Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery

STEP computer ACADEMY

Урок № 3

Обработка событий, Browser Object Model, Document Object Model

Содержание

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

Обработка событий

Что такое событие и обработчик события?

Представьте себе, что Вы общаетесь с собеседником и в это время звонит телефон. Звонок важный и на него необходимо ответить. Поскольку вести два разговора одновременно невозможно, Вы приостанавливаете текущую беседу и говорите по телефону.

Тема телефонного разговора, наверняка, будет отличаться от предмета вашей беседы, то есть Вам надо «переключиться» на другую тему, вспомнить детали, события, имена и т.п. После окончания телефонного разговора Вы возвращаетесь к предыдущей беседе с того места, в котором она была прервана звонком, «переключившись» обратно к теме беседы, ее деталям и особенностям.

Аналогичным образом устроен компьютер. У него есть центральный процессор, который может в один момент времени выполнять только одну команду. Однако, ему постоянно «звонят» — двигается мышь, нажимаются кнопки на клавиатуре, приходят данные по сети и т.д. Процессору приходится «отвечать» на эти «звонки» — прекращать работу с одной программой, переключаться на обработку другой, и снова возвращаться к предыдущей программе. Такие «звонки» получили название событий.

Понятие «событие» имеет ряд оттенков и особенностей для различных областей использования. Для того чтобы разобраться с этими особенностями, рассмотрим детальнее, какие процессы будут происходить в ком-

пьютере, когда Вы включаете блокнот и печатаете в нем некоторую заметку.

Блокнот является обыкновенной программой, выполняемой операционной системой. Процессор компьютера исполняет инструкцию за инструкцией, обеспечивая отображение окна блокнота, его элементов, а также набранного ранее текста.

В некоторое время на клавиатуре нажимается кнопка. Процессор компьютера прекращает выполнять коды программы «блокнот», сохраняет необходимые данные и переходит к обработке сигнала от клавиатуры. Для этого в памяти компьютера существует специальная программа (драйвер), определяющая, какая кнопка была нажата, и что при этом необходимо сделать. Некоторые кнопки отвечают за набор символов, тогда как другие могут управлять громкостью динамиков или яркостью монитора. Предположим, что была нажата символьная кнопка. В таком случае определяется, какой символ соответствует кнопке, и данная информация сохраняется в системном буфере обмена.

После того как сигнал от клавиатуры будет обработан, процессор вернется к выполнению программы «блокнот», восстановит сохраненные ранее данные, и напечатает символ, который был сохранен в буфере при работе драйвера клавиатуры.

А что значит напечатает? Программа поместит код символа в необходимое место видеопамяти. Затем процессор, аналогично описанному выше способу, переключится на задачу передачи видеоданных на монитор, который уже и отобразит новый символ.

Процессы переключения процессора на выполнение различных задач называется термином «прерывание» (*англ.* — interruption). Прерывания — это события системного уровня, «звонки» от подключенных к компьютеру устройств.

Прерывания обеспечивают работу клавиатуры, мыши, дисков, монитора и других периферийных устройств компьютера. Для разных прерываний предусмотрены разные программы, отвечающие за их обработку. Они так и называются «обработчики прерываний». Эти программы загружаются при включении компьютера и старте операционной системы.

Обработка прерываний происходит быстро, создавая ощущение, что мы просто печатаем текст в блокноте. Тем не менее, пока мы печатаем происходит множество прерываний — идут часы, мигает курсор, работают диски, проверяется электронная почта и т.п.

Идея прерываний, заложенная в основу работы компьютера, нашла применение и в разработке прикладных программ, в том числе веб-страниц. Только в этой области вместо прерывания используется термин «событие» (*англ.* — event).

Событие — это некоторое происшествие, запускающее специальную программу — обработчик события. Вернее, правильнее сказать «подпрограмму», поскольку обработчик события не является отдельной программой (как обработчик прерывания), а входит в состав одной, основной программы. Обычно, обработчики событий — это отдельные функции (в программном понимании функции, как именованной подпрограммы).

Понятие события тесно связано с понятием сообщения (*англ.* — message). Их даже иногда смешивают, определяя событие как сообщение. Тем не менее, между ними есть определенная разница. Когда Вы слышите звук звонящего телефона — Вы получаете «сообщение». Это сообщение сопровождается дополнительной информацией — на экране телефона отображается номер звонящего или его имя, пусть для примера, Адам. Событием становится Ваш вывод: «мне звонит Адам». Разница тонкая: сообщение — это звук звонка, а событие — указанный вывод. Однако, требуются определенные действия для того, чтобы, услышав звук звонящего телефона, прийти к выводу «мне звонит Адам».

Рассмотрим соотношение сообщений и событий более подробно. Механизм обмена сообщениями является неотъемлемой частью операционной системы. Различные программы могут посылать друг другу сообщения, сообщениями могут обмениваться части одной программы, сообщения могут поступать в программу прямо от операционной системы. Каждая программа имеет у себя очередь сообщений, которые обрабатываются одно за другим или накапливаются, если не успевают обрабатываться. Как если во время одного телефонного разговора Вам поступит еще один «параллельный» звонок.

Операционная система имеет свой набор стандартных сообщений, например, движение мыши или нажатие ее клавиш, переключение между окнами, изменение их размеров или положения, нажатие кнопок клавиатуры и т.д. Прикладные программы могут создавать другие (дополнительные) сообщения. Например, браузеры до-

полняют системные сообщения собственным набором, необходимым именно для веб-страниц.

Обработка сообщения, принятие решение о его типе, анализ дополнительных данных уже называется событием. Когда доходит очередь до обработки определенного сообщения, тогда и возникает событие, — программа определяет, какую функцию (подпрограмму) необходимо запустить для обработки данного события. Если для события обработчика нет, то такое сообщение просто игнорируется и удаляется из очереди. Можно сказать, что сообщение — это понятие системное, межпрограммное, тогда как событие — понятие внутреннее, действующее только для данной программы.

Более существенную разницу между сообщением и событием можно привести на следующем примере. Вы ответили на телефонный звонок, а звонящий Адам попросил пригласить к телефону Вашего собеседника, с которым Вы говорили до звонка. В данном случае звонок является сообщением, основным событием — принятое решение «мне звонит Адам», а дополнительным событием становится Ваше действие, приглашающее собеседника к телефону.

С точки зрения веб-программиста, детальный анализ процесса формирования сообщений можно пропустить. Эти сведения больше нужны для разработчиков системных программ. К тому же, как уже отмечалось выше, браузер видоизменяет системный набор сообщений, поэтому перейдем к рассмотрению событий, которые будут возникать в веб-страницах. В JavaScript принято обращаться к событиям по их именам (в операционной системе — по кодам). Имена содержат в себе подсказку о происхождении данного

события. Например, событие с именем «click» возникает, когда по элементу совершают щелчок мыши.

Полный перечень возможных событий достаточно большой. Ознакомиться с ним можно, например, на странице. В качестве обобщения события можно условно разделить на несколько групп:

- События интерфейса пользователя;
- События жизненного цикла;
- Индивидуальные события.

События интерфейса пользователя возникают за счет активности пользователя веб-страницы. Это движения курсора мыши, нажатие кнопок клавиатуры и кнопок, нарисованных на странице, прокрутка страницы или содержимого элемента и т.д.

События жизненного цикла посылаются элементам при их создании, загрузке, в том числе ошибках загрузки, при получении системных сообщений, переходу в online или offline режим, запуске анимации и т.п.

Индивидуальные события характерны только для определенных объектов и для объектов другого типа не используются. Примером может быть событие «pause», возникающее при остановке воспроизведения (паузе) медиа-контента. Очевидно, что это событие касается только медиа-контейнеров. Событие «input» возникает, когда элемент получает ввод от пользователя, то есть в этот элемент добавляется текст, который печатается на клавиатуре. Такое событие актуально только для элементов, в которых можно что-то вводить (печатать). Подобные события есть и у других групп элементов.

Обработка событий в сценариях

Для всех стандартных событий у элементов страницы предусмотрены обработчики. Традиционно их имена формируются из префикса «on» и имени события. Например, обработчик события щелчка мыши «click» будет иметь название «onclick».

Существует несколько способов определить тело для обработчика событий. Первый — это указать его как HTML-атрибут при объявлении элемента. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_1.html*)

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Events</title>
    <style>
        div {
            background: navy;
            height: 50px;
            width:  100px;
        }
    </style>
</head>
<body>
    <div onclick="alert('DIV have been clicked')"></div>
</body>
</html>
```

Основным содержимым страницы является блок (div), для которого в заголовочной части указаны стилевые атрибуты: ширина, высота и фоновый цвет. В теле

документа, при объявлении блока указан атрибут «onclick», являющийся обработчиком события щелчка мыши. В значении данного атрибута указываются инструкции JavaScript, которые будут выполнены при наступлении события, то есть при щелчке мыши.

Обратите внимание на необходимость чередования кавычек разного типа для всего значения атрибута «onclick» и для текста сообщения диалогового окна «alert». Подобные ситуации возникают довольно часто при внедрении JavaScript инструкций:

- Во-первых, сам код обработчика должен быть заключен в кавычки: <div onclick="…".
- Во-вторых, текст сообщения для команды «alert» также требует кавычек: alert('…').

Если применять кавычки одного типа, то возникнет ошибка, так как внутренние кавычки будут закрывать внешние (для наглядности добавлены дополнительные отступы):

```
<div onclick="alert("DIV have been clicked")" ->
<div onclick="alert("   DIV have been clicked   ")"
```

Для сложных инструкций современный стандарт языка допускает применение трех типов кавычек. Напомним их:

- Одиночные прямые кавычки: '…';
- Двойные кавычки: "…";
- Одиночные обратные кавычки: `…`.

Для текстовых выражений все три типа кавычек полностью эквивалентны. Отличия заключаются в том, что в обратных кавычках можно подставлять значения переменных, например, предположим, что в скрипте объ-

явлена переменная «x=10», тогда следующие выражения приведут к соответствующим результатам:

```
"x=${x}"   ->   x=${x}
'x=${x}'   ->   x=${x}
'x=${x}'   ->   x=10
```

Сохраните созданный файл и откройте его в браузере. Наведите курсор мыши на синий блок и щелкните по нему левой кнопкой мыши. В результате должно появиться сообщение, указанное в обработчике (рис. 1).

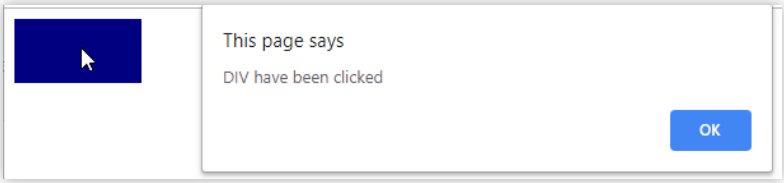


Рисунок 1

Выполните щелчок, отведя курсор мыши за пределы блока. Убедитесь, что в этом случае сообщение не появляется. Это свидетельствует об адресации событий. Кроме того, что событие возникает само по себе как факт «системой зафиксирован щелчок мыши», это событие имеет цель (*англ.* — target) — элемент, для которого это событие предназначено. Другие объекты это событие не получают, в чем мы убедились, щелкая мышью за пределами блока.

Второй способ определить обработчик события — это указать одноименный метод («onclick» в нашем примере) в скриптовой части кода. Внесите в созданный файл следующие изменения (*код с изменениями доступен в папке Sources, файл JS_3_2.html*):

```
<body>
    <div id="clickableDiv"></div>
    <script>
        clickableDiv.onclick=function(){
            alert('DIV have been clicked')
        }
    </script>
</body>
```

Блоку (div) присваивается идентификатор id="clickableDiv". По этому идентификатору данный блок становится доступным в скриптовой части, где для него указывается обработчик события «clickableDiv.onclick». В качестве значения задается функция, выполняющая те же действия, что и в предыдущем примере.

Сохраните изменения, откройте файл в браузере или обновите открытую страницу. Убедитесь в том, что события обрабатываются точно также, как и ранее.

Третий способ задать обработчик события для элемента — это применить специальный метод «addEventListener». Внесите следующие изменения в скриптовую часть документа (*код с изменениями доступен в папке Sources, файл JS_3_3.html*):

