятельно составить инструкции, запрашивающие:

- последний дочерний элемент в списке,
- предыдущего соседа для последнего дочернего элемента списка,
- родительский элемент для переменной «**l1**» (проверьте, что он равен «**ul**» и не равен «**body**»).

# Свойства и методы модели DOM. Модель событий DOM

Модель документа, кроме введения правил расположения его составных элементов и отношений между ними, должна предусматривать возможности внесения изменения, а также реагирование на определенные события. Если собрать воедино все требования, которые должны быть сформулированы при построении документа, получится следующая схема, утвержденная стандартом:

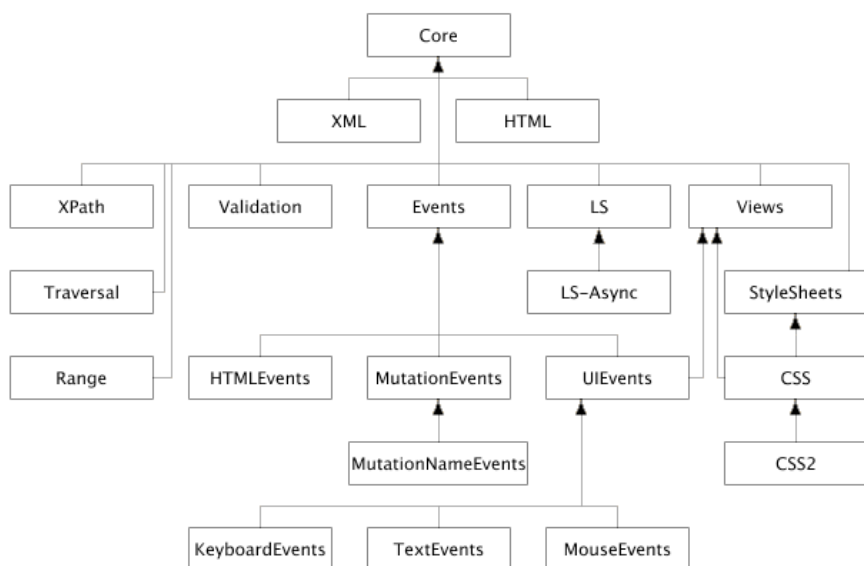


Рисунок 13

Как видно, даже сама схема свойств и методов модели DOM является достаточно сложной и объемной. Для того



чтобы понимать предназначение различных элементов нужно хорошо разбираться в программировании и понимать механизмы, происходящие при работе браузера и операционной системы. Для общего знакомства приведем краткий обзор составных элементов модели, не вдаваясь в технические подробности:

- **Core** — основные определения — константы, коды исключений, функции, объекты, например, определение основного объекта модели «**document**»;
- **HTML** — определения для языка HTML;
- **XML** — определения для языка XML, альтернативного языка разметки, применяющегося для создания веб-ресурсов, а также для хранения и передачи данных в текстовой форме;
- **Validation** — средства проверки корректности при внесении изменений в документе;
- **XPath** — альтернативные средства управления документом, ориентированные на XML;
- **Traversal** — дополнительный (необязательный) модуль для древовидного/итеративного представления документа;
- **Range** — набор средств для «логического выделения» — заключения элементов документа между двумя точками-маркерами;
- **LS (Load and Save)** — набор средств для загрузки, фильтрации и сохранения XML объектов;
- **Views** — средства управления видом (видами) документа;
- **StyleSheets** — интерфейс для представления стилей элементов документа;

- **CSS (Cascade Style Sheets)** — набор правил (декларативный синтаксис) для каскадного определения стилей элементов и их групп;
- **Events** — платформи- и языконезависимая система регистрации, передачи и обработки событий.

Остановимся на событиях более подробно, поскольку они представляют основу для взаимодействия веб-страницы с пользователем и программным окружением.

Еще с появлением первых операционных систем с графическим интерфейсом появилось понятие «событийно-ориентированного программирования». Смысл его заключается в том, что каждый объект имеет в своем составе специальные функции (методы), запускаемые при наступлении определенных событий — так называемые «обработчики событий». Операционная система и, в нашем случае, браузер управляет событиями, собирает их в очередь и отправляет их тем объектам, которым они предназначены, запуская у них соответствующие обработчики событий.

Мы привыкли к тому, что кнопки на странице «нажимаются» и выполняют определенные действия. На самом же деле кнопки — это просто рисунки на экране монитора. Когда пользователь нажимает на клавишу мыши операционная система формирует соответствующее событие — «левая клавиша мыши нажата в координатах  $x=151$ ,  $y=418$  экрана монитора». Для упрощения обработки событий им даются имена, например «**MouseDown**».

Далее она определяет, для какого приложения предназначено это событие. Ведь у пользователя может быть

открыто много приложений одновременно, их окна могут находиться одно за другим, и координаты  $x=151$ ,  $y=418$  на экране могут принадлежать сразу нескольким из них. Системе нужно определить активное окно — находящееся «сверху» других. Затем событие попадает в конкретное приложение. Пусть в нашем случае это будет браузер с некоторой веб-страницей.

Эта веб-страница, в свою очередь, тоже может иметь достаточно сложную структуру — одни объекты перекрываются с другими, какие-то находятся в отношении подчинения, другие полностью вмещаются в некоторые «контейнеры», в том числе в невидимые или прозрачные. Снова возникает задача, какому объекту следует адресовать полученное от системы событие.

На данном этапе включается принцип «всплытия» событий (англ. — *propagation*). Сначала событие адресуется самому верхнему элементу в данной точке окна. Если этот элемент содержит в себе обработчик для данного события, то он запускается и событие считается обработанным, то есть исчезает из очереди событий. Если же у верхнего элемента нет обработчика, то событие «всплывает» — передается следующему элементу, находящемуся непосредственно за самым верхним.

Снова проверяется наличие обработчика. Если он есть — событие обрабатывается, если нет — всплывает дальше к следующему элементу. Если ни у одного элемента не находится обработчика, событие передается последнему элементу (*body*) и дальше убирается из очереди как необработанное, если и в последнем элементе не нашлось обработчика.

Понимание процесса всплытия событий для веб-разработки крайне важно, поскольку при проектировании веб-страниц очень часто одни блоки включаются в другие с неоднократной вложенностью. Вполне возможна ситуация, когда событие, предназначенное для определенного элемента, будет обработано более «верхним» элементом и просто не дойдет до адресата. Чуть ниже мы продемонстрируем это на примере.

В JavaScript обработчики событий можно создать двумя способами. Покажем их на примере создания блока-кнопки, который реагирует на клик левой кнопки мыши (нажатие и отпускание). Данное событие имеет имя «[click](#)».

Первый способ заключается в указании атрибута при объявлении элемента в HTML коде. Создайте новый HTML-документ, наберите или скопируйте в него следующий код (код также доступен в папке *Sources* — файл *js4\_9.html*).

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Events</title>
  <style>
    div {
      background-color:tomato;
      color:azure;
      height:23px;
      padding-top:3px;
      text-align:center;
      width:75px;
    }
  </style>
```

```
</head>

<body>
  <div onclick="alert('Click event handled')">
    Press me
  </div>
</body>

</html>
```

В теле документ создается блок `<div>`, в котором указывается атрибут «`onclick`». Значением данного атрибута является фрагмент кода, запускающего диалоговое окно-сообщение «`alert`», чем подтверждая факт получения и обработки события.

В заголовочной части документа задаются стили для блока, делая его похожим на кнопку.

Сохраните документ, откройте его в браузере. Результат должен быть подобный приведенному на рисунке:

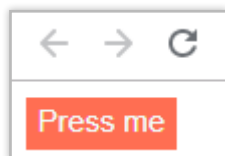


Рисунок 14

Наведите указатель мыши на блок и щелкните левой клавишей. В результате должно появиться сообщение «`Click event handled`».

Второй способ создать обработчик события — это вызов специального метода «`addEventListener`» у нужного элемента.

Внесите изменения в созданный файл:

- Вместо атрибута «**onclick**» укажите идентификатор объекта «**id="btn"**».
- Создайте раздел `<script>` в теле документа
- В этом разделе впишите инструкцию `btn.addEventListener("click",function(){ alert('Click event handled') })`

В заголовочную часть изменений вносить не нужно. Тело документа примет следующий вид (код с необходимыми изменениями доступен в папке *Sources* — файл *js4\_10.html*).

```
<body>
  <div id="btn">Press me</div>
  <script>
    btn.addEventListener("click",function(){
      alert('Click event handled') })
  </script>
</body>
```

Сохраните документ и откройте его в браузере или обновите уже открытую страницу. Внешний вид и поведение не должно измениться.

Разберем приведенные способы создания обработчиков более подробно. Сразу отметим, что способы являются эквивалентными и реализуют одинаковую функциональность.

Первый способ делает HTML код более читаемым. Мы сразу видим какие события для каких элементов будут обрабатываться. Блоки кода становятся самодостаточными для анализа — нет необходимости искать дополнительные коды в других блоках.

Если же все обработчики будут определены в отдельном блоке кода при помощи метода «`addEventListener`» (вторым способом), то при анализе кода нам придется сверять HTML разметку и инструкции в этом блоке. Хуже, если такие инструкции не собраны в один блок, а распределены по различным скриптам. В таком случае читаемость кода значительно ухудшается.

С другой стороны, второй способ позволяет разделить работу дизайнеров и программистов в больших проектах. Для того чтобы определить обработчик в атрибутах HTML элемента программисту нужно вмешиваться в файлы, созданные дизайнерами. Если в дальнейшем потребуются дизайнерские правки, то фрагменты кода могут быть повреждены или удалены, за счет использования шаблонов разметки. Снова потребуется вмешательство программистов.

С этой точки зрения гораздо удобнее вынести программную часть в отдельный файл-скрипт и работать независимо от стилевых и разметочных правок. К тому же отделение файлов может ускорить повторные загрузки страницы, так как при первой загрузке файлы будут сохранены в кеш браузера и в дальнейшем будут использоваться уже сохраненные копии.

Подытоживая, можем отметить, что первый способ лучше соответствует принципу модульности, то есть позволяет создавать относительно универсальные блоки (модули), которые можно скопировать в другие проекты, так как все необходимые определения заключены внутри этих блоков. Этот способ подходит для небольших проектов или их частей, предназначенных для копирования в другие проекты.

Второй способ является предпочтительным для больших проектов, над которыми работают группы разработчиков. Также этот способ упрощает оптимизацию сайта в контексте отделения файлов, которые крайне редко изменяются и могут кешироваться браузерами.

Вернемся к модели событий и рассмотрим более детально процесс их обработки. Как уже было отмечено, при появлении события у объекта запускается соответствующий обработчик. В то же время, в обработчик передается специальный объект типа «**event**», содержащий в себе данные о событии. Если подробные данные в обработчике не нужны, они могут быть просто проигнорированы. Если же такая необходимость есть, при определении обработчика следует указать параметр функции, через который данные о событии будут в нее переданы.

Исследуем передачу и обработку данных о событии на примере событий «мыши». Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое: (код также доступен в папке Sources — файл *js4\_11.html*)

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Mouse events</title>
</head>

<body>
  X: <span id="coordX"></span><br>
  Y: <span id="coordY"></span>
  <script>
```



```
document.onmousemove =  
function(e) {  
    coordX.innerHTML = e.pageX;  
    coordY.innerHTML = e.pageY;  
}  
document.onmousedown =  
function() {  
    coordX.style.fontSize =  
    coordY.style.fontSize = "x-large";  
}  
document.onmouseup =  
function() {  
    coordX.style.fontSize =  
    coordY.style.fontSize = "medium";  
}  
</script>  
</body>  
</html>
```

Во-первых, обратите внимание на альтернативный способ определения обработчиков событий. Мы не рассматривали его в сравнительном анализе, поскольку он по сути совпадает с методом «[addEventListener](#)» (вторым способом), определяя обработчики в отдельной секции, а не в самом HTML теге. Тем не менее, для установки обработчика используются те же атрибуты, которые можно было бы указать в теге: «[onmousemove](#)», «[onmousedown](#)» и «[onmouseup](#)». Как несложно догадаться, эти обработчики отвечают за движение мыши, нажатие и отпускание клавиш на ней (соответственно).

Во-вторых, отметьте отличия в определении обработчиков: «[onmousemove](#)» определяется как функция с параметром «[function\(e\)](#)», тогда как два других обработчика

параметры не декларируют «`function()`». Таким образом в обработчике движения мыши мы используем дополнительные данные о событии, а в обработчиках нажатия и отпускания кнопки — игнорируем их.

Далее, в обработчике «`onmousemove`» определяются значения полей «`e.pageX`» и «`e.pageY`» принятого параметра «`e`». Они отвечают за координаты указателя мыши на экране браузера. Отмеченные значения помещаются в поля «`innerHTML`» текстовых блоков «`coordX`» и «`coordY`», созданных выше тегами «`span`». В результате чего значения координат должны отображаться на экране.

Сохраните файл и откройте его в браузере. Наведите указатель мыши на окно браузера и убедитесь в обновлении координат при движениях указателя.