```
<script>
    clickableDiv.addEventListener("click", function(){
       alert('DIV have been clicked')
    })
</script>
```

Блок все также управляется при помощи своего идентификатора «clickableDiv», только вместо метода «onclick»

вызывается метод «addEventListener». В качестве аргументов для него передаются два значения — имя события и функция-обработчик. Обратите внимание, в данном случае передается имя события «click» (без префикса «on», используемого для имени обработчика). Тело функции обработчика сохранено из предыдущего примера.

Сохраните изменения, откройте файл в браузере или обновите открытую страницу. Убедитесь в том, что работоспособность кода не изменилась.

Любым из описанных способов можно создать обработчик для произвольного события. Принципиальных отличий в различных способах нет. Первый способ, определяющий тело обработчика непосредственно в HTML теге, может подойти для небольших программных инструкций, не загромождающих разметку элементов. Для более объемных кодов удобнее отделять их в самостоятельные функции и подключать к элементам вторым или третьим способом. Единственное, что при этом следует отметить, так это то, что смешивать несколько способов не допускается. Если определить обработчик в HTML теге, а затем в скриптовой части снова указать функцию для того же события, то новое значение будет использовано вместо старого. Другими словами, новое тело обработчика сотрет старое и займет его место. Если обработчик устанавливается несколько раз, то актуальным будет тот, который установлен последним.

Объект event и его свойства

При обработке события часто бывает необходима дополнительная информация, связанная с возникновением самого события. Например, событие нажатия клавиши

на клавиатуре возникает при нажатии любой клавиши. Очевидно, что программы должны по-разному реагировать на нажатия разных клавиш, а значит в обработчик события эта информация должна быть передана.

При обработке системного сообщения и создании программного события браузер формирует специальный объект «event», в котором собираются все данные о событии. В теле обработчика этот объект может использоваться для получения этой дополнительной информации.

Информация в объекте «event» зависит от типа события, которое обрабатывается. Логично, что для событий мыши не нужны данные о кнопке клавиатуры, а для событий клавиатуры не нужны координаты указателя мыши.

В то же время, объект «event» имеет свойства, определяемые для всех типов событий. Одним из таких свойств является целевой объект события (target). Выше мы уже отмечали, что любое событие имеет своего адресата. Рассмотрим пример, иллюстрирующий работу с объектом «event» и его свойством «target». Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_4.html*). Стилевые определения могут быть скопированы из предыдущих примеров.

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Events</title>
    <style>
        div {
```

```
            background: navy;
            height: 50px;
            width:  100px;
        }
    </style>
</head>

<body onclick = "out.innerText =
                 'Click detected over: ' +
                  event.target.nodeName">
    <div></div>
    <p id="out"></p>
</body>
</html>
```

Аналогично с рассмотренными ранее кодами, в теле документа располагается блок, цвет и размеры которого задаются стилями в заголовочной части. Дополнительно после блока размещается абзац (параграф) с идентификатором «out». Он будет служить нам контейнером для вывода сообщений.

В отличие от предыдущих примеров, обработчик события «onclick» определен в теге «body». Это позволяет перехватывать все события, относящиеся к данному документу. В теле обработчика формируется сообщения из текста «Click detected over:», к которому добавляется имя целевого объекта события «event.target.nodeName».

Сохраните документ и откройте его в браузере. Выполните щелчки мыши по блоку и за его пределами. Обратите внимание на изменения сообщения о целевом объекте. Выполните щелчок над самим текстом сообщения. Убедитесь, что целевой объект меняется и в этом случае.

Рисунок 2

Рисунок 3

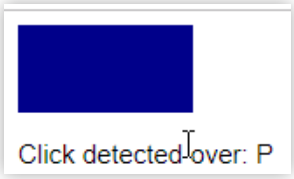


Рисунок 4

Как видно из примера, объект «event» не требует специального описания. Он создается браузером и передается в обработчик, уже заполненный необходимыми данными. Для того чтобы узнать, какие данные доступны для конкретного события, можно вывести объект «event» в консоль разработчика. Дополните код обработчика события инструкцией «console.log(event)», обработчик должен принять следующий вид:

```
<body onclick = "out.innerText =
                 'Click detected over: ' +
                 event.target.nodeName;
                 console.log(event)">
```

Сохраните изменения, обновите страницу браузера. Откройте в браузере панель разработчика (кнопка F12), переключитесь на вкладку «Console» (если она сразу не активна). Переведите курсор мыши на блок и выполните щелчок. В консоли раскройте детали сообщения. Результат должен быть подобен приведенному на рисунке 5 (возможны некоторые отличия при использовании разных браузеров, пример соответствует браузеру «Chrome»)

Рисунок 5

Не будем детально описывать значения всех полей, остановимся на тех, которые наиболее часто находят применение в веб-разработке. Конечно же таковыми являются координаты курсора мыши в момент совершения щелчка и сведения о целевом объекте.

В объекте «event» можно обнаружить несколько различных пар значений для координат. Среди них есть стандартизированные:

- screenX, screenY — координаты курсора относительно экрана монитора;
- pageX, pageY — координаты относительно начала веб-страницы;
- clientX, clientY — координаты относительно клиентской части окна браузера.

Дополнительные пары координат являются экспериментальными или не стандартизированными. При их использовании необходимо убедиться в поддержке данным типом браузера. Можно даже сказать, что их использование не рекомендуется, из-за отличий для разных браузеров.

- layerX, layerY — координаты относительно целевого элемента, имеющего позиционирование (стилевое свойство «position» которого имеет значение, отличное от «static»);
- offsetX, offsetY — координаты относительно любого целевого элемента;
- x, y — псевдонимы для clientX, clientY.

Как видно из анализа числовых данных, браузер «Chrome» в качестве layerX и layerY использует значения из пары (x, y), что не соответствует целевому элементу. Если запустить

данный пример в браузере «Firefox», то можно убедиться, что в нем для координат offsetX и offsetY указываются нули независимо от положения курсора. До тех пор, пока не будет установлен стандарт для указанных выше полей, их использование крайне нежелательно.

Отличия между способами отсчета стандартных координат иллюстрирует следующий рисунок:

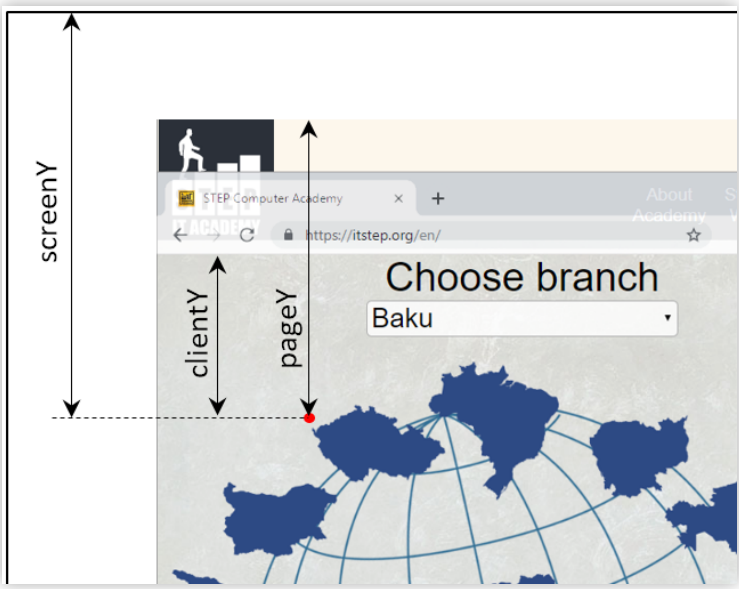


Рисунок 6

Черными линиями на рисунке 6 показаны границы экрана (монитора). На экране изображено окно браузера, в котором открыт сайт. Этот сайт прокручен, поэтому в окне отображается лишь его часть. Скрытая часть также приведена на рисунке, хотя в реальной ситуации её, естественно, не видно. Предположим, что событие щелчка мыши произошло в точке, отмеченной красной точкой.

Координата screenY точки будет отсчитана от границ самого экрана. Это системная координата, не привязанная ни к какому из окон. Координата clientY берет отсчет от окна браузера, точнее, от его клиентской части, которая не учитывает адресную строку, заголовок вкладки, кнопки управления размерами окна браузера. Значение pageY определяется относительно начала веб-страницы. В данном случае это значение будет больше, чем clientY, т.к. начало страницы находится выше края окна за счет прокрутки. Если страница не прокручена, то значения координат clientY и pageY совпадают.

Полностью аналогично определяются координаты точки события screenX, clientX и pageX, только их отсчет происходит по горизонтальной оси.

Следует отметить еще одну особенность использования объекта-события. Объект «event» автоматически связывается с параметром функции-обработчика, то есть служит ее аргументом. В том случае, когда обработчик события определяется вне HTML-тега, для получения данных о событии можно указать параметр функции обработчика. Не обязательно использовать для него имя «event», чаще всего для экономии места его обозначают просто как «e».

```
element.onclick = function(e){…}
element.addEventListener("click", function(e){…})
```

В теле функции параметр «e» является тем самым объектом «event», который сопровождает событие. Координаты точки события можно узнать применяя запись, на подобие «e.pageX», а целевой объект — «e.target».

Как уже отмечалось выше, данные в объекте «event» зависят от типа события, которое передается на обработку. В данном разделе мы рассмотрели только одно событие, сопровождающее щелчок мыши. О свойствах «event», которые характеризуют другие события, мы будем говорить дальше при детальном рассмотрении соответствующих событий. В любом случае, проверить состав объекта «event» можно путем вывода его в консоль разработчика.

Задание для самостоятельной работы. Реализуйте функциональность рассмотренного выше примера по определению целевого объекта события при помощи второго и третьего способов установки обработчика события (element.onclick и element.addEventListener).

Управление стилями элементов web-страницы

В предыдущих упражнениях в ответ на поступление события мы использовали текстовые сообщения, выводя их прямо на страницу или в диалоговые окна. Гораздо больший спектр возможностей реагирования на события предоставляет управление стилями — цветом, размером, положением элементов и многим другим. Это позволит более гибко обеспечить взаимодействие нашей веб-страницы с посетителем.

При помощи управления стилями можно создать ощущение, что сайт каким-либо образом «откликается» на действия пользователя. Если он забыл заполнить нужное поле, то сайт не просто выдаст сообщение, напоминающее о необходимости ввода данных, а еще и выделит пропущенный элемент, например, красной

рамкой. Также управление стилями может обеспечить эффекты анимации, делая страницу динамичной и привлекательной.

Для иллюстрации принципа управления стилями модифицируем рассмотренный в предыдущих разделах пример следующим образом: при щелчке мышью синий блок должен перемещаться в точку, где находится курсор мыши.

Приведем текст документа и далее рассмотрим принцип его работы. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_6.html*)

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Events</title>
    <style>
        body {
        position: relative;
        height: 2000px;
    }
        div {
            background: navy;
            height: 50px;
            width:  100px;
            position:absolute;
        }
    </style>
</head>

<body onclick="moveStranger(event)">
    <div id="stranger"></div>
```

```
    <script>
        function moveStranger(e){
            stranger.style.left=e.pageX+'px';
            stranger.style.top =e.pageY+'px'
        }
    </script>
</body>

</html>
```

Для того чтобы поместить блок в заданную позицию можно воспользоваться двумя технологиями:

а) указать отступы (margin) для блока или

б) задать блоку абсолютное позиционирование и указать координаты (left, top).

Если блок на странице один, то методы полностью эквивалентны. Однако, когда на странице есть другие элементы, изменение отступов блока (вариант а) приведет к «толканию» соседних элементов. Поэтому, хотя блок у нас на странице будет один, используем для практической реализации вариант б).

В стилях блока, по сравнению с предыдущими примерами, добавлено определение «position:absolute». Для того чтобы использовать абсолютные координаты элемента, необходимо указать от чего эти координаты будут отсчитываться. Контейнером для блока является тело документа «body», поэтому указываем в его стилях «position: relative». Дополнительно укажем большую высоту для тела «height: 2000px», это даст нам возможность прокручивать страницу вверх-вниз для проверки правильности выбора координат

из объекта-события (мы помним из предыдущего раздела, что их есть как минимум три пары).

В теле документа все так же присутствует один блок, теперь с идентификатором «stranger». По этому имени блок будет доступен в скриптах. При определении тела (body) устанавливается обработчик события. Для сокращения записей в HTML-теге использовано имя дополнительной функции «moveStranger», в которую передается объект «event» (он нужен для определения координат). В скриптовой части приводится тело функции «moveStranger», принимающей параметр «e».

При запуске функции происходит изменение стилевых определений блока «stranger». Как видно из кодов, доступ к этим стилям обеспечивает объект «stranger.style». Для стилевого атрибута «stranger.style.left» формируется значение из координаты события «e.pageX» и единицы измерения ('px'). Помним, что единицы измерения обязательны для указания длин в стилях, тогда как в объекте «event» присутствуют только числовые значения. Аналогично устанавливается стиль «stranger.style.top» из координаты «e.pageY».

Сохраните файл и откройте его в браузере. Щелкните мышью в произвольной точке страницы, убедитесь, что блок перемещается в точку щелчка. Прокрутите страницу и повторите щелчок. Блок все также должен следовать за курсором мыши. Это свидетельствует о правильном выборе координат из объекта «event». В качестве проверки можете использовать другую пару координат из объекта «event» (например, clientX, clientY) и убедиться, что при прокрутке окна работа нарушается. Верните коды к исходному состоянию.

Присмотритесь внимательно к взаимному размещению блока и курсора мыши (рис. 7).

Рисунок 7

между острием курсора и углом блока присутствует определенный отступ. Это внешний отступ (margin) тела документа (body). Обычно, он устанавливается в значение «8px» и создает небольшую рамку, отделяющую страницу от границ окна браузера. В нашем случае этот отступ приводит к неправильному отсчету координат, т.к. начало страницы и границы окна изначально не совпадают.

Добавьте стилевое определение «margin:0» для элемента «body» (для нулевого значения единицы измерения разрешается не указывать). Сохраните файл и обновите страницу в браузере. Убедитесь, что блок стал следовать за курсором мыши более точно, совпадая левым верхним углом с острием указателя.

Как нам уже известно, любой элемент, созданный HTML разметкой, имеет свое представление в виде программного объекта, доступного по названию своего идентификатора (атрибута «id»). Пусть, для примера, в нашем документе присутствует элемент с атрибутом «id="element"».

Все стилевые атрибуты этого элемента собраны в коллекции «element.style». Как и любая другая коллекция

в JavaScript, она предусматривает два способа доступа к своим компонентам — при помощи объектного синтаксиса

```
element.style.left
```

или при помощи синтаксиса доступа к массивам

```
element.style["left"]
```

Оба эти способа эквиваленты, использовать можно любой из них по Вашему предпочтению. Однако, отличия все же наблюдаются для составных имен стилевых атрибутов.

В CSS принято, что знак «-» является допустимым в именах атрибутов, например, «background-color». При этом в JavaScript данный знак обозначает арифметическую операцию (вычитание) и не может быть частью имени объекта или его свойства. В то же время, на ключи массива данное ограничение не распространяется. Это значит, что запись

```
element.style["background-color"]
```

является допустимой и правильной, тогда как

```
element.style.background-color
```

представляет собой операцию вычитания, а не доступ к полю «background-color» объекта.

Для обеспечения работы с составными именами стилевых атрибутов в объектном синтаксисе используется стиль «lowerCamelCase», применяемый и для других имен

в JavaScript. Знаки «-» из названий стилевых атрибутов убираются и каждое новое слово (кроме первого) начинают с заглавной буквы. При использовании синтаксиса доступа к коллекции, как к массиву, названия атрибутов точно такие же, как и в CSS определениях. В следующей таблице приведено несколько примеров использования коллекции «element.style» при помощи различных синтаксисов

Синтаксис массивов	Синтаксис объектов
element.style["background-color"]	element.style.backgroundColor
element.style["margin-top"]	element.style.marginTop
element.style["list-style-type"]	element.style.listStyleType
element.style["border-top-right-radius"]	element.style. borderTopRightRadius

Любые изменения, внесенные в коллекцию «element.style», сразу же поменяют стиль элемента, независимо от того, какой синтаксис был использован. Дополнительных действий по фиксации новых стилей совершать не нужно — изменения сразу же вступают в силу после записи в коллекцию стилей. То есть инструкция

```
element.style["background-color"] = "tomato"
```

сама-по-себе поменяет фоновый цвет элемента, никаких обновлений или перерисовок окна запускать не требуется. В коллекции «element.style» предусмотрены все возможные стилевые атрибуты, которые доступны и в CSS.

Продемонстрируем способ программного управления фоновым цветом блока на следующем примере. Мы хотим, чтобы щелчок мыши на блоке приводил к изменению его цвета. Уточним, что цвет должен формироваться

случайным образом без какой-либо предварительной закономерности.

Приведем текст документа и далее рассмотрим принцип его работы. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_7.html*)

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Random color</title>
    <style>
        div {
           background: navy;
           height: 255px;
           width:  255px;
        }
    </style>
</head>
<body >
    <div id="colorBlock"></div>
    <script>
      colorBlock.addEventListener("click", function(){
        colorBlock.style.backgroundColor =
             "rgb(" +
             Math.round(255*Math.random()) +
             "," +
             Math.round(255*Math.random()) +
             "," +
             Math.round(255*Math.random()) +
             ")" ;
        })
    </script>
</body>
</html>
```

Основу страницы все так же представляет один блок, изначально синего цвета. Для доступа к нему при помощи JavaScript, ему указан атрибут «id="colorBlock"».

В скриптовой части документа для данного блока устанавливается обработчик события щелчка мыши при помощи метода «colorBlock.addEventListener». Отсутствие специального алгоритма для формирования цвета позволяет нам использовать в качестве обработчика события функцию без параметра «function()».

В теле функции-обработчика мы формируем случайный цвет при помощи CSS функции «rgb». Для этого мы вызываем генератор случайных чисел «Math.random()». Он генерирует дробные случайные числа в диапазоне от 0 до 1.

Как нам известно из курса HTML/CSS, в качестве цветовых компонентов для «rgb» нужны целые числа от 0 до 255, поэтому полученные от генератора случайные числа мы умножаем на 255 и округляем функцией «Math.round». Инструкция получения целого случайного числа, подходящего для цветового компонента, приобретает вид

```
Math.round(255*Math.random())
```

Далее, три таких инструкции объединяются в одну строку. Ее конечный вид должен соответствовать формату «rgb(127,201,57)», то есть между числами добавляются запятые, а в начале и конце — круглые скобки. Вместо чисел вставлены описанные выше выражения.

Результирующая строка помещается в поле стилевой коллекции блока, ответственного за фоновый цвет «colorBlock.style.backgroundColor». Сохраните файл и откройте его при помощи браузера. Наведите курсор мыши

на блок и совершите щелчок левой кнопкой мыши. Убедитесь в том, что цвет блока меняется при каждом щелчке случайным образом.

В качестве следующего примера модифицируем созданную программу. Зададим алгоритм формирования цвета: пусть красный компонент всегда будет равен среднему значению (127), зеленый компонент будет зависеть от координаты «X» курсора мыши — если курсор находится возле левого края блока, то его значение близко к нулю, если возле правого, то максимальное. Синий компонент аналогичным образом будет зависеть от координаты «Y» курсора мыши.

Для большей оперативности потребуем, чтобы изменения цвета происходили при каждом перемещении курсора без необходимости нажатия кнопок мыши.

Проведем анализ новых условий. Во-первых, вместо случайных чисел необходимо использовать сведения о положении курсора. Соответственно, обработчик события должен принимать параметр, отвечающий за дополнительные сведения о событии (объект «event»).

Во-вторых, обработчик необходимо привязать к другому событию, возникающему при любом перемещении мыши. Это событие имеет имя «mousemove».

В-третьих, воспользуемся некоторыми упрощениями для исключения сложных расчетов. Нам известно, что максимальное значение цветового компонента — это число «255». Ограничим размеры блока этими значениями. Обратим внимание, что блок уже имеет нужный нам размер. Однако, как мы разбирали в одном из предыдущих примеров, необходимо еще убедиться, что

сама страница (и блок вместе с ней) не будет смещена относительно окна браузера, иначе координаты курсора в блоке будут отличаться от желаемого диапазона. Для того чтобы предотвратить смещения, укажем стилевой атрибут «margin: 0» для тела документа «body».

Внесите описанные изменения в документ. Результат должен быть подобен приведенному далее коду (*код с изменениями также доступен в папке Sources, файл JS_3_8.html*)

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Changing color</title>
    <style>
        body {
            margin: 0;
        }
        div {
            background: navy;
            height: 255px;
            width:  255px;
        }
    </style>
</head>

<body >
    <div id="colorBlock"></div>
    <script>
        colorBlock.addEventListener("mousemove",
                                    function(e){
            colorBlock.style.backgroundColor =
                 "rgb(127," +
                 e.pageX +
```