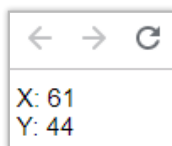


Рисунок 15

Обработчики клавиши мыши управляют стилями блоков «`coordX`» и «`coordY`». При нажатии клавиши размер шрифта увеличивается, при отпускании уменьшается до исходного. Убедитесь в правильности их работы.

Также обратите внимание на то, что события приходят от всех клавиш мыши, в том числе при нажатии на колесо (если оно нажимается). При этом после отпускания правой клавиши дополнительно открывается контекстное меню браузера.

Для того чтобы определить различные действия для различных клавиш мыши необходимо анализировать параметры события. Для этого в определении обработчика следует указать параметр «**function(e)**». В событии мыши за номер клавиши отвечает поле «**which**»: 1 — левая, 2 — средняя (или колесо), 3 — правая клавиши. Если мы хотим, чтобы действие запускалось только левой клавишей, нужно внести следующие изменения в определение обработчика

```
document.onmousedown =  
    function(e) {  
        if (e.which==1) {  
            coordX.style.fontSize =  
            coordY.style.fontSize = "x-large";  
        }  
    }
```

Условный оператор «**if(e.which==1)**» ограничит выполнение инструкций только для левой клавиши.

Для того чтобы изменить поведение правой клавиши мыши необходимо вмешаться в еще одно событие — вызов контекстного меню «**oncontextmenu**». Добавьте еще один обработчик события со следующим содержанием:

```
document.oncontextmenu =  
    function(e) {  
        e.preventDefault()  
    }
```

Единственной инструкцией функции является команда «**e.preventDefault()**», которая останавливает обработку события, установленную по умолчанию. В таком случае контекстное меню на экране появляться не будет.

Убедитесь в этом, обновив страницу в браузере после внесения изменений в коды.

Остановимся более подробно на эффекте всплытия событий. Для его иллюстрации рассмотрим следующий пример. Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое: (*код также доступен в папке Sources — файл js4\_12.html*)

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Events propagation</title>
  <style>
    #div1 {
      background-color:tomato;
      border:1px solid red;
      height:100px;
      width:100px;
    }
    #div2 {
      background-color:lime;
      height:60px;
      width:60px;
      margin:20px;
    }
  </style>
</head>

<body>
  <div id="div1"><div id="div2"></div></div>
  <script>
    div1.addEventListener("click",function(){
      console.log('Click on div1') })
    div2.addEventListener("click",function(){
      console.log('Click on div2'); })
```

```
document.addEventListener("click",function(){  
    console.log('Click on document') })  
</script>  
</body>  
</html>
```

На странице создаются два блока «**div1**» и «**div2**» с разным фоновым цветом для наглядности. Блоки вложены один в другой. Внутреннему блоку заданы отступы для размещения по центру внешнего блока.

В скриптовой части для них устанавливаются обработчики события щелчка мышью «**click**» в которых формируется сообщение в консоль браузера. Полностью аналогичный обработчик устанавливается и для корневого объекта «**document**».

Сохраните файл и откройте его в браузере. Откройте консоль разработчика для контроля сообщений о событиях. Совершите щелчок по документу (не по блоку). Убедитесь в том, что в консоли появляется сообщение. Далее щелкните по внешнему блоку и обратите внимание на то, что в консоль приходит два сообщения — от блока и от документа. Щелкните по внутреннему блоку — сообщений приходит три: от двух блоков и документа

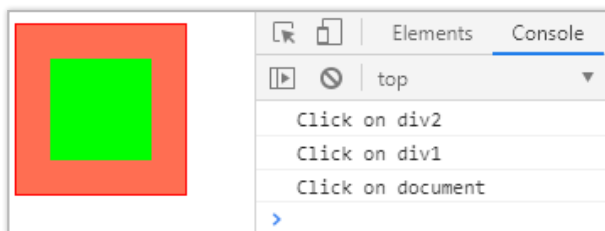


Рисунок 16

Появление нескольких сообщений связано со всплытием событий — процессом последовательной передачи их всем объектам, которые находятся под курсором мыши — сначала внутреннему блоку, затем внешнему блоку, затем документу.

Если неправильно распределить обработчики событий, то такая ситуация может привести к многократному срабатыванию некоторых функций, за счет вызова нескольких обработчиков при одном событии. Для управления процессом всплытия у событий предназначен метод «`stopPropagation`». При его вызове прекращается всплытие и событие далее не передается.

Замените определение обработчика события для первого блока на следующее:

```
div1.addEventListener("click",function(e){  
    console.log('Click on div1'); e.stopPropagation();})
```

Во-первых, для получения объекта-события, указываем параметр для функции «`function(e)`».

Во-вторых, останавливаем дальнейшее всплытие данного события добавляя инструкцию «`e.stopPropagation()`».

Сохраните изменения, обновите страницу браузера. Повторите действия по созданию событий-щелчков вне блоков, по внешнему и по внутреннему блокам.

Поскольку блок «`div1`» останавливает всплытие событий, после него они до документа не доходят. Блок «`div2`» всплытие не останавливает, поэтому после него событие передается родительскому блоку и в нем уже всплытие отменяется. Щелчок вне блоков обрабатывается так же, как и раньше.

Добавьте команды, останавливающие всплытие события, для внутреннего блока. Убедитесь, что оно перестанет передаваться как внешнему блоку, так и документу.

Для закрепления материала по модели событий создадим страницу, реализующую технологию «Drag-and-Drop» — перемещения элементов при помощи мыши. Главной особенностью реализации данной технологии является то, что события нажатия и отпускания кнопок мыши, а также события ее движения обрабатываются в трех разных функциях-обработчиках. Необходимо определенным образом синхронизировать их работу. Алгоритм их взаимодействия будет заключаться в следующем:

- обмен данными между различными функциями можно обеспечить при помощи глобальных переменных, видимых во всех функциях нашей страницы. Создадим такую переменную с именем «`isDrag`» и установим для нее начальное значение «`false`»
- при нажатии кнопки мыши на объекте, который перетягивается, переменная «`isDrag`» будет установлена в значение «`true`»
- при отпускании кнопки мыши переменной «`isDrag`» будет восстановлено первоначальное значение «`false`»
- при движении мыши проверяем значение переменной «`isDrag`»: если оно «`true`», то это означает, что нужно перемещать наш объект в заданную позицию вместе с курсором мыши.

Обработчик нажатия кнопки мыши, очевидно, должен принадлежать тому объекту, который будет переме-

щаться. Если кнопка нажимается над другим объектом, «включать» перемещение данного объекта не нужно. Что же касается двух других событий, то их принадлежность не так очевидна.

События мыши создаются системой не в каждой точке, которую проходит курсор мыши, а через некоторые интервалы времени. При медленном движении мыши это практически незаметно, но при резких быстрых движениях расстояние между точками от двух последовательных событий может быть существенным. Если этого расстояния будет достаточно для того, чтобы курсор мыши вышел за пределы объекта, то объект перестанет получать сообщения о движении мыши и сам перестанет двигаться. Отпустив кнопку за пределами объекта мы также не запустим его обработчик, отменяющий перемещение. В итоге объект остановится, а при возврате мыши в его область снова начнет перемещаться, хотя кнопка уже была отпущена.

Для решения этих проблем можно воспользоваться всплытием событий. Мы уже убедились, что события получают все элементы, находящиеся в данной точке экрана, в том числе и сам «экран» — корневой элемент «document». Ведь, с точки зрения логики нашей задачи, если кнопка мыши была отпущена, то процесс перемещения должен быть остановлен независимо от того, над каким объектом отпустили кнопку. Значит, это событие можно обработать в последнем объекте «document». В нем же можно реализовать обработчик для события движения мыши, что предотвратит эффект потери управления при резких ее рывках.



Таким образом, событие нажатия кнопки мыши должно обрабатываться внутри объекта, а события движения и отпускания кнопки — в объекте «[document](#)».

Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое: (код также доступен в папке *Sources* — файл *js4\_12.html*)

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Drag-and-Drop</title>
  <style>
    body{
      position: relative;
    }
    #div1 {
      background-color: tomato;
      border: 1px solid red;
      border-radius: 50%;
      height: 100px;
      width: 100px;
      position: absolute;
    }
  </style>
</head>
<body>
  <div id="div1"></div>
  <script>
    var isDrag = false;
    div1.addEventListener("mousedown",
      function(){ isDrag = true })
    document.addEventListener("mouseup",
      function(){ isDrag = false })
    document.addEventListener("mousemove",
      function(e) {
```

```

        if (isDrag) {
            div1.style.left = e.pageX + "px";
            div1.style.top  = e.pageY + "px";
        }
    })
</script>
</body>
</html>

```

Реализация обработчиков событий соответствует описанному выше алгоритму. Как несложно догадаться из кода, событие нажатия кнопки мыши имеет название «**mousedown**», отпускания кнопки — «**mouseup**», движения мыши — «**mousemove**».

Для перемещения блока по странице использовано абсолютное его позиционирование. У тела при этом должно быть указано относительное позиционирование.

В обработчике события «**mousemove**» стилевым атрибутам блока «**left**» и «**top**» задаются значения согласно координатам курсора мыши «**e.pageX**» и «**e.pageY**», переданным через аргумент. По формализму CSS к значениям атрибутов должны быть добавлены единицы измерения, что обеспечивается инструкциями «+ "px"»

Сохраните файл и откройте его в браузере. На пустой странице должен отображаться один блок в виде круга. Такая форма блока позволит нам исследовать дополнительный вопрос. Круглый вид достигается установкой стилового атрибута «**border-radius**», который не влияет на исходные размеры блока (ширину и высоту), и блок все так же остается квадратным, только с закрашенной круглой областью. Суть вопроса заключается в следую-

щем: попадают ли события нажатия кнопки мыши в блок, если они происходят внутри квадрата, но снаружи закрашенной области?

Попробуйте нажать кнопку мыши внутри закрашенной области и, не отпуская кнопку, переместить мышь. Блок сдвинется, следуя за движениями курсора мыши. Попробуйте нажать на не закрашенную область блока и подвигать мышью, убедитесь, что в этом случае блок события не получает.

Для того чтобы убедиться в том, что блок все же остается квадратным обратите внимание на положение блока относительно курсора мыши при его перетаскивании. Курсор выходит за пределы круга и находится в углу описывающего его квадрата:

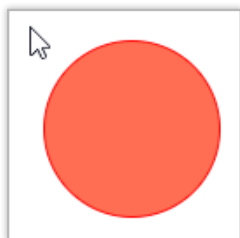


Рисунок 17

Из этого следует, что инструкции установки левой и верхней координаты блока применяются к полному блоку — квадрату. Тем не менее события блок принимает только от закрашенной области. Запомните полученные выводы, они понадобятся при разработке более сложных веб-страниц.

Разобравшись с особенностями отображения и обработки блоков нетривиальной формы, отметим, что

реализованное перетягивание (*Drag-and-Drop*) никак нельзя признать красивым — хотя блок и следует за указателем мыши, но как-то сбоку от него. Хотелось бы, чтобы курсор оставался на видимой части блока, а не выходил за ее пределы.

Внесем изменения в обработчик события движения мыши «`mousemove`». Инструкции позиционирования перепишем в виде

```
div1.style.left = (e.pageX-50) + "px";  
div1.style.top  = (e.pageY-50) + "px";
```

Поскольку размер блока **100x100** пикселей, смещение на 50 пикселей по каждой координате должно соответствовать центру блока.

Сохраните измененный файл и обновите страницу браузера. Нажмите на блок и потащите его. Обратите внимание на положение курсора мыши — он должен быть в районе центра круга (события движения мыши, как уже отмечалось выше, приходят не на каждый пиксель экрана, из-за чего возможны отклонения курсора от центра круга).



Рисунок 18

Такое поведение значительно лучше, однако и его можно улучшить. В реализованном способе круг перетя-

гивается за центр независимо от того, в какой начальной точке была нажата кнопка мыши. Это создает эффект рывка в начале движения, особенно если начальное положение курсора мыши далеко от центра круга.

Для того чтобы нивелировать это явление необходимо запоминать точку отсчета — начальные координаты мыши при первом нажатии (в обработчике события «[mousedown](#)») и в дальнейшем при движении мыши вычитать из ее координат не фиксированные числа «50», а запомненные значения первоначального сдвига. Реализуйте это задание самостоятельно.

## Изменение дерева DOM

Как уже отмечалось выше, при анализе HTML кода браузер выстраивает древовидную структуру программных объектов, называемую «деревом DOM». Мы рассмотрели, как можно анализировать эту структуру, обращаться к соседним, родительским или дочерним элементам, получать их коллекции. В то же время дерево DOM допускает изменения — добавление или удаление его узлов. Рассмотрим эти процессы более детально.

Для внесения изменений в структуру дерева DOM предусмотрены следующие функции:

- `removeChild(e)` — удаляет дочерний узел «`e`», переданный как аргумент функции;
- `appendChild(e)` — добавляет дочерний узел «`e`» в конец существующей коллекции дочерних узлов;
- `insertBefore(e1, e2)` — вставляет узел «`e1`» в коллекцию дочерних элементов перед узлом «`e2`».