```
                    "," +
                    e.pageY +
                    ")" ;
        })
    </script>
</body>
</html>
```

Как видно из кода, подбор размеров блока, и коррекция смещений позволяет в качестве цветовых компонент указать координаты курсора «e.pageX» и «e.pageY» без дополнительного их пересчета. Для смещенного блока или при других его размерах необходимо будет обрабатывать эти значения.

Сохраните изменения и обновите страницу браузера. Наведите курсор мыши на блок, убедитесь, что его цвет изменяется в зависимости от положения курсора (рис. 8). Поскольку алгоритм изменения цвета стал детерминированным, результаты должны быть одинаковыми для различных браузеров и при разных запусках, чего не наблюдалось при использовании случайных чисел.

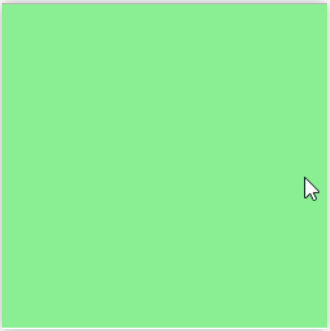


Рисунок 8

Задание для самостоятельной работы. Создайте программу, которая будет увеличивать или уменьшать размеры блока — левая верхняя точка блока всегда будет находиться в начале страницы (в левом верхнем углу), а другая точка будет задаваться курсом мыши. При щелчке мышью блок должен заполнить область от начала страницы до текущего положения курсора.

События интерфейса пользователя

События интерфейса пользователя (User Interface, UI events) представляют собой группу событий, возникающих вследствие активности человека, просматривающего нашу веб-страницу (пользователя). Можно охарактеризовать эту группу событий «от обратного» — если пользователь отойдет от компьютера, то эти события перестанут поступать в программу.

В следующей таблице приводится описание основных событий интерфейса пользователя. Полный перечень и детальное описание можно также посмотреть в стандартах, например, по ссылке. События имеют имена, отражающие смысл или условие их возникновения, что облегчает их использование и запоминание.

Событие	Условие возникновения
События мыши	
auxclick	Щелчок дополнительной кнопки мыши (если есть)
click	Щелчок левой кнопкой
dblclick	Двойной щелчок
mousedown	Нажата кнопка мыши

Событие	Условие возникновения
mouseenter	Курсор мыши зашел в пределы элемента (получает верхний элемент)
mouseleave	Курсор мыши вышел за пределы элемента (получает верхний элемент)
mousemove	Курсор мыши переместился
mouseout	Курсор мыши вышел за пределы элемента (получают все нижние элементы)
mouseover	Курсор мыши зашел в пределы элемента (получают все нижние элементы)
mouseup	Отпущена кнопка мыши
wheel	Повернуто колесо мыши
События клавиатуры	
keydown	Нажата кнопка клавиатуры
keyup	Отпущена кнопка клавиатуры
keypress	Нажата и затем отпущена символьная кнопка
События устройств ввода текста	
compositionstart	Начало формирования текста
compositionupdate	Обновлено содержание текста
compositionend	Завершено формирование текста
События ввода данных	
beforeinput	Посылается перед обновлением содержания элемента
blur	Элемент потерял фокус ввода
focus	Элемент получил фокус ввода
focusin	Посылается перед получением фокуса
focusout	Посылается перед потерей фокуса
input	Посылается после обновления содержания элемента

Рассмотрим более подробно некоторые тонкости, связанные с событиями данной группы. Среди событий мыши есть две пары очень похожих событий: «mouseenter» и «mouseover» возникают, когда курсор мыши входит в область, принадлежащую данному элементу, «mouseleave» и «mouseout» — когда курсор выходит из этой области.

Различие между этими событиями заключается в том, что события «mouseenter» и «mouseleave» получает только самый «верхний» элемент, тогда как другая пара событий «mouseover» и «mouseout» передается всем элементам, находящимся под курсором мыши. Данный эффект называется всплытием событий (*англ.* bubbling или propagation). При использовании стандарта из приведенной выше ссылки обращайте внимание на атрибут «Bubbles»: если для него указано «No», то значит данное событие не всплывает — не передается следующему элементу. Если значение атрибута «Yes», то такое событие предусматривает всплытие.

Процесс всплытия можно продемонстрировать следующим рисунком (рис. 9). Предположим, что на нашей веб-странице есть два блока, вложенных один в другой. Внутренний блок находится «выше», то есть он первый принимает сообщения от мыши. Стрелками приблизительно указаны точки возникновения событий (на самом деле события возникают на границе блока). Стрелка, иллюстрирующая невсплывающие события «mouseenter» и «mouseleave», заканчивается на верхнем блоке. Другая стрелка, отвечающая за всплывающую пару «mouseover» и «mouseout», продолжается после верхнего блока и транслирует события нижнему блоку.

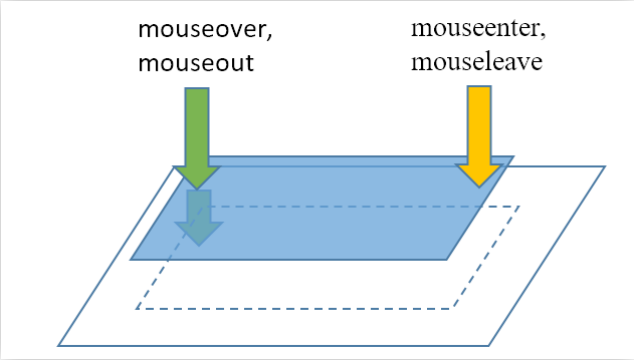


Рисунок 9

Для практического анализа описанных процессов сделаем страницу с вложенными блоками и обработчиками для событий разного типа. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_9.html*)

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Bubbling basics</title>
    <style>
        #d1 {
        border:1px solid navy;
            height: 200px;
            width:  200px;
        }
    #d2 {
        background: skyblue;
        height: 100px;
        margin: 50px;
        width:  100px;
```

```
    }
    </style>
</head>

<body>
    <h1>Mouse enter / Mouse over</h1>
    <div id="d1">
         <div id="d2"></div>
    </div>
    <p>Mouse enter <span id="s1"></span></div>
    <p>Mouse over <span id="s2"></span></div>
    <script>
        d1.onmouseenter = function(){
            s1.innerText = "d1"
            console.log("d1.onmouseenter")
        }
        d1.onmouseover = function(){
            s2.innerText = "d1"
            console.log("d1.onmouseover")
        }
        d2.onmouseenter = function(){
            s1.innerText = "d2"
            console.log("d2.onmouseenter")
        }
        d2.onmouseover = function(){
            s2.innerText = "d2"
            console.log("d2.onmouseover")
        }
    </script>
</body>

</html>
```

Основу страницы составляют два блока с идентификаторами «d1» и «d2». Блок «d2» вложен в блок «d1».

При помощи стилей внешнему блоку устанавливается рамка, внутреннему — фоновый цвет. Также задаются размеры блоков. После блоков следуют текстовые параграфы, предназначенные для вывода информации о событиях.

В скриптовой части для блоков устанавливаются обработчики событий «onmouseover» и «onmouseenter». В них данные о событиях записываются в соответствующие элементы текстовых абзацев, а также дублируются в консоль.

Сохраните файл и откройте его в браузере. Поместите курсор мыши справа от блоков и начните медленно двигать его в сторону блоков. Обратите внимание на изменения, происходящие после пересечения курсором границ внешнего и внутреннего блоков (рис. 10, 11).

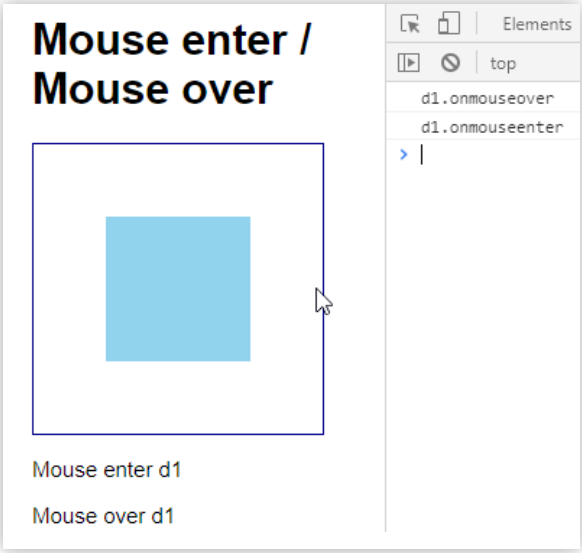


Рисунок 10

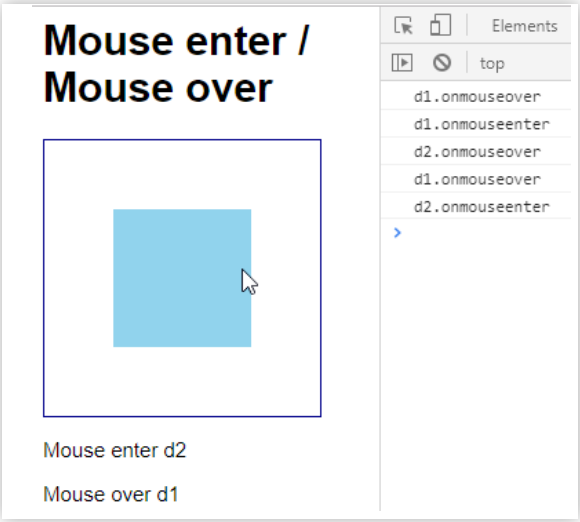


Рисунок 11

При пересечении границ внешнего блока оба события имеют одинаковый эффект. Обе надписи «Mouse enter» и «Mouse over» устанавливаются в значение «d1», в консоли также видны эти два сообщения.

Когда курсор мыши пересекает границу внутреннего блока, возникает три события, что следует из сообщений в консоли. Событие «onmouseover» приходит дважды — сначала от блока «d2», затем от «d1». В то же время событие «onmouseenter» приходит только один раз от верхнего блока «d2». Дублирование события «onmouseover» приводит к тому, что текст для сообщения «Mouse over» также устанавливается дважды. Последнее из них стирает предыдущее и в результате на экране остается результат «Mouse over d1», хотя курсор зашел на блок «d2». Об этих особенностях следует помнить при использовании событий разного типа.

О всплытии событий и управлении этим процессом детальнее будет рассказано в следующем уроке, после знакомства со структурой документа и компоновкой его элементов. Перейдем к рассмотрению событий клавиатуры.

Как правило, клавиатура используется для ввода определенных данных в специальные поля веб-страницы. Однако, для манипуляций с этим процессом предусмотрена отдельная группа событий, возникающих именно в процессе ввода данных. Такие события клавиатуры, как нажатие кнопки (keydown) или ее отпускание (keyup) обычно используют для задач, не связанных напрямую с вводом данных, например, для игровых или анимационных эффектов.

Выберем себе в качестве программы для реализации именно такую игровую задачу — «выстрел из пушки». Предположим, что у нас есть некий объект-пушка и, на некотором расстоянии от нее, мишень. При нажатии клавиши «пробел» из пушки должен вылетать снаряд, пролетать до мишени и исчезнуть после попадания в нее.

Остановимся более подробно на задаче обеспечения полета снаряда. Классически, анимационные эффекты во многих языках программирования реализуются при помощи периодического выполнения некоторой функции или периодической посылкой некоторого сообщения (таймера) и являются частью модели событий, которые мы рассматриваем в данном разделе. В языке JavaScript, согласно спецификации ECMAScript, таких событий не предусмотрено.

Описанные возможности предоставляет нам окружение языка JavaScript — браузер, в виде функций «setTimeout» и «setInterval». Акцент на разделение обязанностей языка и браузера сделан не случайно, ведь, казалось бы, какая

разница, на каком уровне реализованы функции, если их можно использовать в своих целях. Дело в том, что сам язык JavaScript не берет на себя ответственности за внутренний контроль времени из-за отсутствия таких механизмов. Браузер, в свою очередь, не имеет абсолютно полного влияния на процесс выполнения скриптов, то есть не может прервать одну инструкцию и перейти к другой по прошествии заданного временного интервала. В итоге, отсчет времени становится неточным и накапливает эти неточности шаг за шагом. Указанные механизмы нельзя использовать для задач, где требуется точный контроль времени.

Для нашей задачи большой точности не требуется, поэтому можно свободно применить одну из отмеченных браузерных функций. Рассмотрим их более детально:

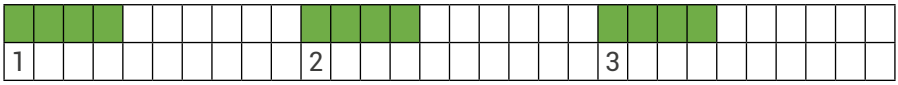
Функция	Описание
handle=setTimeout(func,dt)	Планирует запуск функции «func» через «dt» миллисекунд
clearTimeout(handle)	Отменяет запланированный запуск «handle»
handle=setInterval(func,dt)	Устанавливает периодический запуск «func» каждые «dt» миллисекунд
clearInterval(handle)	Отменяет периодический запуск «handle»

Функции делятся на те, которые планируют отложенный запуск функции, и те, которые эти планы отменяют. Планирующие функции «setTimeout» и «setInterval» возвращают идентификатор (дескриптор) своего «плана» — «handle». Возможно создать несколько различных планов отложенного запуска, отличающихся этим значением. При помощи дескриптора «handle» отменяющие

функции определяют один конкретный план, который следует прекратить.

Функция «setTimeout» предполагает однократный отложенный запуск указанной функции, тогда как «setInterval» планирует бесконечную цепочку перезапусков, пока она не будет прервана отменяющей функцией «clearInterval» или не закроется вкладка браузера с данной страницей. Обе функции требуют указать интервал времени до следующего запуска «dt» в миллисекундах. Напомним, что 1000 мс — это 1 секунда. То есть 500 мс — это 0,5 секунды, что соответствует 2 запускам за 1 секунду. 100 мс будет соответствовать 10 запускам за 1 секунду и т.д.

Хотя функция «setInterval» кажется более привлекательной для создания таймера — периодического запуска определенной функции, она имеет ряд недостатков. Во-первых, она устанавливает время между началами запуска функции. Например, если функция сама-по-себе выполняется 0,4 секунды, и мы указали интервал запуска 1 секунду, то пауза между остановкой предыдущей и запуском новой функции будет 0,6 секунд



Во-вторых, функция «setInterval» накапливает ошибку. Если по каким-то причинам происходят задержки таймера, то в дальнейшем они не корректируются и могут привести к отклонениям ожидаемого и реального процессов. Часы, созданные при помощи функции «setInterval», будут отставать от действительного времени, причем, чем дольше будет работать программа, тем сильнее будет отставание.

В-третьих, браузер может сам приостановить или замедлить работу таймера, когда страница неактивна или компьютер переходит на питание от батарей. Также по-разному таймер может искажать свой ход при появлении диалоговых окон.

Как итог, функция «setInterval» крайне ненадежна. Вместо нее рекомендуется использовать периодический перезапуск функции однократного отложенного запуска «setTimeout». Можно добавить, что подобный прием является основным способом создания периодических процессов в некоторых языках, например, в Python.

Определившись с выбором функции управления перезапуском, перейдем к созданию интерфейса. Его вид приведен на следующем рисунке 12.

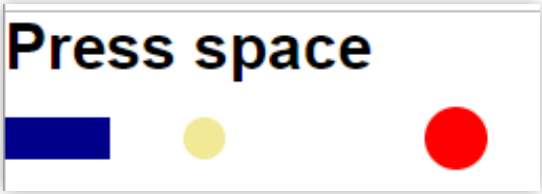


Рисунок 12

Основу страницы будут составлять три блока: прямоугольник, символизирующий ствол пушки, и два круга — для летящего снаряда и мишени.

Как мы уже исследовали в предыдущих примерах, управлять размещением элементов программным способом удобнее, применяя абсолютное позиционирование. Поэтому сразу укажем для всех элементов соответствующие стилевые атрибуты. Разметочная часть нашего документа примет следующий вид

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Keyboard events</title>
    <style>
        body {
            margin: 0;
            position: relative;
        }
        #gun, #bullet, #aim {
            position: absolute;
        }
        #bullet, #aim {
            border-radius: 50%;
        }
        #gun {
            background: navy;
            height: 20px;
            left: 0;
            top:  50px;
            width:  50px;
        }
        #bullet {
            background: khaki;
            height: 20px;
            left: 30px;
            top:  50px;
            width: 20px;
            z-index: -1;
        }
        #aim {
            background: red;
            height: 30px;
            left: 200px;
            top:  45px;
            width: 30px;
        }
    </style>
```

```
</head>
<body>
    <h1>Press space</h1>
    <div id="gun"></div>
    <div id="bullet"></div>
    <div id="aim"></div>
</body>
</html>
```

Блок, отвечающий за снаряд «bullet», изначально размещен за блоком-пушкой. Его координаты «left» и «top» подобраны таким образом, чтобы он располагался возле правого конца пушки. Атрибут «z-index: -1» перемещает блок «bullet» на задний план, из-за чего он будет скрываться как за пушкой, так и, в последствие, за мишенью. Для того чтобы убедиться в правильности компоновки можете указать для блока «bullet» значение «z-index: 1», в таком случае он отобразится поверх других элементов и станет видимым для анализа размещения.

Перейдем к реализации скриптовой части. Во-первых, создадим переменные и функцию, отвечающие за движение снаряда.

```
<script>
    var bulletX = 30;
    var onFly = false;
    function moveBullet() {
        if(onFly){
            bulletX += 5;
            if(bulletX >= 200){
                bulletX = 30;
                onFly = false;
            }
```

```
            else {
                setTimeout(moveBullet, 50);
            }
            bullet.style.left = bulletX + "px";
        }
    }
</script>
```

Поскольку движение снаряда будет происходить только по горизонтали, нам нужна одна переменная, отвечающая за координату «x» снаряда — «bulletX». Ее начальное значение (30) соответствует стилевому атрибуту «left» блока «bullet». Также, для обеспечения перезапуска функции нам потребуется переменная «onFly», изначально установленная в «false».

Функция «moveBullet» отвечает за перемещение снаряда. Все тело функции окружено условным оператором «if(onFly)», то есть функция будет выполняться только в режиме полета снаряда. Координата «bulletX» увеличивается на несколько единиц. В нашем случае это «5», в дальнейшем можно поменять эту величину для более быстрого или медленного полета.

Далее выполняется проверка на то, что снаряд достиг мишени «if(bulletX >= 200)». Величина «200» также взята из стилевых определений атрибута «left», только у блока «aim». В том случае, если снаряд достиг мишени, его координата устанавливается в исходное состояние «bulletX = 30» (за пушкой) и режим полета прекращается изменением значения переменной «onFly».

Если же снаряд не достиг мишени, планируется отложенный запуск этой же функции «moveBullet» через 50 мс

(то есть 20 раз в секунду). При перезапуске повторятся описанные действия.

В любом случае после изменения координаты «bulletX» ее значение переносится в стилевое определение блока-снаряда «bullet.style.left = bulletX + "px"». Не забываем, что для этого атрибута необходимо указывать единицы измерения.

Обратите внимание, что инструкция «setTimeout» не остановит работу текущего запуска функции, а лишь запланирует новый отложенный запуск. Поэтому блок «else» завершится и выполнится следующая за ним инструкция по изменению стиля, отвечающего за положение элемента.

Осталось добавить обработчик события нажатия клавиатуры, который будет запускать полет снаряда, то есть функцию «moveBullet». Добавим в определение тела документа следующий атрибут

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

А в скриптовую часть — код для функции-обработчика

```
function keyHandler(e){
    if(e.code == "Space" && !onFly) {
        onFly = true;
        moveBullet();
    }
}
```

Как уже было отмечено, функция проверяет тот факт, что нажатая клавиша соответствует пробелу. Для этого анализируется свойство «code» объекта-события. Согласно стандарта, это свойство имеет символьное описание нажатой кнопки, то есть для пробела имеет значение «Space».

Дополнительно проверяется условие, что в данный момент снаряд не находится в состоянии полета.

При выполнении описанных условий устанавливается режим полета «onFly = true» и инициируется первый запуск функции «moveBullet». В дальнейшем, она сама себя будет перезапускать, пока снаряд не достигнет мишени.

Создайте новый HTML-файл. Наберите или скопируйте в него приведенное выше содержание (*полный код программы также доступен в папке Sources, файл JS_3_10.html*). Убедитесь, что внешний вид страницы соответствует приведенному выше рисунку, а при нажатии кнопки «пробел» происходит «выстрел» — пролет снаряда от блока-пушки до блока-мишени.

Расширим условие задачи: добавим в нашу программу возможность отмены выстрела при нажатии клавиши «Esc» (*Escape*).

Для начала, попробуем узнать детали события, происходящего при нажатии клавиши «Escape». Добавим в обработчик события «keyHandler» команду логгирования параметра «e» в консоль

```
function keyHandler(e){
console.log(e)
    if(e.code == "Space" && !onFly) {
      onFly = true;
      moveBullet();
    }
}
```

Сохраните изменения, обновите вкладку браузера, откройте консоль разработчика. После этого переведите фокус ввода из консоли на страницу — щелкните мышью

в произвольном месте вкладки (иначе нажатие кнопок будет приниматься консолью, а не вкладкой).

Нажмите пробел, убедитесь, что происходит «выстрел», а в консоли появляется сообщение о нажатии клавиши «пробел» (рис. 13).

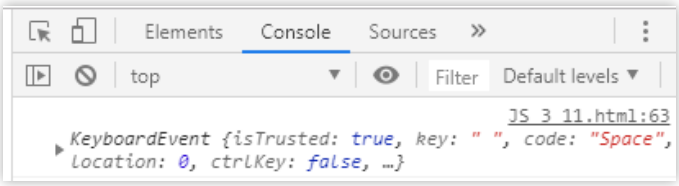


Рисунок 13

Нажмите клавишу «Escape». Обратите внимание, что сообщение в консоли не появляется. Это происходит вследствие особенности события «keypress». Оно не формируется для некоторых клавиш, которые имеют служебное назначение. Например, стрелки управления курсором или функциональные клавиши «F1», «F2» и т.д. Можете исследовать на базе созданной нами программы какие клавиши создают событие «keypress», а какие нет.

Для того чтобы иметь возможность обработать события от нажатия служебных клавиш необходимо использовать событие «keydown». Это системное событие, сопровождающее любые действия клавиатуры. Замените обработку события «keypress» на «keydown». Тег, формирующий тело документа, должен при этом принять следующий вид

```
<body onkeydown ="keyHandler(event)" >
```

Сохраните изменения и обновите вкладку браузера. Убедитесь, что фокус ввода находится на вкладке и на-

жмите клавишу «Escape». Убедитесь, что теперь в консоли появляется соответствующее сообщение (рис. 14).

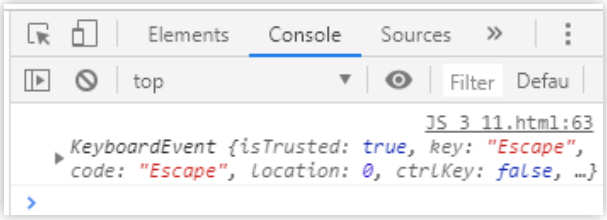


Рисунок 14

Также можете убедиться, что события приходят и от тех клавиш, от которых не приходили события «keypress». Обратите внимание, что среди параметров события также есть «code», который мы использовали в обработчике предыдущего события. Значит, тело обработчика должно функционировать и для нового события. Нажмите клавишу «пробел» и убедитесь в этом.

Главной информацией для нас является кодовое название клавиши «code: "Escape"». По этому значению мы можем задать действие для нажатия клавиши. Добавим в обработчик события следующий код

```
if(e.code == "Escape" && onFly) {
    onFly = false;
    bulletX = 30;
    bullet.style.left = bulletX + "px";
}
```

В условном операторе производится проверка на то, что событие пришло именно от клавиши «Escape», а также снаряд находится в состоянии полета «onFly». В таком

случае переменной «onFly» присваивается значение «false». Это прекратит работу функции, обеспечивающей полет «moveBullet», поскольку в ней первым делом проверяется указанная переменная.

Затем, возвращаем снаряд в исходное состояние: указываем значение для координаты снаряда «bulletX = 30» и заносим это значение в стилевой атрибут «bullet.style.left», отвечающий за позицию блока-снаряда. В итоге, он должен «спрятаться» за блоком-пушкой, создав эффект исчезновения с экрана.

Сохраните изменения (*код с изменениями доступен в папке «Sources», файл «JS_3_12.html»*). Обновите вкладку браузера. Нажмите пробел, убедитесь, что происходит «выстрел». Во время полета снаряда нажмите клавишу «Escape» — снаряд должен прекратить полет и визуально исчезнуть. Проверьте, что можно совершить повторные выстрелы и сохраняется возможность их отмены.

В процессе работы над программой мы столкнулись с двумя событиями, отвечающими за нажатие кнопок клавиатуры «keypress» и «keydown». Рассмотрим более детально различия между ними. Событие «keydown» посылается каждой кнопкой клавиатуры, что дает более широкие возможности их обработки в скриптах. Однако, в качестве символа нажатой кнопки «key» обычно посылается только базовый идентификатор кнопки — латинский символ в верхнем регистре. То есть нажатие кнопки «s» или «shift-s» приведет к одному и тому же значению «key» — «S». Более того, это значение будет сохраняться, если пользователь поменяет раскладку клавиатуры или переключится на другой язык ввода.

Событие «keypress» учитывает состояние регистра (нажатая кнопка «Shift» или включенный режим «Caps Lock»), раскладку клавиатуры и язык ввода. То есть значение «key» отвечает не за кнопку, которая была нажата, а за символ, который ей соответствует. Это является более удобным при анализе текстового ввода. Также, событие «keypress» не посылается служебными кнопками, что также не нужно для работы с текстовой информацией.

Как итог — событие «keypress» более удобно для работы с текстовым вводом, тогда как событие «keydown» — для других задач, в которых может быть задействована клавиатура.

В качестве замечания можно отметить тот факт, что с развитием браузеров и операционных систем значение для поля «key» в событии «keydown» также может учитывать раскладку и язык ввода. Для создания универсальных приложений лучше пользоваться стандартизированным полем «code», хранящим обозначение клавиши, одинаковое для всех браузеров.

События жизненного цикла

Когда Вы включаете браузер и открываете в нем новую веб-страницу происходит определенная цепочка процессов: браузер находит сервер, на котором находится страница, и запрашивает ее HTML код. Получив ответ, браузер, обрабатывает его и формирует структуру страницы. Обычно, в HTML коде находятся ссылки на дополнительные ресурсы — стилевые файлы, файлы скриптов, рисунков, фреймы и т.п. Обработав HTML код и собрав данные о всех ссылках, браузер формирует за-

просы на эти ресурсы, получая каждый из них отдельно, и подключает их к ранее загруженной странице. Когда все ресурсы будут получены, веб-страница будет считаться загруженной окончательно.

После загрузки страницы начинается процесс взаимодействия с пользователем. При этом на странице могут появляться, исчезать или изменяться ее составные элементы. Некоторые из них могут потребовать новых обращений к серверу, например, добавление новых изображений.

Закончив работу с веб-страницей пользователь закрывает вкладку, и браузер приступает к освобождению памяти, занятой этой страницей и ее ресурсами. Также удаляются или перемещаются в специальное хранилище (кеш) загруженные с сервера файлы стилей, скриптов и изображений.

Описанный процесс носит название жизненного цикла веб-страницы. Как сама страница, так и отдельные ее элементы, требующие отдельной загрузки, получают события, связанные с началом или окончанием того или иного этапа жизненного цикла. События данной группы позволяют встраивать определенную функциональность в нужном месте цикла. Например, нецелесообразно вызывать функции, описанные в отдельном файле, до тех пор, пока файл еще полностью не загружен.

Отметим также тот факт, что в процессе загрузки любого из дополнительных ресурсов возможно возникновение ошибки. Есть вероятность, что ресурс был перемещен или удален, обновлен, из-за чего поменял имя, или просто возникли сбои с сетевым подключением. В этих случаях события жизненного цикла могут предоставить

нам сведения о проблемах с подключением ресурсов. Если страница может работать и без дополнительных функций, то можно принять соответствующее решение. Например, если за счет неудачной загрузки скриптов не будет работать «карусель», меняющая один за другим слайды, то это можно считать некритической ошибкой и продолжать работу страницы. Если же ресурсы обеспечивают основную функциональность и без них работа невозможна, то события жизненного цикла позволят нам отреагировать на такие ситуации.

Жизненный цикл веб-страницы сопровождается следующими событиями

- DOMContentLoaded — браузер полностью загрузил HTML, файлы стилей и скриптов, построил структуру документа (DOM-структуру, детальнее о DOM будет рассказано в следующем уроке);
- load — браузер загрузил все дополнительные ресурсы — изображения, фреймы;
- beforeunload — браузер получил команду закрыть страницу (вкладку);
- unload — браузер закрыл страницу (вкладку).

Событие «DOMContentLoaded» является одним из наиболее популярных среди событий жизненного цикла. Оно посылается объекту «document» тогда, когда загружен код HTML, стилевые файлы и скрипты. На момент посылки этого сообщения могут быть еще не загружены изображения или фреймы. Тем не менее, структура страницы уже определена, то есть в обработчике события все элементы страницы доступны и к

ним можно обращаться по их атрибутам («id», «class», «name» и т.п.).

Обработчик события «DOMContentLoaded» подключается *только* при помощи команды «addEventListener»:

```
document.addEventListener( "DOMContentLoaded" ,
    function() {
        alert("DOM loaded")
    }
)
```

Использование атрибутов с префиксом «on», как для большинства других событий, не даст должного эффекта. Ошибки не возникнет, т.к. мы имеем право создать любое дополнительное поле в произвольном объекте, однако, такой обработчик не будет вызван при загрузке документа

```
document.onDOMContentLoaded = function() {
    alert("DOM loaded")
}
```

Событие «load» посылается после загрузки всех дополнительных ресурсов — изображений, фреймов и т.п. Данное событие относится к объекту «window» и может быть обработано всеми допустимыми вариантами — как при помощи метода «addEventListener», так и указанием атрибутов с префиксом «on»

```
<body onload="alert('body loaded')">
window.onload = function() {alert('body loaded')}
window.addEventListener( "load",
       function() {alert('body loaded')})
```

Следует обратить внимание на то, что загрузка изображений еще не означает их отображение. Событие «load» посылается по факту окончания загрузки, но не по окончанию прорисовки. В большинстве случаев изображения хранятся в сжатых форматах, и для их вывода на страницу браузеру необходимо повести «декомпрессию» — восстановление графической информации из сжатого файла. На это уходит определенное время, которого может не хватить до вызова обработчика события.

Рассмотрим процессы прихода событий и состояния страницы в эти моменты на следующем примере. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_13.html*). Скопируйте также из папки «*Sources*» в свою рабочую папку файл-изображение «*step.jpg*»

```
<!doctype html />
<html>
<head>
    <style>
        img {
            height: auto;
            width:  200px;
        }
    </style>
</head>

<body onload="alert('body loaded')">
    <img src="step.jpg" />
    <script>
       document.addEventListener("DOMContentLoaded" ,
           function() {
               alert("DOM loaded")
           }
```

```
        )
    </script>
</body>

</html>
```

На данной странице размещается единственный элемент — изображение (рис. 15).

Рисунок 15

Также в коде установлены два обработчика событий жизненного цикла: «DOMContentLoaded» и «load». Первый — при помощи метода «addEventListener», второй — через атрибут «onload» тела документа.

Сохраните файл, откройте его в браузере. В процессе загрузки появится два сообщения, после нажатия на кнопку «OK» диалогового окна страница будет принимать следующий вид (рис. 16, 17)

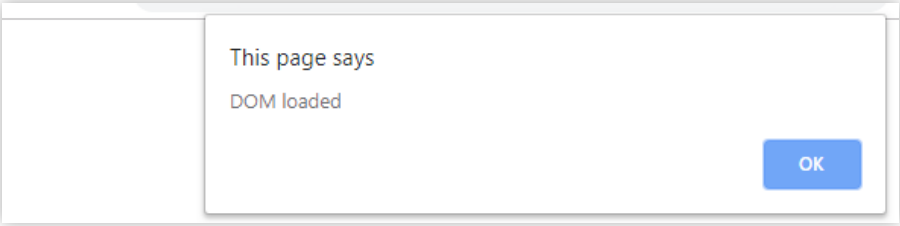


Рисунок 16

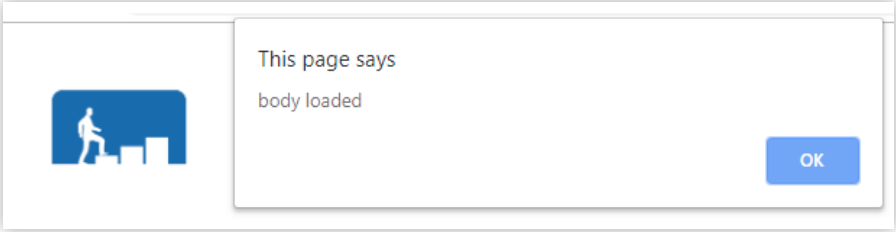


Рисунок 17

Событие «DOMContentLoaded» приходит сразу после загрузки HTML кода страницы и построения ее составных элементов. Само содержимое элементов может еще быть не загруженным. Поэтому, в нашем случае, в момент прихода события и вызова сообщения на вкладке браузера ничего не отображается (наш единственный элемент требует загрузки дополнительного ресурса — файла «*step.jpg*»). Появление диалогового окна (alert) останавливает выполнение скриптов и загрузку страницы, благодаря чему мы успеваем увидеть последовательность процессов загрузки и посылки сообщений.

Нажмите кнопку «OK». Появится следующее сообщение «body loaded». Это соответствует моменту окончания загрузки всех ресурсов, в данном случае — изображения. Однако на вкладке мы видим лишь его часть — тот фрагмент, который браузер успел восстановить из полученного файла и отобразить на странице. Вызов диалогового окна снова остановил работу и зафиксировал момент частичной прорисовки изображения.

Нажмите «OK» во втором диалоговом окне, и изображение должно появиться полностью. Следует отметить, что в зависимости от быстродействия компьютера, планшета, смартфона или другого устройства, на котором

открывается страница, промежуточный фрагмент изображения может быть разного размера. Возможно, при малом быстродействии, фрагмент не появится вообще, т.к. браузер не успеет обработать достаточно данных даже для частичного отображения. При высоком быстродействии или при малом размере изображения оно может успеть обработаться полностью. В таком случае на момент события «load» на странице будет видно все содержимое картинки.

Для того чтобы исследовать процессы загрузки страницы и ее ресурсов в инструментах разработчика браузера есть специальная вкладка «Network». Откройте консоль уже известным Вам способом и переключитесь на указанную вкладку сетевого монитора. Для сбора информации необходимо перезагрузить страницу с открытой инструментальной вкладкой «Network», поэтому обновите страницу, нажав клавишу «F5» клавиатуры либо кнопку обновления в браузере. Инструмент начнет сбор данных.

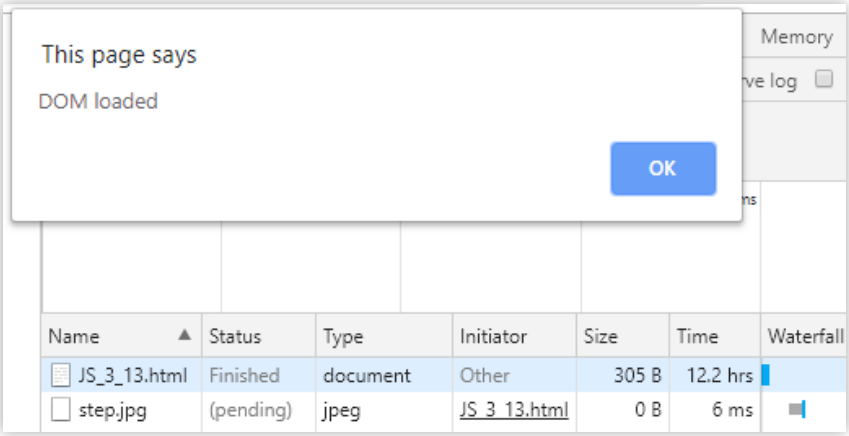


Рисунок 18

В момент прихода первого сообщения мы увидим следующие параметры: основной документ «*JS_3_13.html*» уже загружен, о чем свидетельствует статус «Finished», в то же время ресурс «*step.jpg*» находится в состоянии приема «(pending)». В колонке «Waterfall» отображается диаграмма последовательности процессов. Как видно, ресурс начал загружаться, но загрузка еще не закончена, о чем свидетельствует синий бегунок в конце диаграммы. Вполне возможно, что при других параметрах быстродействия, загрузка может еще не начаться либо, наоборот, будет загружена большая часть ресурса.

Нажмите кнопку «OK». Состояние сетевого монитора сменится на следующее (рис. 19).

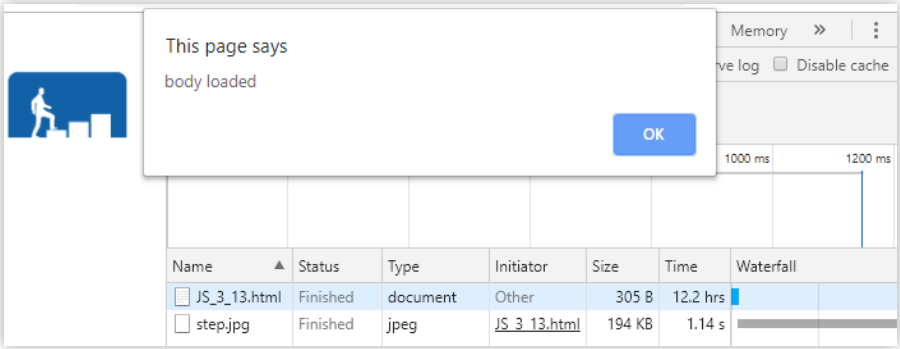


Рисунок 19

Как видно из статуса процессов, загрузка всех ресурсов завершена. Однако, диаграмма «Waterfall» еще не определила момент конца загрузки, т.к. не пришел отклик от браузера об обработки полученных данных. Из-за этого диаграмма загрузки «*step.jpg*» не имеет конца. Рисунок на странице, как мы уже видели ранее, отображается частично.

Закройте диалоговое окно, нажав «ОК». Сетевой монитор получит сведения от браузера об окончании обработки ресурса и сможет определить время, потраченное на загрузку и обработку изображения (рис. 20).

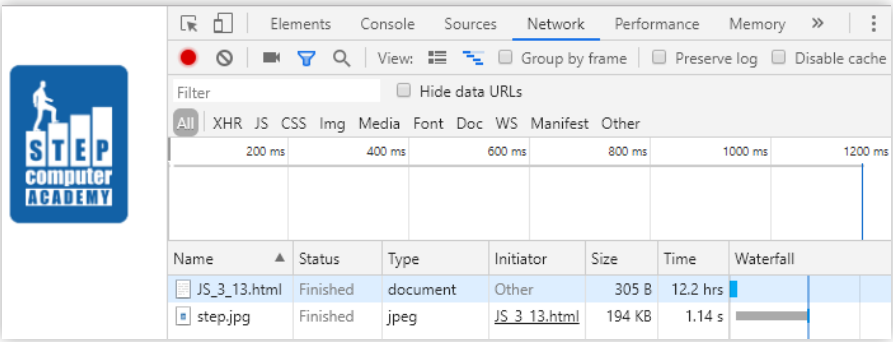


Рисунок 20

Диаграмма загрузки ресурса отобразится как законченный блок, линия в конце него означает, что завершены все загрузки, необходимые для страницы. Как следствие, рисунок на странице отображается полностью.

Поскольку дополнительные ресурсы страницы загружаются отдельно от HTML кода, для них также посылаются события об итоговом статусе. Такими событиями являются «load» и «error». Событие «load» посылается элементу, когда его загрузка успешно завершается. В случае неудачной загрузки элемент получает событие «error». Эти события дают возможность убедиться в том, что ресурс полностью загрузился (или возникла ошибка загрузки).

Нам уже известно, что изображения загружаются в самостоятельных потоках. Значит, для них будут посылаться события «load» или «error». Рассмотрим их обработку на следующем примере, определяющем успеш-

ность загрузки ресурсов-изображений. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_14.html*). Для правильной работы примера ему нужен файл «*step.jpg*». Убедитесь, что этот файл доступен из предыдущего примера или скопируйте его из папки «Sources» в свою рабочую папку.