Эти функции присутствуют в каждом узле дерева DOM и позволяют оперировать с собственной коллекцией дочерних элементов. Если необходимо добавить или удалить элемент внутри дочернего элемента, следует перейти к нему и вызывать нужный метод у дочернего элемента. Аналогично, для управления соседями следует перейти к родительскому элементу и вызывать его методы управления дочерними элементами.

Для того чтобы добавить в дерево новый элемент, его необходимо сначала создать. Создаются новые узлы

при помощи метода «`createElement`», принадлежащего объекту «`document`»:

```
document.createElement (tagName)
```

В качестве аргумента метод принимает имя тега для элемента, который будет создан. Имя указывается без угловых скобок «`<>`». Например, создать новый абзац (HTML тег `<p>`) можно командой

```
document.createElement ("p")
```

Аналогично, для того чтобы создать блок (HTML тег `<div>`) следует применить команду

```
document.createElement ('div')
```

Имена тегов при вызове метода заключаются в кавычки. Вид кавычек, согласно стандартов JavaScript, роли не играет.

Продemonстрируем методы добавления узлов к дереву DOM на следующем примере. Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое: (код также доступен в папке *Sources* — файл *js4\_13.html*)

```
<html>
<head>
</head>
<body>
  <ul id='list'>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
```

```

        <li>Item 4</li>
    </ul>
    <button onclick="addItem()">Add item</button>
    <button onclick="insertItem()">Insert item</button>

    <script>
        function addItem(){
            var newItem = document.createElement('li');
            newItem.innerText = "New item";
            list.appendChild(newItem);
        }

        function insertItem(){
            var firstItem = list.childNodes[0];
            var newItem = document.createElement('li');
            newItem.innerText = "New item";
            list.insertBefore(newItem, firstItem);
        }
    </script>
</body>

</html>

```

Основу страницы составляет маркированный список `<ul id='list'>` с четырьмя пунктами, названными **Item 1-4**. После списка расположены две кнопки для демонстрации двух различных способов добавления новых элементов.

Нажатие первой кнопки вызывает функцию «**addItem()**». В теле этой функции происходит следующее:

1. Создается новый элемент с именем тега `<li>` (именно такие элементы являются дочерними для списка) и помещается в переменную «**newItem**»: `var newItem = document.createElement('li');`



2. Для нового элемента устанавливается текст при помощи инструкции: `newItem.innerText = "New item";`
3. Новый элемент добавляется в коллекцию дочерних элементов списка. Напомним, что список доступен по имени своего идентификатора «`id='list'`»: `list.appendChild(newItem)`

Подобным образом работает и вторая функция «`insertItem()`», добавляющая новый элемент в начало списка при помощи метода «`insertBefore`». Поскольку для этого метода необходимо два аргумента, дополнительно определяется первый дочерний элемент списка путем обращения к коллекции дочерних элементов списка:

```
var firstItem = list.childNodes[0]
```

При вызове метода «`insertBefore`» указывается, что новый элемент необходимо вставить перед первым, то есть в начало списка.

Сохраните файл и откройте его в браузере. Нажимая на кнопки убедитесь, что новые элементы появляются как в начале, так и в конце списка.

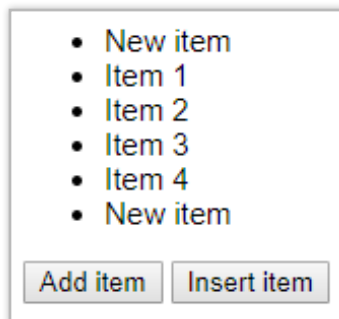


Рисунок 19

При рассмотрении процесса удаления элементов из дерева DOM следует снова вспомнить об особенностях сборки этого дерева. Казалось бы, что нет ничего сложного и второй элемент из списка можно удалить одной инструкцией

```
list.removeChild( list.childNodes[1] )
```

Эта инструкция действительно удалит вторую строку списка, если

1. Второй элемент у списка вообще существует;
2. В списке нет дополнительных элементов, сбивающих нумерацию коллекции.

Условие 1 несложно проверить. Из первого урока мы знаем о том, что все неопределенные переменные имеют тип «**undefined**» и проверка наличия второго элемента в коллекции выглядит следующим образом:

```
if(typeof list.childNodes[1] !=="undefined"){...}
```

Условие 2 возвращает нас к выводам, полученным в разделе 3 данного урока: в коллекции дочерних элементов списка, кроме элементов **<li>** могут присутствовать анонимные текстовые элементы, связанные с оформлением его HTML кода. Сложности добавляет тот факт, что эти элементы могут и не присутствовать, если оформление кода поменяется. То есть для того чтобы получить второй элемент списка (как мы его видим на странице) необходимо перебирать коллекцию его дочерних элементов и считать только те, которые отвечают за тег **<li>**. Остальные игнорировать.

Добавим к HTML коду нашей страницы еще одну кнопку, укажем функцию «[removeItem](#)» в качестве обработчика события ее нажатия

```
<button onclick="removeItem()" ">  
    Remove second item  
</button>
```

В скриптовой части документа добавим эту функцию (код с изменениями доступен в папке *Sources* — файл *js4\_5.html*):

```
function removeItem(){  
    var n = 0;  
    var element2 = false;  
    for(var element of list.childNodes){  
        if(element.tagName == "LI") n++;  
        if(n==2) {  
            element2 = element;  
            break;  
        }  
    }  
    if(element2) list.removeChild(element2);  
}
```

В начале функции устанавливаем две переменные: счетчик строк — «[n](#)» и сам элемент, отвечающий за вторую строку «[element2](#)». Далее организовываем цикл «[for-of](#)», проходящий все элементы коллекции «[list.childNodes](#)».

В теле цикла проверяем отвечает ли данный элемент коллекции за тег «[<li>](#)». Для этого используем свойство «[tagName](#)», определяющее имя тега. Согласно спецификации JavaScript это имя хранится в верхнем регистре

(большими буквами) и соответствует строке "LI". Если данный элемент коллекции имеет указанное имя, то увеличиваем счетчик строк «n++».

Затем, также в теле цикла, проверяем значение счетчика: если оно равно 2, то данный элемент цикла является нашим искомым — второй строкой списка. В таком случае сохраняем это значение в переменной «element2» и останавливаем цикл. В противном случае цикл перейдет к следующей итерации.

После окончания цикла проверяем, был ли вообще найден второй элемент. Если был, то вызываем метод «list.removeChild» для него.

Сохраните файл и обновите страницу браузера. Убедитесь в работоспособности новой кнопки, а также в отсутствии ошибок, когда в списке остается только одна строка.

**Задание для самостоятельной работы:** дополните созданную программу кнопкой, по нажатию на которой новый элемент будет добавляться в центр списка (или перед центральным элементом, если их количество нечетное).

Рассмотрим еще один пример, иллюстрирующий манипуляции с деревом DOM. Поставим себе задачу: разработать список, в котором можно менять порядок элементов при помощи технологии Drag-and-Drop, то есть перетягивать элементы списка мышкой и вставлять их в нужное место списка.

Технологию Drag-and-Drop мы рассматривали выше. Для нашей задачи ее надо немного видоизменить. Когда пользователь нажимает на некоторый пункт списка, он не должен исчезнуть из списка при перетягивании. Иначе

возникнет сбой нумерации списка и скачок его размера. Также может быть не совсем понятно, в какое место списка элемент вставится, если его отпустить.

Сделаем так:

- При нажатии кнопки мыши и начале перетягивания элемент в списке получит некоторое выделение для того, чтобы понятно было какой элемент является активным. Из списка он не исчезнет, чем сохранит нумерацию и размеры всего списка.
- Вместе с курсором мыши будет двигаться копия активного элемента, создавая видимость процесса перетягивания.
- При смене позиции курсора мыши выделенный (активный) элемент списка будет менять свое место в списке, следуя за курсором.

Приведем полный код документа и далее проведем анализ его работы. Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое (код также доступен в папке *Sources* — файл *js4\_15.html*):

```
<!doctype html>
<html>
<head>
  <style>
    body{
      position: relative;
    }
    .item, .phantom {
      height: 30px;
      width: 300px;
    }
  </style>
</head>
<body>
```

```

        .phantom{
            position: absolute;
        }
    </style>
</head>

<body>
    <ol id='list'>
        <li class='item' style='background-color:gold'>
        </li>
        <li class='item' style='background-color:red'>
        </li>
        <li class='item' style='background-color:green'>
        </li>
        <li class='item' style='background-color:blue'>
        </li>
    </ol>

    <script>
        var draggedElement = false;
        var phantomElement = false;
        document.onmousedown = function(e){
            e.preventDefault();
            var clickedElement = document.
            elementFromPoint(e.clientX, e.clientY);
            if(clickedElement.className.
                indexOf('item')>-1){
                clickedElement.style.opacity = '0.5';
                draggedElement = clickedElement;
            }
        }

        document.onmousemove = function(e){
            if(draggedElement){
                if(!phantomElement){
                    phantomElement =
                        document.createElement('div');

```

```

phantomElement.style.backgroundColor =
    draggedElement.style.
        backgroundColor;
phantomElement.style.left =
    e.pageX-draggedElement.
        offsetWidth/2 + 'px';
phantomElement.style.top =
    e.pageY-draggedElement.
        offsetHeight/2 + 'px';
phantomElement.className =
    "phantom";
document.body.appendChild(
    phantomElement);
}
else
{
    phantomElement.style.left =
        e.pageX-phantomElement.
            offsetWidth/2 + 'px';
    phantomElement.style.top =
        e.pageY-phantomElement.
            offsetHeight/2 + 'px';
    phantomElement.style.zIndex = '-1';
    var lowerElement = document.
        elementFromPoint(e.clientX,
            e.clientY);
    phantomElement.style.zIndex = '1';

    if(lowerElement != null
        && lowerElement != draggedElement
        && lowerElement.className.
            indexOf('item')>-1){
        if(lowerElement ==
            window.list.lastChild &&
            e.pageY > (lowerElement.offsetTop+
                lowerElement.offsetHeight/2)){
            window.list.
                removeChild(draggedElement);

```

```

        window.list.
            appendChild(draggedElement);
    }
    else
    {
        if(e.pageY >
            (lowerElement.offsetTop +
            lowerElement.offsetHeight/2)){
            if(lowerElement.
                previousSibling ==
                draggedElement){
                window.list.
                    removeChild(
                        draggedElement);
                window.list.insertBefore(
                    draggedElement,
                    lowerElement.
                    nextSibling);
            }
        }
        else
        {
            window.list.removeChild(
                draggedElement);
            window.list.insertBefore(
                draggedElement,
                lowerElement);
        }
    }
}

}

}

}

document.onmouseup = function(e){
    if(draggedElement){
        draggedElement.style.opacity = '1';
    }
}

```



```
        draggedElement = false;
    }

    if (phantomElement) {
        document.body.removeChild(phantomElement);
        phantomElement = false;
    }
}

</script>

</body>
</html>
```

Основу HTML части составляет список «`<ol id='list'>`» в котором создано четыре элемента. Эти элементы выделены разными цветами для большей наглядности. Список является нумерованным, что позволит нам следить за тем, что количество и нумерация элементов при перетягивании не меняется. Внешний вид списка на вкладке браузера представлен на следующем рисунке.



Рисунок 20

Далее следует скриптовая часть. Для того чтобы обеспечить функциональность технологии Drag-and-Drop мы вводим глобальные переменные:

```
var draggedElement = false;  
var phantomElement = false;
```

Первая «**draggedElement**» будет отвечать за реальный элемент списка, который будет перемещаться — за активный элемент. Вторая «**phantomElement**» будет ссылаться на дополнительный элемент — копию активного элемента списка, который будет следовать за указателем мыши.

С целью отделения кода JavaScript и разметки HTML обработчик события нажатия кнопки мыши реализован у объекта «**document**». Ранее мы приводили аргументы в пользу того, что данный обработчик логично реализовывать у перетягиваемого объекта. Однако в нашей новой задаче перетягиваемых объектов несколько. Более того, их количество может меняться, если в список будут добавляться или удаляться элементы. Вместо того чтобы перебирать все эти элементы и устанавливать для них отдельные обработчики создадим универсальный метод, подходящий для всех элементов. В таком случае он должен принадлежать одному из родительских элементов, общим для которых является «**document**»:

```
document.onmousedown = function(e)
```

Следует отметить, что браузеры сами по себе могут поддерживать технологию Drag-and-Drop, позволяя перетянуть элементы из браузера в проводник или рабочий стол, обеспечивая таким образом быстрое сохранение картинок или других объектов веб-страницы. Для того

чтобы наша задача не конфликтовала со встроенными механизмами браузера, первым делом отменим стандартную обработку события нажатия кнопки мыши командой «`e.preventDefault()`».

Далее нам необходимо определить элемент списка, находящийся в точке под курсором мыши. Поскольку событие получает не сам элемент, а документ, это необходимо сделать в два этапа:

1. Определяем элемент под курсором мыши и сохраняем его в переменной «`clickedElement`». Используем стандартный метод «`elementFromPoint`» объекта «`document`»: `var clickedElement = document.elementFromPoint(e.clientX,e.clientY);`
2. Проверяем, является ли данный объект элементом списка, т.к. документ будет получать события и от совершенно других элементов. Делаем это путем проверки наличия класса «`item`» у определенного объекта: `if(clickedElement.className.indexOf('item') > -1);`
3. Обратите внимание, что все элементы списка подключают этот стилевой класс при HTML объявлении.