```
<!doctype html />
<html>
<head>
    <style>
        img {
            height: auto;
            width:  100px;
        }
    </style>
</head>

<body>
    <img src="step.jpg"
       onload="addMessage('step.jpg load OK')"
       onerror="addMessage('step.jpg load error')"/>
    <img src="step2.jpg"
       onload="addMessage('step2.jpg load OK')"
       onerror="addMessage('step2.jpg load error')"/>
    <p id="txt"></p>

    <script>
        function addMessage(msg) {
            window.txt.innerHTML += msg + "<br/>"
        }
    </script>
</body>
</html>
```

В теле документа расположены два изображения «<img src="step.jpg"» и «<img src="step2.jpg"». Для одного из них файл-ресурс существует, для другого — нет. Поэтому первое изображение должно загрузиться с успешным статусом, тогда как второе — с ошибкой.

В каждом из изображений установлены по два обработчика событий «load» и «error». Обработчики вызывают дополнительную функцию «addMessage», которая описана в скриптовой части и «выводит» сообщение, добавляя его к внутреннему содержимому абзаца «<p id="txt">» через его атрибут «*txt.innerHTML*».

Сохраните файл и откройте его при помощи браузера. Изображение, для которого файл-ресурс присутствует в рабочей папке, должно отобразиться нормально. Если с ним возникает ошибка, то убедитесь, что файл скопирован в ту же папку, в которой находится HTML файл с текущим примером.

Второе изображение, которое ссылается на несуществующий файл, загружается с ошибкой и, соответствующим образом отображается на странице. Итоговый вид страницы представлен на следующем рисунке 21:

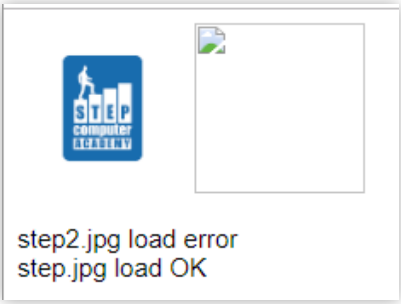


Рисунок 21

Обратите внимание, что сообщение об ошибке загрузки второго изображения появляется раньше, чем об успешной загрузке первого. Это связано с тем, что загрузка первого изображение занимает какое-то время, тогда как второе изображение сразу вызывает ошибку. Поэтому эта ошибка приходит раньше, чем сообщение об успехе, и надпись появляется первой, хотя изображение обрабатывается позже.

Задание для самостоятельной работы: добавьте на страницу еще несколько изображений с контролем загрузки. В конце блока сообщений добавьте итоги, например: «успешно загружено — 3», «ошибки загрузки — 1».

События «beforeunload» и «unload» посылаются странице на последних этапах ее жизненного цикла — при закрытии вкладки браузера, содержащую данную страницу, либо при переходе по ссылке на другой сайт (в этой же вкладке). Сам процесс закрытия страницы состоит из нескольких этапов, проходящих в обратной последовательности по сравнению с загрузкой. Сначала происходит освобождение памяти, занятой ресурсами страницы. В нашем примере — это изображение, восстановленное из сжатого «JPG» файла. Затем удаляются из памяти структурные элементы страницы. На последнем этапе закрывается окно вкладки или весь браузер, либо начинается процесс загрузки новой страницы — в зависимости от действий пользователя.

Событие «beforeunload» посылается окну (объекту «window») и сигнализирует о начале закрытия (выгрузки, unload) страницы. На данном этапе можно предупредить пользователя, что на странице остались несохраненные

данные, незаконченные процессы отправки или получения данных. При этом пользователь может отменить закрытие страницы и вернуться к работе с ней. Использование данного события очень популярно для страниц, на которых используется пользовательский ввод — текстовые редакторы, онлайн компиляторы различных языков программирования и т.п.

Событие «unload» приходит в самом конце жизненного цикла. Отменить закрытие страницы на данном этапе уже невозможно, поэтому данное событие используется крайне редко. В обработчике данного события обычно закрывают связанные с данной страницей дополнительные окна (если они создавались).

Обработка события «beforeunload» имеет несколько особенностей. Рассмотрим их на практическом примере. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_15.html*).

```
<!doctype html />
<html>
<head>
    <style>
        #text {
            height: 200px;
            width:  400px;
        }
    </style>
</head>

<body>
    <textarea id="text"></textarea><br/>
    <a href="http://itstep.org">IT Step</a>
```

```
    <script>
        window.onbeforeunload =
            function(e) {
                if(text.value.length > 0) {
                var msg = 'Text not saved';
                e.returnValue = msg;
                return msg;
            }
            return null;
        }
    </script>
</body>

</html>
```

В теле документа расположен текстовый блок (<textarea id="text">), для которого в заголовочной части определены стили, устанавливающие размеры блока. Ниже блока расположена ссылка на внешний ресурс (<a href="http://itstep.org">), нажатие на которую приведет к выгрузке текущей страницы и загрузке новой.

В скриптовой части документа устанавливается обработчик события «window.onbeforeunload». Принцип работы обработчика заключается в том, что он должен вернуть непустое сообщение, если необходимо уведомить пользователя о несохраненных данных. В приведенном примере в качестве сообщения использовано «msg = 'Text not saved'». Для совместимости с разными типами браузеров необходимо:

а) установить свойство «e.returnValue=msg» для объекта-события «e»;

б) вернуть сообщение в качестве результата работы функции-обработчика «return msg».

В случае, если сообщение пользователю выводить не нужно, обработчик возвращает значение «null». Критерием проверки для выдачи сообщения является наличие пользовательского ввода в текстовом блоке. Это проверяется путем определения длины текста, содержащегося в текстовом блоке: «if(text.value.length > 0)».

Обратите внимание, что мы не создаем никаких диалоговых окон на подобие «alert» или «confirm». Мы лишь обеспечиваем возврат ненулевого значения (сообщения) из обработчика события.

Сохраните файл и откройте его в браузере. Не вводя никакого текста, нажмите на ссылку под текстовым блоком. Должен произойти переход на другую страницу.

Вернитесь на предыдущую страницу (нажмите комбинацию Alt-«стрелка влево» либо кнопку «назад» в браузере). Введите в текстовый блок произвольные данные и снова нажмите на ссылку. В результате должно появиться сообщение. Вид этого сообщения сильно зависит от используемого браузера (рис. 22-25):

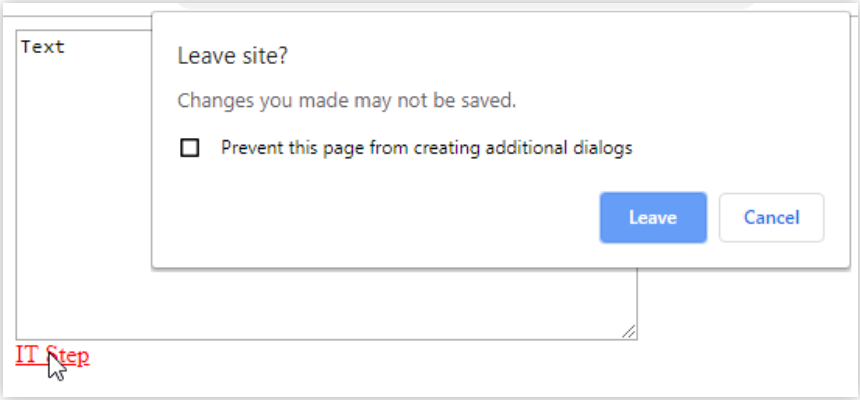


Рисунок 22. Google Chrome

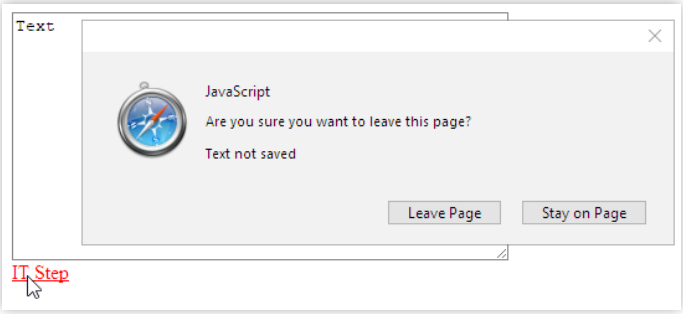


Рисунок 23. Apple Safari

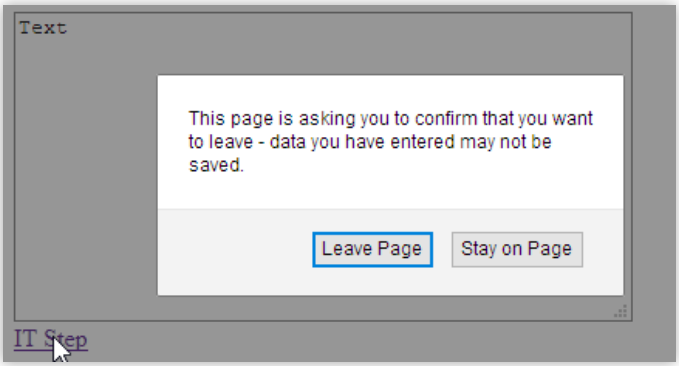


Рисунок 24. Mozilla Firefox

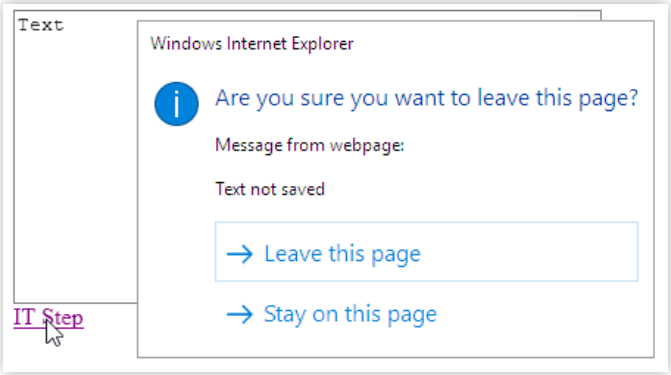


Рисунок 25. Internet Explorer

Кроме того, что сообщения отличаются внешним видом, характерным для каждого вида браузеров, обращает на себя внимание тот факт, что текст нашего сообщения ('Text not saved') отображается не во всех случаях. Браузеры «Google Chrome» и «Mozilla Firefox» полностью игнорируют содержание данного сообщения. Браузеры «Apple Safari» и «Internet Explorer» добавляют его к своим собственным сообщениям. Если у Вас установлен другой браузер, то и сообщение может отличаться от приведенных выше.

Почему браузеры игнорируют сообщения, заложенные в скриптовой части? Это делается с целью безопасности. Исторически, браузеры предоставляли программисту полный контроль над процессом выгрузки (закрытия) страницы. Недобросовестные разработчики сайтов с целью удержания посетителей на своих страницах использовали разнообразные негативные приемы. Например, «переставленные кнопки» — при нажатии на кнопку «покинуть страницу» на самом деле генерировалась команда «остаться на странице». Также использовались сообщения пугающего содержания, на подобие «При нажатии на кнопку с Вашего счета будет списано $10».

Для упреждения подобных уловок, было принято решение о полном или частичном игнорировании данных, передаваемых в сообщение. Браузеры в любом случае не дают возможность управлять внешним видом диалогового окна — менять его текст или порядок кнопок. А дополнительный текст сообщения, если и показывается, то так, чтобы было полностью понятно, что это лишь дополнительный текст, не искажающий смысл данного

диалогового окна. Как мы уже убедились, в ряде случаев этот текст вообще будет проигнорирован.

Также с целью безопасности в обработчике события «onbeforeunload» не допускается создание дополнительных диалоговых окон. Если к коду обработчика добавить строку «alert(3)», то в консоли разработчика при приходе события появится предупреждение:

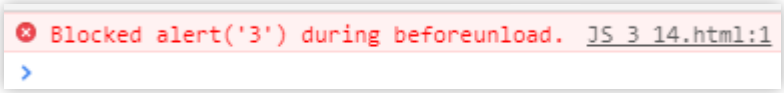


Рисунок 26

Можете убедиться в этом на практике, добавив указанную инструкцию в тело обработчика.

Задание для самостоятельной работы: добавьте на страницу еще два поля ввода текста. Дополните обработчик события «onbeforeunload» проверкой на наличие текста хотя бы в одном из полей ввода — если такой текст есть, то выдается предупреждение о несохраненных данных.

Обработчики событий по умолчанию (стандартные обработчики), запрет вызова стандартного обработчика

Для некоторых событий в браузере предусмотрены специальные стандартные обработчики, которые запускаются автоматически — без необходимости добавления их программным путем (обработчики событий по умолчанию). Обычно, это широко распространенные для многих приложений и операционных систем действия: вызов контекстного меню при нажатии правой клавиши

мыши, масштабирование контента при прокрутке колеса мыши с нажатой кнопкой «Ctrl» на клавиатуре, совершение определенных действий при нажатии «горячих» комбинаций, например, сохранение страницы при нажатии «Ctrl-S», копирование и вставка выделенного текста «Ctrl-C» — «Ctrl-V», и многие другие.

В некоторых случаях обработчики событий по умолчанию могут мешать логике работы основного содержимого страницы. Такие ситуации возникают, если в интерфейсе страницы используются средства управления, совпадающие с «зарезервированными» возможностями. Например, когда необходимо задействовать для своих целей правую клавишу мыши, ее колесо или сочетания некоторых удобных для нажатия кнопок клавиатуры, которые традиционно вызывают действия по умолчанию.

Рассмотрим в качестве примера следующую задачу: создать блок, который будет менять цвет при нажатии на нем клавиши мыши: зеленый, если нажата левая клавиша; синий, если средняя (или колесо), красный — если правая.

При нажатии клавиши мыши объект получает событие «mousedown». Это событие формируется основными клавишами мыши — левой, средней (или нажатие на колесо) и правой. Если мышь оснащена дополнительными клавишами (кроме перечисленных), то их нажатие сопровождается событием «auxclick» (см таблицу описания событий в разделе 3). Будем обрабатывать в нашем примере только события от основных клавиш мыши.

Поскольку все три клавиши мыши посылают одно и то же событие «mousedown», они устанавливают различные значения для свойства «which» объекта-события,

передаваемого в функцию-обработчик: 1 — для левой клавиши, 2 — для средней, 3 — для правой. Используем эти значения для определения того, от какой клавиши поступило событие, и применим к блоку необходимый цвет. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_16.html*).

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Defaults</title>
    <style>
        #d1 {
            border: 2px solid gray;
            height: 200px;
            width:  200px;
        }
    </style>
</head>
<body>
    <div id="d1"></div>
    <script>
        window.d1.onmousedown = function(e) {
            var bgColor;
            switch( e.which ) {
                case 1:
                    bgColor = "lime" ;
                    break ;
                case 2:
                    bgColor = "aqua" ;
                    break ;
                case 3:
                    bgColor = "tomato" ;
                    break ;
```

```
            }
            window.d1.style.backgroundColor = bgColor;
        }
    </script>
</body>
</html>
```