В случае, если проверка проходит успешно, выделяем данный элемент, изменяя в два раза его прозрачность:

```
clickedElement.style.opacity = '0.5';
```

А также сохраняем данный элемент в глобальной переменной для обеспечения возможности доступа к нему из других функций, в частности, из обработчика событий движения мыши:

```
draggedElement = clickedElement;
```

Далее рассмотрим основную функцию — обработчик движения мыши. Поскольку все действия должны выполняться только в том случае, если происходит процесс перетягивания, все тело функции заключено в соответствующее условие:

```
document.onmousemove = function(e) {  
    if (draggedElement) {
```

Затем проверяем наличие элемента-копии (фантома), который следует за курсором мыши. При нажатии кнопки мыши мы его не создавали, т.к. щелчок мышью еще не означает перетягивание. Анализируем это именно при движении мыши.

Если фантома нет, то создаем его как отдельный блок «div»:

```
phantomElement = document.createElement('div');
```

Мы не должны добавлять фантом как элемент списка, иначе количество элементов списка увеличится. Фантом не будет принадлежать списку, играя роль лишь визуализации перетягивания.

После создания нового элемента устанавливаем для него такой же цвет, как у активного элемента в списке, чтобы дополнительно информировать пользователя какой элемент перемещается:

```
phantomElement.style.backgroundColor =  
    draggedElement.style.backgroundColor
```

Далее устанавливаем координаты блока учитывая координаты курсора мыши. По аналогии с предыдущими

примерами по технологии «Drag-and-Drop» вы наверняка заметили, что блок смещается таким образом, чтобы курсор мыши находился в его центре (смещается на половину ширины и высоты). Остальные стилевые атрибуты для блока-фантома задаются при помощи подключения стилевого класса «**phantom**». После чего блок добавляется к дочерним элементам тела документа:

```
document.body.appendChild(phantomElement)
```

Еще раз повторимся, блок не должен принадлежать списку, чтобы не влиять на его структуру. Фантом принадлежит телу документа.

Вторая часть условного оператора (**else**) отвечает за ситуацию, когда фантомный элемент уже был создан ранее в предыдущих вызовах события. Первым делом для фантома устанавливаются новые координаты, согласно координатам курсора мыши, переданным в аргументе события. Затем определяется элемент списка, находящийся под курсором мыши. Здесь тоже есть своя особенность: для того чтобы фантомный элемент был «поверх» остальных элементов списка ему установлен стиливой атрибут «**z-index**». Однако, в таком случае именно он всегда будет тем элементом, который находится под курсором мыши. Поэтому элемент списка мы определяем по следующему алгоритму:

1. Прячем фантомный элемент ниже списка, устанавливая отрицательное значение стилевого атрибута «**z-index**»:

```
phantomElement.style.zIndex = '-1'
```

2. Определяем элемент, находящийся под курсором мыши уже известной нам командой «`elementFromPoint`». Результат сохраняем в переменной «`lowerElement`»:

```
var lowerElement =  
document.elementFromPoint(e.clientX, e.clientY);
```

3. Возвращаем фантомному элементу исходное значение атрибута «`z-index`» чтобы он и далее отображался поверх остальных элементов:

```
phantomElement.style.zIndex = '1';
```

В результате мы получим элемент, находящийся под курсором мыши и под фантомным элементом, т.к. в момент определения мы перемещали его в нижний слой разметки страницы.

Далее необходимо проверить, что элемент под курсором вообще существует. Если движение мыши происходит по пустой части страницы, то такого элемента не будет и в переменной «`lowerElement`» сохранится значение «`null`». Также следует убедиться, что данный элемент принадлежит списку. Это мы уже разбирали — достаточно проверить наличие у элемента стилевого класса «`item`». Дополнительно нужно проверить то, что курсор мыши перешел к следующему элементу списка, а не находится еще в пределах активного элемента.

В итоге, комплексная проверка на то что элемент под курсором «`lowerElement`» существует, этот элемент относится к списку и не является активным для перетягивания будет реализована условием:

```
if(lowerElement != null
  && lowerElement != draggedElement
  && lowerElement.className.indexOf('item')>-1){
```

Если все эти условия выполнены, то курсор мыши перешел к другому (неактивному) элементу списка, а значит необходимо поменять местами в списке элементы, хранящиеся в переменных «`lowerElement`» и «`draggedElement`».

Если реализовать перестановку элементов непосредственно, то возможно появление эффекта «дребезга» — попеременная перестановка элементов списка в случаях, когда курсор мыши находится на границе между соседними элементами. Незначительные движения мыши на такой границе будут приводить к тому, что курсор находится то над одним элементом, то над другим. Возникнет неприятное мерцание со сменой элементов друг друга.

Для того чтобы нивелировать эффектдребезга добавим дополнительное условие — курсор мыши должен не просто перейти к соседнему элементу (превысив значение «`lowerElement.offsetTop`»), а пройти дальше, чем находится его центр по высоте («`lowerElement.offsetHeight/2`»). Это дополнительное условие будет выражено как

```
e.pageY > (lowerElement.offsetTop +
            lowerElement.offsetHeight/2)
```

Координата **X** курсора мыши на перестановку списка не влияет, т.к. список расположен вертикально. Соответственно, ее значение в условиях не учитывается.

Перестановка местами элементов «`lowerElement`» и «`draggedElement`» также будет отличаться для различных их взаимных расположений. Если «`lowerElement`» является последним в списке, то для перестановки нужно элемент «`draggedElement`» добавить в конец списка. Это обеспечивается методом «`appendChild`».

Если мышшь движется вверх и активный элемент перемещается ближе к началу списка, то «`draggedElement`» должен быть вставлен в список непосредственно перед «`lowerElement`». Такую перестановку обеспечит вызов метода «`insertBefore(draggedElement, lowerElement)`».

Если движение происходит в обратном направлении, то «`draggedElement`» необходимо вставлять в список после «`lowerElement`». Однако такого метода как «вставить после» в модели DOM не предусмотрено. Воспользовавшись знаниями об отношениях узлов дерева DOM отметим, что «вставить после узла» это значит «вставить перед следующим соседом узла». То есть вызов метода примет вид:

```
insertBefore(draggedElement, lowerElement.nextSibling)
```

Если же у элемента нет следующего соседа, то значит элемент является последним, а эту ситуацию мы разобрали выше отдельным пунктом обсуждения.

Отметим, что перед вставкой элемента «`draggedElement`» в список в новую позицию, его необходимо удалить в старой позиции командой «`removeChild(draggedElement)`». Это касается всех трех вариантов перестановки. Также отметим, что все команды управления деревом DOM должны вызываться у списка, т.к. речь идет о пере-



становках его дочерних элементов. То есть все описанные выше команды относятся к объекту «[window.list](#)».

В завершение, рассмотрим обработчик события отпущения кнопки мыши. Он состоит из двух условных операторов, определяющих были ли ранее созданы объекты «[draggedElement](#)» и «[phantomElement](#)». Напомним, что они создаются раздельно и, следовательно, требуют отдельных проверок.

Элемент «[draggedElement](#)» отвечает за реальный объект в списке, позиция которого меняется при движении мыши. Окончание перетягивания, наступающее после отпущения кнопки мыши, должно снять с этого элемента выделение. При нажатии кнопки мыши мы уменьшали прозрачность элемента. Значит вернем ее в исходное значение.

```
draggedElement.style.opacity = '1';
```

Затем установи значение «[false](#)» для переменной «[draggedElement](#)» для того чтобы обработчик события движения мыши не выполнял команды по перемещению элементов списка. Удалять объект «[draggedElement](#)» не нужно, т.к. при этом изменится сам список.

Элемент «[phantomElement](#)», наоборот, создан в документе дополнительно, а значит по завершению перетягивания должен быть удален из тела документа:

```
document.body.removeChild(phantomElement);
```

Полностью аналогично, самой переменной «[phantomElement](#)» также устанавливается значение «[false](#)».

Разобравшись в алгоритме работы программной части, сохраните файл и откройте его в браузере. Внешний

вид списка должен соответствовать приведенному ранее рисунку.

Наведите курсор мыши на любой элемент списка и нажмите кнопку мыши. Цвет элемента должен поблекнуть. Не отпуская кнопку начните движение мыши. Возле ее курсора появится блок с теми же размерами и цветом, как у активного элемента списка. Проведите мышью вверх и вниз списка, обратите внимание как происходит смена позиции активного элемента.



*Рисунок 21*

Отпустите кнопку мыши, убедитесь в том, что фантомный элемент исчезает, активный элемент восстанавливает насыщенность цвета и новый порядок списка остается неизменным. Повторите описанные действия с другими элементами списка, устанавливая их в произвольном порядке.

**Задание для самостоятельной работы:** реализуйте в программе с перестановкой списка улучшение, позволяющее «тянуть» выбранный элемент за ту же точку, на которой произошел щелчок мыши, а не только за центр блока (воспользуйтесь тем же методом, что и при решении задачи из примера «Drag-and-Drop»)

# Поиск элементов

Остановим внимание на достаточно популярной и, в то же время, мало рассмотренной ранее задаче поиска элементов. Как следует из названия, сутью задачи является поиск в структуре DOM и объединение найденных элементов, отвечающих заданным требованиям, в программные переменные или массивы.

Поиск позволяет выделить из дерева DOM некоторую группу элементов, которые отвечают определенным условиям отбора. Получив сведения о таких группах, можно управлять их свойствами или объединять их значения. Например, обвести красной рамкой все пустые текстовые поля, которые забыл заполнить пользователь, или выделить все те поля, которые необходимы для заполнения. Обработывая информацию о найденной группе элементов, мы можем установить для них одинаковые методы или обработчики событий, например, установить обработчик события «`mousedown`» для всех элементов, реализующих класс «`draggable`». Задачи управления группами элементов имеют очень широкий перечень возможностей и реализуются практически во всех интерактивных сайтах.

Существует несколько методов поиска элементов в зависимости от характера и критериев поиска. Один из таких методов нам уже известен: «`document.getElementById("val")`». Он позволяет найти элемент по его идентификатору, то есть по значению HTML-атрибута «`id="val"`». Если элемента с указанным идентификатором не существует, метод возвратит значение «`null`». Мы неод-

нократно использовали ранее этот метод, поэтому детально на нем останавливаться не будем. Отметим только, что для данного метода есть альтернативы — использование для доступа к элементу с идентификатором «`val`» одноименного поля глобального объекта «`window.val`» или даже без указания «`window`» — просто «`val`». В предыдущих примерах мы пользовались этими альтернативами. Несмотря на простоту альтернативных методов, стандартом рекомендуется использовать для поиска элемента именно метод «`getElementById`».

Похожим образом работают такие методы поиска элементов как: «`getElementsByName`», «`getElementsByClassName`» и «`getElementsByTagName`»:

- Метод «`getElementsByName("theName")`» производит поиск элементов по их имени, заданных HTML-атрибутом «`name="theName"`»;
- Метод «`getElementsByClassName("theClass")`» ищет элементы по имени стилевого класса, указанного в атрибуте «`class="theClass"`»;
- Метод «`getElementsByTagName("DIV")`» собирает элементы по имени HTML тега, которым элементы были созданы (`<DIV></DIV>`).

В отличие от метода «`getElementById`», который находит один элемент, приведенные выше методы рассчитаны на поиск групп элементов. Это отражается в названии методов, обратите внимание, что слово «`Element`» включается в имена методов по-разному: в единичной форме «`Element`» либо в множественной «`Elements`». Очевидно, что в документе может быть несколько элементов, соз-

данных тегом «**DIV**», разные элементы могут реализовывать один и тот же стилевой класс. Одинаковое имя (атрибут «**name**») используется, например, для создания групп зависимых радиокнопок (детальнее это будет рассмотрено в следующем уроке). Результатом работы любого из методов группового поиска будет коллекция типа «**HTMLCollection**».

Еще одно отличие в методах поиска заключается в том, что метод «**getElementById**» может быть вызван только у объекта «**document**», а остальные методы могут вызываться в любом узле дерева элементов DOM. В таком случае поиск будет осуществляться только в той части дерева, корнем которого является данный узел.

Также, в отличие от метода «**getElementById**», который возвращает значение «**null**» в случае неудачного поиска, методы группового поиска возвращают пустые коллекции. Соответственно, анализ результатов поиска для них будет отличаться.

Наиболее широкие возможности поиска элементов обеспечивает метод «**querySelectorAll**». В качестве аргумента он принимает CSS селектор в такой же форме, как и в стилевом определении (или в стилевом файле), например:

Селектор	описание
*	все элементы
<b>P</b>	элементы с тегом <p>, аналог <code>getElementsByTagName("p")</code>
<b>#d1</b>	элемент с <code>id= "d1"</code> , аналог <code>getElementById("d1")</code>
<b>.c1</b>	элементы с классом <code>class="c1"</code> , аналог <code>getElementsByClassName("c1")</code>

Селектор	описание
<code>[name="n1"]</code>	элементы с атрибутом "name" равным "n1", в данном случае аналог <code>getElementsByName("n1")</code>
<code>[type= "text"]</code>	элементы с атрибутом "type" равным "text"
<code>[selected]</code>	элементы с указанным атрибутом "selected"
<code>p.c1</code>	элементы с тегом <code>&lt;p&gt;</code> и классом "c1": <code>&lt;p class="c1"&gt;</code>
<code>p, div</code>	элементы с тегом <code>&lt;p&gt;</code> и элементы с тегом <code>&lt;div&gt;</code> (объединение)
<code>p div</code>	элементы с тегом <code>&lt;div&gt;</code> дочерние для элементов с тегом <code>&lt;p&gt;</code> ( <code>&lt;p&gt;&lt;div&gt;element&lt;/div&gt;&lt;/p&gt;</code> )

Более полное описание CSS селекторов можно посмотреть в стандартах, например, на [странице](#).

Метод `querySelectorAll` также может быть вызван в любом узле дерева DOM, в таком случае он учитывает только поддерево данного узла. Например, если использовать глобальный поиск во всем документе при помощи инструкции:

```
document.querySelectorAll("LI")
```

то результатом поиска будет коллекция из всех элементов, созданных тегом `<li>`, найденных в документе и, возможно, принадлежащих разным спискам. Если в документе существует список `<ul id="list1">...</ul>`, то поиск может быть вызван только для него:

```
list1.querySelectorAll("LI")
```

В таком случае элементы `<li>` будут отбираться только из данного списка. Точнее, из данного списка и всех вложенных в него списков, если такая вложенность присутствует.

В качестве результата метод возвращает коллекцию элементов «`NodeList`» (такой же тип, как у коллекции «`childNodes`», отличается от результатов «`getElements...`»-методов). Аналогично методам группового поиска, результат всегда будет являться коллекцией: если элементов по заданному селектору не найдется, то результатом поиска будет пустая коллекция. Если элемент будет найден один, все равно будет возвращена коллекция с одним объектом в своем составе.

Если нам известно, что элемент с заданным селектором в документе существует только один, то можно воспользоваться упрощенной версией метода — «`querySelector`». Данный метод возвращает один результат — первый из найденных, соответствующий указанному селектору. Его действие аналогично вызову «`querySelectorAll("selector")[0]`», только происходит значительно быстрее, так как останавливается после первого нахождения.

Для иллюстрации способов применения различных методов поиска рассмотрим следующий пример. Расположим на странице несколько элементов, создав их различными тегами, зададим им имена и стилевые классы. Для того чтобы иметь возможность выбирать различные группы из этих элементов, используем для некоторых из них одинаковые теги, для других групп — одинаковые имена или классы (в различных комбинациях). Для запуска разных методов поиска предусмотрим несколько кнопок, наглядность их работы будет заключаться в изменении стилей выбранной группы элементов.

Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое (код также доступен в папке *Sources* — файл *js4\_16.html*):

```
<!doctype html>
<html>
<head>
  <style>
    div, p {
      border: 1px solid navy;
      display: inline-block;
      height: 60px;
      margin: 10px;
      padding-top: 40px;
      text-align: center;
      width: 100px;
    }

    .c1 {
      background: #eee;
    }

    button {
      margin-left: 25px;
      margin-top: 30px;
      width: 90px;
    }
  </style>
</head>

<body>
  <div id="d1" class="c1" name="n1">d1, c1, n1</div>
  <div id="d2" class="c2" name="n1">d2, c2, n1</div>
  <div id="d3" class="c1 c2">d3, c1 c2</div>
  <p id="p1" class="c1">p1, c1</p>
  <p id="p2" class="c2" name="n1">p2, c2, n1</p>
```



```
<br>
<button onclick="selD1()">id = d1</button>
<button onclick="selC1()">class = c1</button>
<button onclick="selP()">tag = P</button>
<button onclick="selN1()">name = n1</button>
<button onclick="clr()">Clear</button>

<script>
    function selD1() {
        var elem = document.getElementById("d1");
        elem.style.backgroundColor = "#cfc";
    }

    function selC1() {
        var elems =
            document.getElementsByClassName("c1");
        for(elem of elems)
            elem.style.backgroundColor = "#fcc";
    }

    function selP() {
        var elems =
            document.getElementsByTagName("p");
        for(elem of elems)
            elem.style.backgroundColor = "#ccf";
    }

    function selN1() {
        var elems =
            document.getElementsByName("n1");
        for(elem of elems)
            elem.style.backgroundColor = "#bef";
    }

    function clr() {
        var elems =
            document.querySelectorAll("div,p");
```

```

        for(elem of elems)
            elem.style.backgroundColor = "#eee";
    }
</script>
</body>
</html>

```

Сохраните файл и откройте его в браузере. Результат должен быть подобным приведенному на рисунке 22.

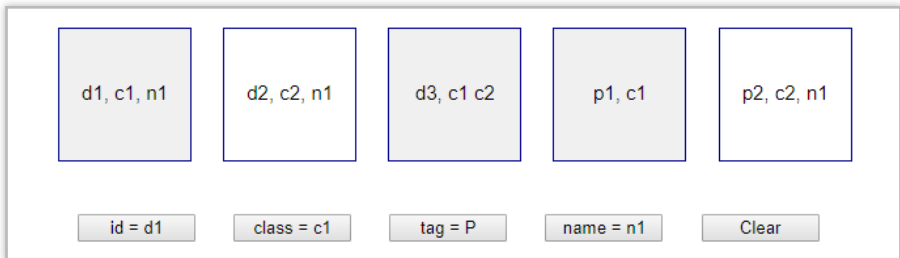


Рисунок 22

Основу страницы составляют пять блоков — три созданы тегом `<div>`, два — тегом `<p>`. Для них применены одинаковые размеры и рамки, поэтому они отображаются без явных отличий.

Для разных блоков указаны различные комбинации идентификаторов (`id`), классов (`class`) и имен (`name`). Для наглядности значения этих атрибутов продублированы в телах блоков и отображаются на странице внутри рамок блоков. Идентификаторы блоков (`<div>`) начинаются с буквы «**d**», абзацев (`<p>`) — с буквы «**p**». Это дает возможность отличить их между собой на странице.

Для элементов, реализующих стилевой класс «`c1`», изначально указан серый цвет фона. Это позволяет убе-

даться в правильности подключения стилей и предупредить ошибки или опечатки.

Вторую часть страницы представляют собой кнопки, реализующие различные методы поиска элементов. Функции, отвечающие за нажатия кнопок, построены по схожему принципу:

1. Выполняется команда поиска, например, `var elems = document.getElementsByName("n1");`
2. Организовывается цикл по полученной коллекции: `for(elem of elems);`
3. В теле цикла для всех членов коллекции устанавливается фоновый цвет `elem.style.backgroundColor = "#bef"`.

Различные функции отличаются методами поиска и цветом, который устанавливается для фона. Принципиальные отличия содержит функция нажатия первой кнопки «`selD1`», демонстрирующая использования метода «`getElementById`». Поскольку этот метод возвращает не коллекцию, а один элемент, цикл в этой функции не используется.

Итак, первая кнопка должна найти элемент с идентификатором «`d1`». Нажмите ее и убедитесь, что первый блок, имеющий этот идентификатор, поменял фоновый цвет, тогда как остальные остались без изменений.

Вторая кнопка отвечает за поиск элементов по имени класса. Ее основным методом поиска является инструкция «`getElementsByClassName("c1")`». Нажмите эту кнопку и убедитесь, что изменения коснулись тех элементов, которые реализуют этот класс. В том числе и среднего блока, который реализует два класса «`c1`» и «`c2`».

Третья кнопка демонстрирует поиск по имени тега «<p>» при помощи метода «`getElementsByTagName("p")`». Нажмите ее, — фоновый цвет должны поменять только последние два блока, созданные именно этим тегом.

Четвертая кнопка реализует поиск по имени (по HTML-атрибуту «name»). Ее поисковая инструкция имеет вид «`getElementsByName("n1")`». Убедитесь, что ее нажатие затронет только те элементы, в которых указано имя «n1».

Последняя кнопка выполняет задачи очистки — установки для всех блоков серого фонового цвета. Для выбора всех элементов документа, представляющих собой блоки, применен селектор, объединяющий выбор всех элементов с тегом <div> и всех элементов с тегом <p>: «`querySelectorAll("div,p")`». Нажмите эту кнопку — цветное выделение всех блоков должно исчезнуть.

**Задание для самостоятельной работы.** Дополните приведенную страницу еще несколькими блоками с различными комбинациями тегов, имен, классов, атрибутов и идентификаторов. Примените для выбора различных групп блоков расширенные возможности метода «`querySelectorAll`». Создайте кнопки, демонстрирующие выбор следующих групп:

- блоки с тегом <p> и классом «c2»;
- блоки с тегом <p> и классом «c2» + блоки с тегом <div> и классом «c1»;
- блоки, имеющие имя (произвольное значение атрибута «name»).

## Управление ссылками: объекты Link и Links

Отдельную группу HTML элементов представляют собой ссылки. В отличие от большинства других элементов, которые отвечают только за разнообразное отображение, ссылки содержат указатели на другие ресурсы: файлы, страницы или потоки. Использование ссылок так или иначе связано с обращениями к этим ресурсам, а значит для них необходимы специальные возможности браузера, напрямую не нужные для других элементов страницы.

Ссылки можно разделить на две группы, работа с которыми имеет свои особенности. Первую группу ссылок представляют так называемые ссылки-якоря (*англ. anchors*), реализуемые тегом `<a>`. Обычно, эти ссылки используются для перехода к новой странице, загрузки файла или прокрутки данной страницы к определенному месту. Для создания анонимных невидимых ссылок, располагающихся, как правило, на различных частях рисунков, применяется тег `<area>`. Возможности этих ссылок детально рассматривались в первой части курса при изучении языка HTML.

Вторая группа ссылок включает в себя указатели на дополнительные подключаемые ресурсы. Как правило эти ссылки указываются в заголовочной части документа при помощи тега `<link>`. Наиболее популярными областями использования таких ссылок являются подключение стилевых файлов (`<link rel="stylesheet" ...>`) и картинок

и кнопок для страницы (`<link rel="icon" ...>`). Однако у ссылок этого типа есть и ряд других возможностей.

Обе приведенные группы обобщаются термином «ссылки», поэтому при подробном рассмотрении следует уточнять о ссылках какого типа идет разговор, особенно, в связи с некоторой путаницей в терминологии.

Все ссылки-якоря документа автоматически собираются в коллекцию «`document.links`». Это первая причина возможных ошибок в формальном понимании: элементы, созданные тегом `<a>`, попадают в коллекцию «`links`», тогда как элементы, созданные тегом `<link>`, в ней отсутствуют. В коллекцию «`document.links`» также попадают элементы, созданные тегом `<area>`.

Для элементов `<link>` не создается специальная отдельная коллекция. Для того чтобы получить доступ к ссылкам типа `<link>`, необходимо выполнить поиск элементов при помощи рассмотренной в предыдущем разделе функции «`getElementsByTagName("LINK")`».

В коллекции «`document.links`» дополнительную информацию про ссылки (якоря) можно получить запрашивая атрибуты «`href`», отвечающий за адрес ссылки, и «`innerText`», представляющий текстовую надпись для ссылки. Отдельные компоненты полного адреса (`href`) также доступны в свойствах «`protocol`», «`host`» и «`hash`». Детальное описание составных частей адресов было приведено в первой части урока.

Элементы `<link>` также содержат атрибут «`href`», указывающий на подключаемый ресурс. Однако для них, в отличие от ссылок-якорей, необходим атрибут «`rel`», отвечающий за тип подключаемого ресурса. Если не ука-

затем данный атрибут, то ссылка может быть обработана неправильно.

Рассмотрим особенности работы с ссылками различного типа на следующем примере. Поставим себе две задачи:

1. Реализовать смену стилей страницы при помощи воздействия на ссылку `<link>`, подключающую стилевой файл;
2. Собрать сведения обо всех ссылках, созданных тегом `<a>`, находящихся в документе.

Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое (код также доступен в папке *Sources* — файл *js4\_17.html*):

```
<!doctype html>
<html>

<head>
  <link rel="stylesheet" href="style.css" />
</head>

<body>
  <ul>
    <li>
      <a href="js4_12.html">Drag-and-Drop basics</a>
    </li>
    <li>
      <a href="js4_13.html">Adding DOM nodes</a>
    </li>
    <li>
      <a href="js4_14.html">Adding and removing
        DOM nodes</a>
    </li>
```

```

    <li>
        <a href="js4_15.html">Drag-and-Drop
        list ordering</a>
    </li>
</ul>
<input type="button"
    value="Get links"
    onclick="getLinks()" />
<br/>
Apply new style:<br/>
<input type="radio"
    name="styler"
    onclick="changeStyle(1)" /> style1
<br/>
<input type="radio"
    name="styler"
    onclick="changeStyle(2)" /> style2
<br/>

<script>
    function getLinks() {
        var str = "";
        var links = document.links;
        for(link of links)
            str += link.innerText + "(" +
                link.href + ")\n" ;
        alert(str);
    }

    function changeStyle(num) {
        var styleLink = document.
            getElementsByTagName("LINK")[0];
        document.head.removeChild(styleLink);
        styleLink = document.createElement("LINK");
        styleLink.href = "style"+num+".css";
        styleLink.rel = "stylesheet";
        document.head.appendChild(styleLink);
    }

```



```

    }
  </script>
</body>
</html>

```

Дополнительно к данному файлу создайте три стилевых файла: «[style.css](#)», «[style1.css](#)» и «[style2.css](#)» со следующим содержимым (файлы также доступны в папке *Sources*):

```

style.css
ul {
  list-style-type: disc;
}

style1.css
ul {
  list-style-type: circle;
}

style2.css
ul {
  list-style-type: square;
}

```

В заголовочной части документа при помощи ссылки `<link rel="stylesheet" href="style.css" />` подключается первый стилевой файл. Далее в теле документа оформляется список с несколькими ссылками на html-файлы предыдущих упражнений данного урока. Ниже списка размещается кнопка, обрабатывающая ссылки типа `<a>`, за ней следуют две радиокнопки, меняющие стили путем манипуляций со ссылкой на файлы стилевых ресурсов (`<link>`).

В скриптовой части документа описываются две функции — для работы с кнопкой и радиокнопками. Рассмотрим алгоритмы их работы.

Функция «`getLinks()`» собирает информацию о ссылках-якорях. В начале функции создается пустая строка. Далее организовывается циклический обход коллекции «`document.links`», на каждой итерации которого к строке добавляется информация о тексте ссылки (`link.innerText`) и об ее адресе (`link.href`). По завершению цикла выводится собранная из всех ссылок строка в виде диалогового окна «`alert`».

Функция «`changeStyle(num)`» призвана изменить стиль документа. Для этого в качестве аргумента в функцию передается номер стилевого файла: 1 или 2. Передача данных происходит в HTML-тегах радиокнопок в атрибуте «`onclick`».

Сначала функция удаляет из документа старый стиль. Выполняется поиск соответствующей ссылки «`styleLink = document.getElementsByTagName("LINK")[0]`». Нам известно, что такая ссылка в документе одна, поэтому сразу можем указать индекс «`[0]`» после результата поиска. Затем найденный элемент удаляется из дочерних элементов заголовочной части документа командой «`document.head.removeChild(styleLink)`».

Во второй части функции создается новый объект-ссылка «`styleLink = document.createElement("LINK")`». Для хранения используется та же переменная «`styleLink`», т.к. ее предыдущее значение более не требуется в программе. Новой ссылке указывается атрибут адреса «`styleLink.href = "style"+num+".css"`», формируясь с учетом передан-

ного номера «num». Также не забываем указать тип ссылки «`styleLink.rel = "stylesheet"`». По заполнению данных добавляем ссылку к дочерним элементам заголовочной части документа при помощи инструкции «`document.head.appendChild(styleLink)`».

Сохраните документ и откройте его при помощи браузера. Внешний вид страницы должен соответствовать следующему рисунку:

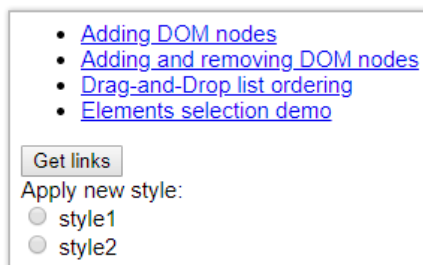


Рисунок 23

Обратите внимание на значки, используемые для маркировки списка, поскольку именно они меняются при помощи внешних стилевых файлов. Нажмите радиокнопку «`style1`». Должен подключиться стиливой файл «`style1.css`», определяющий маркеры списка в виде пустых кружков:

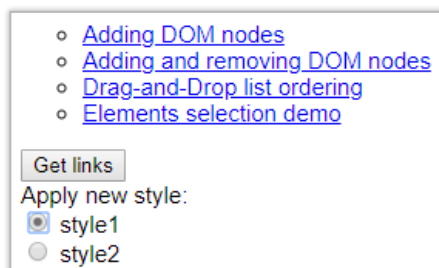


Рисунок 24

Аналогично убедитесь в работоспособности второй радиокнопки. Во втором стилевом файле маркеры определяются как квадраты.

Нажмите на кнопку «[Get links](#)». Как результат мы должны увидеть сообщение с полным перечнем ссылок-якорей в нашем документе. Убедитесь в правильности и полноте их обработки:

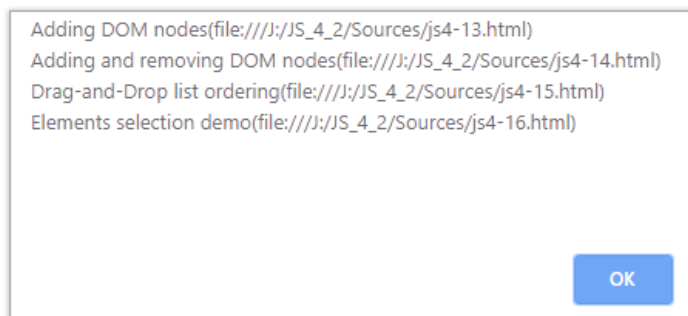


Рисунок 25

При помощи внешних стилевых файлов можно управлять так называемыми «темами отображения». Очевидно, что поменять вид маркеров списка можно гораздо проще, обратившись к стилевым определениям списка напрямую. Однако, если изменений в стилях будет больше, компактнее будет переключать стилевые файлы.

**Задание для самостоятельной работы.** Добавьте возможность подключения третьего стилевого файла. Дополните стилевые файлы определениями для цветов, размеров и оформления списка и ссылок в нем. Убедитесь, что стили переключаются, не влияя на работоспособность кнопки, анализирующей коллекцию ссылок-якорей.

# Управление выделением и текстовым диапазоном: объекты Range, Selection и TextRange

Задачи выделения текста или другого содержимого веб-страницы обычно связаны с их копированием и управляются средствами самого браузера. Дополнительных программ для обеспечения обыкновенного копирования не требуется.

Тем не менее, существуют программные средства, позволяющие вмешиваться в процессы выделения и копирования содержимого нашей веб-страницы. Например, добавить к скопированному тексту заметку об авторстве «Материал скопирован со страницы [mypage.org](#)». Или ограничить возможности копирования какого-либо материала.

Одной из наиболее популярных задач, требующих управления выделением, является уведомление о грамматических ошибках. На многих сайтах можно увидеть сообщение: «Если вы обнаружили на сайте ошибку, выделите ее и нажмите...» далее следует некоторая комбинация клавиш, которые следует нажать пользователю. Разберем особенности выделения текста на примере решения описанной задачи.

Сформируем текст для веб-страницы и нарочно заложим в него несколько ошибок. Применим следующую HTML разметку:

```
<p>If you find some mistakes on pagge<br>
  <i>selectt them and press ctrl-Enter</i></p>
```

Слова «pagge» (правильно «page») и «selectt» (правильно «select») написаны с ошибками. Пользователь должен выделить их (оба) и нажать комбинацию «Ctrl-Enter» на клавиатуре.

Выделение текста будет обеспечивать сам браузер, наша задача провести обработку выделенного текста после нажатия заданной комбинации клавиш. При анализе того, что будет представлять собой выделение текста, мы столкнемся со следующими особенностями:

- слово «page» принадлежит абзацу (тег `<p>`), тогда как слово «selectt» — его дочернему элементу `<i>`
- между словами находится HTML тег `<br>`

Как уже становится понятно, текстовое выделение может включать в себя и более сложные комбинации — объединять элементы разной степени иерархического отношения, включать в себя теги, в том числе и невидимые, рисунки, ссылки и тому подобное. При этом задачи к выделенной области тоже могут быть поставлены разные: использовать только текст, текст и рисунки, текст и разметку (таблицы, например) или полную HTML разметку.

Для создания универсальных средств выделения и работы с ним было реализовано следующий подход: в документе создаются два маркера — для начала и конца области выделения. Каждый из маркеров содержит в себе ссылку на элемент, в котором он находится,

и смещение (отступ маркера от начала элемента). Оба маркера объединяются в специальный объект «**Range**». Получить этот объект можно вызвав метод «**document.getSelection()**» или «**window.getSelection()**». Результаты их работы полностью идентичны между собой.

Создайте новый html-документ, наберите или скопируйте в него приведенную выше разметку с ошибками. Сохраните документ, откройте его в браузере. Выделите два слова, написанные неправильно, и откройте консоль разработчика. Напишите в ней запрос «**getSelection()**» (при обращении к объекту «**window**» его имя можно не указывать). Результат работы запроса будет иметь отличия в зависимости от того, каким браузером Вы пользуетесь. На следующих рисунках приведены два примера результатов, полученных в браузерах «Chrome» и «Firefox».

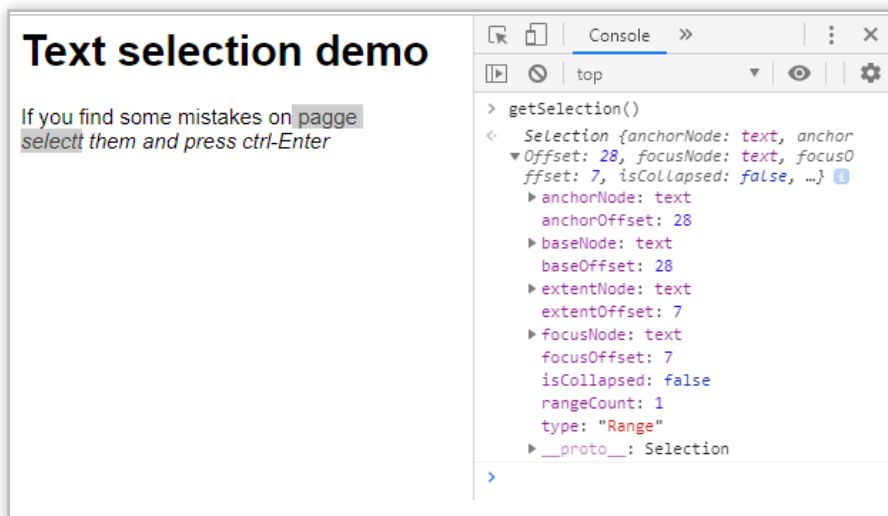


Рисунок 26

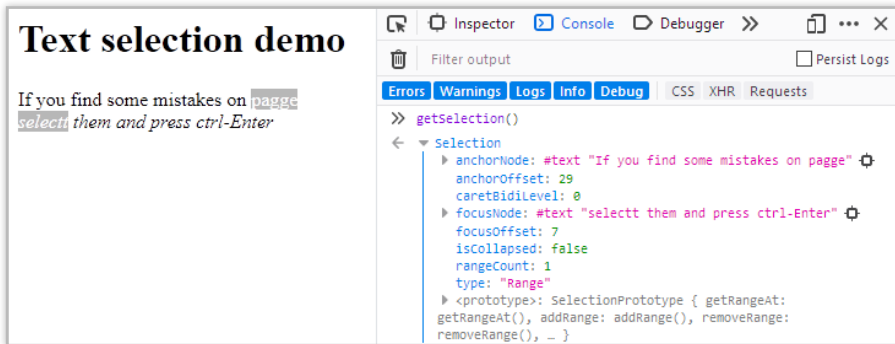


Рисунок 27

В браузере «Firefox» в ответе присутствуют два маркера — «`anchorNode`» и «`focusNode`». Первый маркер определяет элемент DOM, в котором находится начальная точка выделения текста. Второй — конечная. Для каждого элемента указано смещение «`anchorOffset`» (значение 29) и «`focusOffset`» (значение 7). Это означает, что маркер начала находится на 29-м символе элемента, маркер конца — на 7-м.

В браузере «Chrome» ответ содержит четыре маркера. Кроме двух, описанных выше, присутствуют также «`baseNode`» и «`extentNode`». Они представляют собой копии маркеров начала и конца выделения, существовавшие в браузере до вступления в силу стандарта, определяющего названия «`anchorNode`» и «`focusNode`».

Также обратите внимание, что при выделении текста в браузере «Chrome» был выделен дополнительный пробел перед словом «`pagge`». Как видно, это привело к уменьшению смещения «`anchorOffset`» до величины 28. Следует помнить, что в вычислении смещения маркера пробелы учитываются так же, как и другие символы.



Разные браузеры по-разному формируют объект-выделение, возвращаемый методом «`getSelection()`». Аналогично браузеру «Chrome», дополнительные объекты («`baseNode`» и «`extentNode`») создаются в браузерах «Opera», «Safari» и «Maxthon». Тогда как в браузерах «Firefox», «Tor», «Edge» и «Internet Explorer» объект-выделение содержит только два основных маркера («`anchorNode`» и «`focusNode`»). Для того чтобы коды были максимально универсальными, в своих скриптах рекомендуется использовать имена «`anchorNode`» и «`focusNode`», поддерживаемые всеми браузерами.

Завершим поставленную нами задачу определения выделения по нажатию комбинации «`ctrl-Enter`». Дополните созданный html-файл следующим содержимым (код также доступен в папке *Sources*, файл *js4\_18.html*):

```
<!doctype html>
<html>

<head>
  <title>Selection demo</title>
</head>

<body onkeypress="keyHandler(event)">
  <h1>Text selection demo</h1>
  <p>If you find some mistakes on pagge<br>
  <i>selectt them and press ctrl-Enter</i></p>
  <p id="out"></p>
  <script>
    function keyHandler(e) {
      if(e.ctrlKey && (e.key=="Enter" ||
                      e.code=="Enter")) {
        var sel = document.getSelection();
        var msg = "";
```

```

        msg += "Selection starts at " +
            sel.anchorOffset +
            "symbol of node <br>";
        msg += "<b>" + sel.anchorNode.data +
            "</b><br>";
        msg += "Selection ends at " +
            sel.focusOffset +
            "symbol of node <br>";
        msg += "<b>" + sel.focusNode.data +
            "</b><br>";
        msg += "selected string is: <b>" +
            sel.toString() + "</b>";
        out.innerHTML = msg;
    }
}
</script>
</body>
</html>

```

Для того чтобы иметь возможность реагировать на нажатия клавиш клавиатуры, при объявлении тела документа указан обработчик соответствующего события:

```
<body onkeypress="keyHandler(event)">.
```

В скриптовой части документа этот обработчик описан в виде одноименной функции. Тело обработчика заключено в условный оператор, проверяющий необходимую комбинацию: нажата клавиша «ctrl» (`e.ctrlKey` будет иметь значение `true`), а сама клавиша идентифицирована как «Enter». Поскольку разные браузеры хранят название клавиши в разных полях, проверка проводится как по свойству «`e.key`», так и по «`e.code`».

Затем запрашивается выделение. Для демонстрации альтернативного подхода, использован метод «`getSelection()`», принадлежащий объекту «`document`» (выше мы использовали метод объекта «`window`»). Данные о маркерах начала и конца выделения, а также их смещениях собираются в одну строку-сообщение «`msg`». Из неуказанного ранее отметим, что сам текст выделения можно получить при помощи вызова метода «`toString()`». В текст не будут входить HTML теги, находящиеся между маркерами начала и конца.

В завершении работы обработчика события сформированное сообщение помещается в абзац `<p id="out">` `</p>` при помощи установки его атрибута «`innerHTML`».

Сохраните изменения, откройте файл в браузере или обновите открытую ранее страницу. Также выделите два слова, содержащие ошибки, только в этот раз нажмите комбинацию клавиш «`Ctrl-Enter`» на клавиатуре. В нижней части страницы сформируется сообщение о параметрах выделения текста:

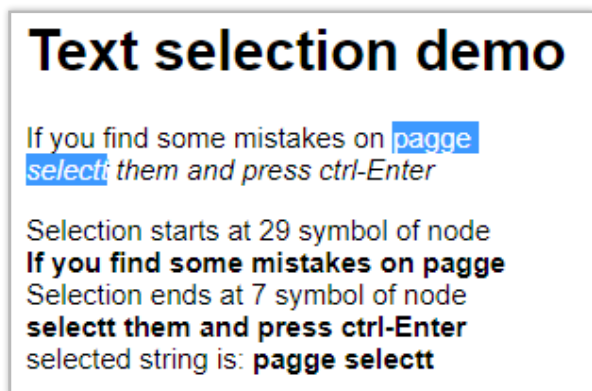


Рисунок 28

Область выделения можно только запросить для чтения, поскольку самим выделением управляет браузер, реагируя на действия пользователя. Тем не менее, область «[Range](#)», ограниченную двумя маркерами, можно создать самостоятельно. Таким образом можно имитировать процесс выделения текста. Рассмотрим эти возможности на следующем примере.

Создадим некоторый текст и расставим между его словами элементы «[checkbox](#)» (галочки). Будем считать, что область выделения будет задаваться двумя отмеченными элементами. После завершения выделения текст необходимо будет пометить, задав ему синий фоновый цвет.

Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое (*код также доступен в папке Sources — файл js4\_19.html*):

```
<!doctype html>
<html>
<head>
  <title>Selection demo</title>
</head>

<body>
  <input type="checkbox"/>
  Mark <input type="checkbox"/>
  selection <input type="checkbox"/>
  ranges <input type="checkbox"/>
  by <input type="checkbox"/>
  checking<input type="checkbox"/>
  boxes<input type="checkbox"/>
  between<input type="checkbox"/>
  the<input type="checkbox"/>
  words<input type="checkbox"/>
  <br>
```

```

<input type="button"
       value="Done"
       onclick="makeSelection() "/>
<p id="out"></p>
<script>
    function makeSelection() {
        var checkedBoxes =
            document.querySelectorAll(
                "[type='checkbox']:checked");
        if(checkedBoxes.length != 2) {
            out.innerHTML = "You should check
                            exactly 2 boxes:
                            start and finish of
                            selection";

            return;
        }
        var range = document.createRange();
        range.setStartAfter(checkedBoxes[0]);
        range.setEndBefore(checkedBoxes[1]);
        var span = document.createElement("SPAN");
        span.style.backgroundColor = 'aqua';
        range.surroundContents(span);
        out.innerHTML = "Selected text:<br>" +
                        range.toString();
    }
</script>
</body>
</html>

```

Основу тела документа представляет собой фраза «*Mark selection ranges by checking boxes between the words*» между каждым из слов которой вставлен элемент `<input type="checkbox"/>`. Такие же элементы добавлены в начале и конце фразы. За ними следует кнопка, обрабатывающая результат выделения.

В скриптовой части задается обработчик события нажатия кнопки «`makeSelection()`». Сначала выполняется поиск элементов при помощи метода «`querySelectorAll`»:

```
var checkedBoxes = document.querySelectorAll(
    "[type='checkbox']:checked");
```

CSS селектор «`[type='checkbox']:checked`» определяет DOM элементы с атрибутом «`type`», имеющим значение «`'checkbox'`» и реализующим псевдокласс «`:checked`», то есть отмеченные элементы, расставленные между словами.

Далее следует проверка размера коллекции, полученной как результат поиска. Если он не равен двум, формируем сообщение о необходимости двух маркеров и прекращаем работу функции. Дальнейший код будет выполняться, если условный оператор не выполнится. Использование оператора «`return`» не требует помещать его в блок «`else`».

Объект типа «`Range`» создается при помощи метода «`createRange()`», принадлежащего объекту «`document`». Вызовем этот метод и сохраним результат в переменной «`range`»:

```
var range = document.createRange();
```

После вызова метода создается пустой объект. Для его практического использования в нем необходимо создать маркеры начала и конца выделения. Для этих целей предусмотрено несколько методов, их краткое описание приводится в следующей таблице:

Метод объекта «Range»	Описание
<code>setStart(elem, offset)</code>	Устанавливает маркер начала в элементе «elem» со смещением «offset» от его начала
<code>setEnd(elem, offset)</code>	Устанавливает маркер конца в элементе «elem» со смещением «offset» от его начала
<code>setStartBefore(elem)</code>	Устанавливает маркер начала перед элементом «elem»
<code>setStartAfter(elem)</code>	Устанавливает маркер начала после элемента «elem»
<code>setEndBefore(elem)</code>	Устанавливает маркер конца перед элементом «elem»
<code>setEndAfter(elem)</code>	Устанавливает маркер конца после элемента «elem»
<code>selectAllChildren(elem)</code>	Устанавливает маркеры начала и конца вокруг элемента «elem»
<code>addRange(range)</code>	Устанавливает маркеры согласно диапазону «range»
<code>removeAllRanges()</code>	Очищает данные о маркерах

В рамках нашей задачи выделение должно начинать сразу после первого отмеченного элемента (`checkedBoxes[0]`) и заканчиваться непосредственно перед вторым отмеченным элементом (`checkedBoxes[1]`). Соответственно, для установки маркеров выделения логично использовать следующие команды:

```
range.setStartAfter(checkedBoxes[0]);
range.setEndBefore(checkedBoxes[1]);
```

Для реализации эффекта выделения используем стилевое оформление — цвет фона. Создадим новый элемент «`span`», не влияющий на сборку и размещение

html-элементов, и поместим в него наш выделенный диапазон. Добавление новых элементов мы рассматривали в предыдущих разделах, инструкции создания элемента «[span](#)» и установки его фонового цвета имеют вид:

```
var span = document.createElement("SPAN");  
span.style.backgroundColor = 'aqua';
```

Для того чтобы поместить выделенный диапазон в новый созданный элемент, можно указать его в качестве дочернего элемента у «[span](#)». Однако, для этой задачи существует специальный метод объекта «[Range](#)» — «[surroundContents](#)». Его применение более безопасно с точки зрения обработки неправильно сформированных диапазонов и предотвращает клонирование элементов, если после помещения в дочернюю коллекцию забыть удалить элемент их предыдущего места. Инструкция, использующая этот метод, имеет вид:

```
range.surroundContents(span);
```

После этого содержимое диапазона будет помещено в элемент «[span](#)» и, как следствие, приобретет голубой фоновый цвет.

В завершение работы функции-обработчика выводим текст выделения в отдельный абзац, используя метод «[range.toString\(\)](#)».

Сохраните документ и откройте его в браузере. Проверьте его работоспособность установив две метки в произвольных местах строки и нажав на кнопку «[Done](#)».



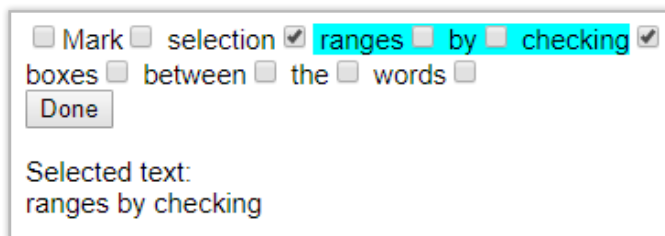


Рисунок 29

**Задание для самостоятельной работы:** Добавьте в созданную программу кнопку, отменяющую выделение (снимающую цветное выделение текста).

Приведем еще один пример обработки результатов выделения текста, иллюстрирующий как можно добавить заметку об авторских правах к копируемому тексту. Создайте новый HTML файл, в котором наберите или скопируйте следующее содержимое (код также доступен в папке *Sources* — файл *js4\_20.html*):

```
<!doctype html>
<html>
<head>
  <title>Selection demo</title>
  <style>
    #copyContainer {
      position:absolute;
      left:-1000px
    }
    textarea {
      height:100px;
      width: 300px;
    }
  </style>
</head>
```

```

<body>
  Select any part of this page and copy it.<br>
  Paste it in any other program or try to paste here:
  <br>
  <textarea></textarea>
  <div id="copyContainer"></div>

  <script>
    document.addEventListener('copy',
                              copyHandler);
    var storedRange = document.createRange();
    function copyHandler() {
      var sel = document.getSelection();
      storedRange.setStart(sel.anchorNode, sel.
                          anchorOffset);
      storedRange.setEnd(sel.focusNode,
                        sel.focusOffset);
      var txt = sel.toString() +
                "(Copied from js4_20.html)";
      copyContainer.innerHTML = txt;
      sel.selectAllChildren(copyContainer);
      setTimeout(function() {
        var sel = document.getSelection();
        sel.removeAllRanges();
        sel.addRange(storedRange)
      }, 100);
    }
  </script>
</body>
</html>

```

Основу страницы представляет небольшой фрагмент текста, предназначенный для выделения и копирования, а также блок `<textarea>`, в который можно вставить скопированный фрагмент и проверить работоспособность кода.

Объект-выделение, получаемый методом «`getSelection()`», не позволяет прямого вмешательства в свое текстовое представление. Для того чтобы модифицировать выделенный текст используем следующий алгоритм:

1. Перехватим событие «`copy`», возникающее при попытке пользователя скопировать текст (нажатие клавиш `Ctrl-C`, `Ctrl-Insert` или выбор пункта «**копировать**» из контекстного меню, вызываемого правой кнопкой мыши).
2. Сохраним информацию о текущем состоянии выделения.
3. Скопируем текст выделения в специальный невидимый контейнер и добавим к нему заметку об авторских правах.
4. Переключим объект-выделение на невидимый контейнер.
5. Закончим перехват события и запланируем отложенный запуск функции, возвращающей выделение в исходное (сохраненное) состояние.

Для чего мы используем отложенный запуск? На момент завершения работы обработчика события «`copy`» в буфер обмена будет скопировано текущее состояние объекта-выделения. Поэтому обработчик должен закончиться с активным выделением скрытого блока, а восстановить первоначальное выделение нужно уже после того, как в буфер попадет отредактированное сообщение.

Соответственно, для реализации алгоритма в документе создается блок «`<div id="copyContainer"></div>`» в свойствах которого задается большой отрицательный

отступ «`left:-1000px`», выводящий его за пределы области видимости. Скрывать блок, применяя стилевой атрибут «`display:none`», нельзя — в выделении не учитываются скрытые элементы. В нашем случае блок не скрыт, а выведен за пределы видимой части страницы.

Далее создаем переменную «`storedRange`», в которой будет сохраняться реальное состояние выделения. Поскольку данная переменная будет использоваться в разных функциях (в обработчике и в отложенной), она создается как глобальная. Нам уже известно, что объект-выделение имеет тип «`Range`» и для того чтобы создать контейнер для хранения диапазона необходимо использовать метод «`document.createRange()`».

В качестве обработчика события «`copy`» задаем функцию «`copyHandler`». В этой функции, согласно построенному алгоритму, получаем текущее состояние выделения:

```
var sel = document.getSelection()
```

и переносим данные о маркерах начала и конца выделения в переменную «`storedRange`»:

```
storedRange.setStart(sel.anchorNode, sel.anchorOffset);  
storedRange.setEnd(sel.focusNode, sel.focusOffset);
```

Затем получаем текстовое представление выделения, добавляем к нему авторскую заметку «(Copied from [js4\\_20.html](#))» и помещаем результат в блок «`copyContainer`». После этого перемещаем выделение на этот блок:

```
sel.selectAllChildren(copyContainer)
```

В конце обработчика планируем отложенный запуск функции. Используем возможность задать функцию непосредственно в команде «`setTimeout`», не вынося ее в отдельное описание. Следует помнить, что это отдельная функция, она запустится после того, как закончит работу обработчик, и в ней не будет доступа к переменной «`sel`», описанной в уже завершенной функции-обработчике. Соответственно, в теле функции заново получаем объект-выделение, сбрасываем данные о текущих маркерах и устанавливаем их из сохраненного объекта «`storedRange`»:

```
sel.removeAllRanges();  
sel.addRange(storedRange)
```

Вторым параметром для «`setTimeout`» указываем время отложенного запуска — 100 миллисекунд. За это время событие «`copy`» завершит стандартную обработку и передаст в системный буфер обмена данные о выделении, установленном на невидимый блок. После этого можно возвращать выделение в исходное состояние (рис. 30).

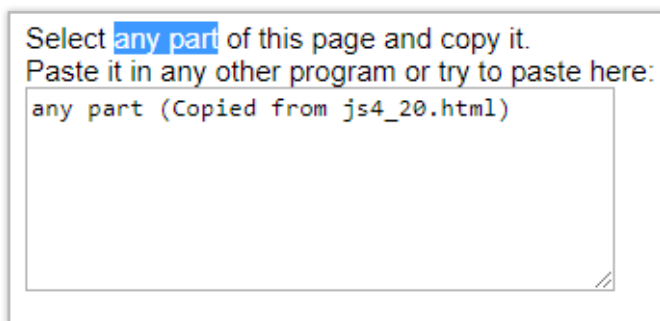


Рисунок 30

Сохраните файл и откройте его в браузере. Выделите произвольный фрагмент текста и скопируйте его. Затем вставьте его в текстовый блок, убедитесь в наличии дополнительной заметки об авторских правах (рис. 30).

Попробуйте использовать различные способы копирования и вставки, а также использовать вставку результатов копирования в другие программы.

### **Задание для самостоятельной работы.**

1. Дополните созданную программу кодами удаления содержимого (очистки) невидимого блока после завершения копирования. Это освободит память, занятую копией фрагмента, и скроет от пользователя механизм работы нашей программы.
2. Обеспечьте проверку на пустоту копируемого фрагмента. Если выделение не содержит текстовой информации, то добавлять заметку об авторстве не нужно.

В завершение раздела отметим, что в старых версиях браузера «Internet Explorer» (8й версии и ранее) для задач управления выделением применялся объект «TextRange». Его использование похоже на работу с объектом «Range», хотя и имеет ряд особенностей. В силу крайне редкого использования настолько устаревшего браузера не будем приводить детали работы с этим объектом. Современные версии браузеров «Internet Explorer» и «Edge» полностью поддерживают рассмотренные выше технологии. У браузера «Internet Explorer» осталась поддержка объектов «TextRange», в остальных браузерах, в т.ч. «Edge», работа с ними невозможна.

# Обобщение сведений об объекте Document

Начиная с самого первого урока мы уже неоднократно говорили о модели DOM, использовали различные свойства и методы объекта «`document`». В данном разделе мы соберем и обобщим основные сведения об этом объекте.

Напомним, что согласно принципам, заложенных в JavaScript, все программные объекты имеют прототип — объект или интерфейс, являющийся фундаментом, основой для работы данного объекта. В этом несложно убедиться, создав в консоли браузера пустой массив «`[]`» или пустой объект «`{}`». Раскрыв подробные сведения, выведенные в консоль как ответ на их создание, мы увидим, что даже пустой массив включает в себя определения прототипа «`Array`», а объект — прототип «`Object`». Обратите внимание, названия прототипов начинаются с большой буквы (рис. 31).

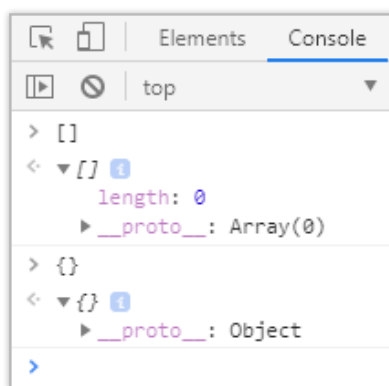


Рисунок 31

Прототипы позволяют обеспечить одинаковую функциональность для различных объектов одного типа. Например, все массивы, и даже пустой, имеют одинаковые наборы методов наподобие «[find](#)» или «[indexOf](#)». Эти методы не требуют описания или реализации при объявлении массива, они берутся из прототипа, на основе которого строятся все массивы.

Для большинства стандартизированных объектов JavaScript предусмотренные собственные прототипы. Так элемент «[Document](#)» (с большой буквы) является прототипом-интерфейсом для основного объекта DOM — «[document](#)» (с маленькой буквы). То есть «[Document](#)» описывает набор методов и свойств, а «[document](#)» их реализует практически. Если говорится, что «[Document](#)» имеет некоторую возможность (свойство, метод или атрибут), то проверить ее работу можно через объект «[document](#)».

Формально, при помощи указанного интерфейса существует возможность создать новый документ: «[document2= new Document\(\)](#)» и использовать его, например, для внутреннего фрейма страницы. Только новый документ будет создан полностью пустым и для его практического использования придется наполнить его необходимыми данными. Поэтому гораздо проще, быстрее и безопаснее использовать для создания фреймов HTML средства.

Внутренняя структура интерфейса «[Document](#)» очень сложна. Достаточно только взглянуть на нее, например, на [веб-странице](#) стандарта «DOM Living Standard», чтобы оценить насколько комплексным этот элемент явля-



ется сам по себе. Раскрывать его детали и особенности мы так или иначе будем в течение всего курса изучения JavaScript. Отметим здесь основные из них, предоставив выше ссылку на полную спецификацию стандарта.

Сложность интерфейса «[Document](#)» и его реализации — объекта «[document](#)» связана с тем, что на него возлагается практически все, что содержит в себе модель DOM. Ее сокращенную и при этом довольно непростую схему можно посмотреть в пятом разделе данного урока. Для каждого из блоков схемы существует набор методов, свойств и атрибутов интерфейса, реализующих необходимую функциональность.

Не все функции интерфейса «[Document](#)» являются популярными и широко применимыми в веб-разработке. Некоторые из них предназначены для специальных задач и практически не используются в обычных веб-страницах. Например, средства обхода деревьев ([TreeWalker](#)) или итераторов узлов ([NodeIterator](#)).

Другие же функции, наоборот, представляют основу работы с DOM и без них невозможно создать более-менее полноценный скрипт. Представьте, насколько бы усложнилась разработка, если бы не возможность изменять HTML-код в узлах при помощи атрибута «[innerHTML](#)».

Можно выделить и некоторую «середину» популярности, например, средства управления диапазонами «[Range](#)». При том, что они позволяют реализовывать дополнительный функционал, в подавляющем большинстве сайтов эти средства не используются.

Рассмотрим основные функции, реализованные в объекте «**document**», и сгруппируем их по типу решаемых задач:

### **1. Рубрикация и обеспечение отношений**

- 1.1. Выделение структурных элементов;
- 1.2. Организация дерева DOM;
- 1.3. Реализация переходов к дочерним, родительским, соседним узлам в дереве;
- 1.4. Обработка HTML тегов в узлах.

### **2. Управление деревом элементов и их свойствами**

- 2.1. Создание элементов;
- 2.2. Поиск элементов;
- 2.3. Удаление элементов;
- 2.4. Установка свойств и атрибутов.

### **3. Управление сборкой документа**

- 3.1. Расчет координат;
- 3.2. Вычисление и изменение размеров окна и элементов;
- 3.3. Прокрутка документа и содержимого элементов.

### **4. Реализация модели событий DOM**

- 4.1. Обеспечение всплытия событий и его принудительного прекращения;
- 4.2. Управление созданием, удалением и запуском обработчиков событий;
- 4.3. Реализация действий по умолчанию и возможности их отключения.

### **5. Управление диапазонами**

- 5.1. Управление выделением;
- 5.2. Создание диапазонов.

Как уже отмечалось, этот список не является полным, он содержит лишь те группы, которые используются в разработке веб-страниц. Полный перечень возможностей интерфейса «Document» можно посмотреть в документированных стандартах DOM.

Некоторые из приведенных задач уже были так или иначе рассмотрены в предыдущих уроках. Некоторые будут более детально разобраны в следующих. Часть задач рассмотрена в данном уроке. Обобщим известную нам информацию по каждой группе задач.

Группа «Рубрикация и обеспечение отношений» объединяет в себе задачи «отображения» HTML разметки на программные структуры. Так для тега `<html>` в объекте «document» создается дочерний объект «document.documentElement», для тега `<head>` — «document.head», для тега `<body>` — «document.body». Через эти объекты можно получить программный доступ к соответствующим рубрикам HTML кода. Для переходов между дочерними и родительскими элементами создаются отношения и структуры, их обеспечивающие. Они детально были рассмотрены в четвертом разделе данного урока, в качестве примера можно повторить основные из них: «childNodes», «parentNode» и «nextSibling».

Также к этой группе задач следует отнести перерубрикацию — изменение структуры объектов DOM после того, как в HTML код вносятся изменения во время работы скриптов. Это обычная практика, заключающаяся в изменениях атрибута «innerHTML» различных элементов.

Вторая группа задач «Управление деревом элементов и их свойствами» касается возможностей управления

деревом DOM без прямого внесения изменений в HTML код документа или его элементов. Можно сказать, что эта задача противоположна первой и призвана внести изменения в HTML разметку после того как структура документа будет модифицирована командами на подобие «`createElement`», «`insertBefore`» или «`removeChild`». Аналогично, после выполнения команды, меняющей атрибут «`className`» некоторого объекта, в HTML коде должна измениться запись «`class=`» у соответствующего элемента, а при присваивании стилевых атрибутов из скрипта «`element.style.color=`» изменения должны отразиться в атрибуте «`style=`».

Третья группа задач «Управление сборкой документа» содержит инструменты определения координат, размеров и взаимного расположения объектов. Среди таких инструментов можно привести примеры свойств «`clientWidth`» и «`clientHeight`», отвечающих за размеры элемента, «`offsetLeft`» и «`offsetTop`», определяющих положение левого верхнего угла (координаты) элемента, а также «`scrollLeft`» и «`scrollTop`», хранящий данные о том, насколько прокручено содержимое элемента. Некоторые свойства позволяют внесение изменений, например, увеличение свойства «`scrollTop`» приведет к дополнительному прокручиванию (скроллингу) страницы или отдельного элемента.

Четвертая группа задач «Реализация модели событий DOM» призвана обеспечить управление событиями DOM. О событиях мы детально говорили в пятом разделе данного урока, повторим несколько возможностей в качестве примера. Прежде всего реализуется возмож-

ность создания собственных обработчиков для различных событий. Для этого существует несколько способов, например, метод `«addEventListener»`.

Ряд событий в рамках данной группы задач обеспечены обработчиками по умолчанию. Они позволяют выделять текст и копировать его, перетягивать рисунки, вызывать меню при нажатии правой кнопки мыши. В некоторых случаях эти обработчики должны быть отменены, что обеспечивается методом `«preventDefault»`.

Прохождение событий связано с процессом их всплытия, что также обеспечивает данная группа задач. При необходимости есть возможность вмещаться в процесс всплытия события прекратив его командой `«stopPropagation»`.

Пятая группа задач «Управление диапазонами» чаще всего используется для работы с выделением — обработка, блокирование или дополнения выделения примечаниями об авторстве. В том числе и для задач создания выделения или эффекта, ему подобного, для того чтобы обратить внимание на определенную группу элементов на странице.

## Домашнее задание

1. Реализуйте возможность «перетаскивания» (Drag-and-Drop) для разных элементов. На начальной странице представлены несколько блоков, каждый из которых может быть перемещен при помощи мыши в произвольное место экрана.
2. Разработайте страницу для «группировки» элементов. Дополните предыдущую программу блоками-контейнерами, в которые исходные элементы могут быть перемещены. При попытке перетянуть элемент, не попав в контейнер, перемещение отменяется.
3. Разработайте страницу, содержащую таблицу с несколькими строками. Обеспечьте возможность менять порядок строк таблицы при помощи перетягивания их курсором мыши.
4. Дополните предыдущую страницу уведомлением об авторских правах. При копировании данных из таблицы к скопированному тексту должно быть добавлено «Copied from ordered table». Включите в это сообщение данные о дате и времени копирования.