В теле документа создается блок , для которого в заголовочной части при помощи стилей задаются размеры 200×200 пикселей и рамка, обеспечивающая видимость его границ. В скриптовой части для блока устанавливается обработчик события «onmousedown». Основным действием функции-обработчика является установка фонового цвета блока «window.d1.style.backgroundColor = bgColor». Сам цвет определяется в теле оператора «switch» в зависимости от значения поля «e.which», отвечающего за номер нажатой клавиши мыши.

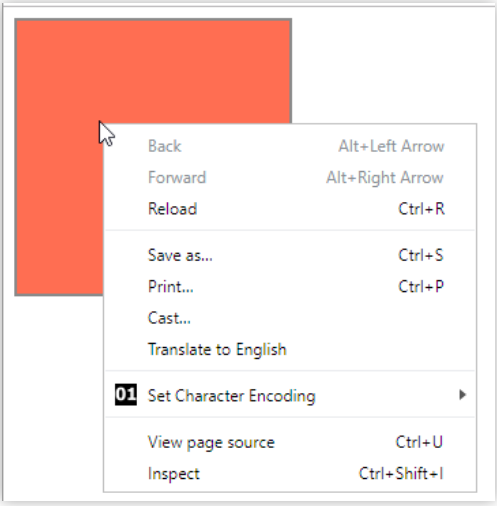


Рисунок 27

Сохраните файл и откройте его при помощи браузера. Нажмите на блок левой клавишей, средней (или нажмите на колесо), правой — убедитесь, что блок меняет свой цвет. Обратите внимание, что при нажатии правой клавиши кроме смены цвета на блоке появляется контекстное меню (рис. 27).

Вызов контекстного меню определяется браузером как действие по умолчанию, предназначенное для правой клавиши мыши. Мы в коде нигде не указывали данную функциональность. В нашем примере такое поведение является нежелательным, поэтому необходимо заменить стандартный обработчик события контекстного меню «oncontextmenu» для блока.

Добавьте в скриптовую часть документа следующее определение (*код с изменениями также доступен в папке Sources, файл JS_3_16a.html*)

```
window.d1.oncontextmenu = function() { return false }
```

Сохраните изменения и обновите страницу, открытую в браузере. Убедитесь, что теперь при щелчке на блоке правой клавишей мыши контекстное меню не появляется. Тем не менее, при щелчке за пределами блока меню все так же вызывается.

Задание для самостоятельной работы. Модифицируйте код таким образом, чтобы контекстное меню не вызывалось и за пределами блока.

Стандартные обработчики, вызываемые определенными комбинациями кнопок клавиатуры, переопределяются несколько иным способом. Рассмотрим это на следующем

примере: создадим программу, выводящую на экран информацию о нажатой кнопке или их комбинации.

Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл JS_3_17.html*).

```
<!doctype html />
<html>
<head>
    <meta charset="utf-8" />
    <title>Defaults</title>
</head>
<body>
    <h1>Click on page and press a key</h1>
    <h2 id="out"></h2>
    <script>
        document.body.onkeydown = function(e) {
            var txt = "" ;
            if(e.ctrlKey)  txt += "Ctrl – " ;
            if(e.altKey)   txt += "Alt – " ;
            if(e.shiftKey) txt += "Shift – " ;
            txt += e.key ;
            out.innerText = txt ;
        }
    </script>
</body>
</html>
```

На странице расположен заголовок — подсказка «Click on page and press a key», напоминающая о назначении программы. Далее следует блок <h2 id="out">, который будет использоваться для вывода сообщений о нажатых кнопках.

В скриптовой части документа устанавливается обработчик события «onkeydown». Мы с ним знакомились ранее в этом уроке, поэтому детально на его описании

останавливаться не будем. В обработчике проверяются статусы нажатия кнопок «Ctrl», «Alt», «Shift» и, если они истинны, к сообщению добавляются соответствующие надписи. Далее сообщение дополняется символом кнопки «e.key» и выводится в блок «out».

Сохраните файл и откройте его в браузере. Щелкните мышью по странице для того, чтобы быть уверенным в том, что фокус ввода клавиатуры принадлежит данной вкладке. Нажимайте кнопки клавиатуры, в том числе удерживая кнопки «Ctrl», «Alt», «Shift» в различных комбинациях. Убедитесь в соответствии нажатых кнопок и сообщений на странице.

Нажмите комбинацию, для которой установлено действие по умолчанию. Например, «Ctrl-e». Как результат, кроме сообщения о нажатых кнопках браузер запустит средства поиска (рис. 28):

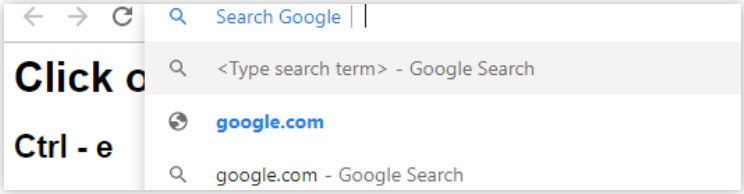


Рисунок 28

Попробуйте другие «горячие» комбинации: «Ctrl-s» будет открывать диалог сохранения страницы, «Ctrl-f» вызывать средства текстового поиска по вкладке, «Ctrl-u» — откроет окно с исходным кодом страницы, «Ctrl-r» — перезагрузит вкладку.

События от клавиатуры не предусматривают отдельных обработчиков действий по умолчанию, как это было

с контекстным меню. Все они также проходят через событие «keydown». Однако, добавлением инструкции «return false» в обработчик события предотвратить действие по умолчанию не удается. Можете убедиться в этом, дополнив функцию-обработчик указанной инструкцией.

Для того чтобы браузер не выполнял действия по умолчанию необходимо вызвать метод «preventDefault» у объекта-события «event» или, в нашем случае, у параметра функции-обработчика «e». Добавьте в любом месте описания функции инструкцию (*код с изменениями также доступен в папке Sources, файл JS_3_17a.html*)

```
e.preventDefault() ;
```

Сохраните файл и обновите страницу браузера. Убедитесь в том, что на комбинацию «Ctrl-e» браузер перестал реагировать и средства поиска не запускаются. Также теряют активность другие комбинации, описанные выше.

Следует отметить, что инструкция «preventDefault» действует только на те «горячие» комбинации, которые касаются вкладки браузера. Данная инструкция не отменяет действия, предусмотренные для браузера в целом, фоновых программ или для операционной системы. Например, комбинация «Ctrl-n» относится к браузеру (создает новое окно) и отдельной вкладкой не может быть отменена. Также коды вкладки не повлияют на системные комбинации, на подобие «Ctrl-Esc» или «Ctrl-Alt-Del». Вмешательство в работу таких комбинаций требует применение средств системного программирования и выходит за возможности JavaScript, ограниченного отдельным браузером.

Browser Object Model

Что такое Browser Object Model?

JavaScript является универсальным языком программирования, позволяющим создавать решения для различных задач, например, оконные приложения или серверные модули. Однако при использовании его для разработки веб-страниц появляются некоторые ограничения. Возможности программ, написанных на JavaScript внутри веб-страницы, не могут выйти за пределы этой страницы, ее HTML структуры. Поскольку коды JavaScript обрабатываются и выполняются браузером, они не могут прямо управлять самим браузером, лишь в той степени, в которой браузер это позволяет. В свою очередь браузер является отдельной программой в операционной системе и доступ к параметрам этой системы также лимитирован и для браузера, как программы, и для JavaScript, как ее части.

Основные ограничения, накладываемые на возможности JavaScript в HTML странице, можно сформулировать следующим образом:

- При помощи JavaScript нельзя запускать другие приложения, давать команды операционной системе. В противном случае веб-страницы могли бы перезагружать компьютер, отключать антивирусы и т.п.
- Инструкции JavaScript не имеют прямого доступа к файлам компьютера. Без разрешения пользователя невозможно даже открыть файл для чтения. Ина-

че разные сайты могли бы читать Ваши файлы и передавать их содержимое по сети.

- Из скриптов одной вкладки нельзя управлять содержимым других вкладок. Единственная возможность — закрыть вкладку, если она была создана из данного скрипта (эту возможность мы рассмотрим далее в уроке).
- Обращаться к серверу при помощи JavaScript можно только в том случае, если сайт был загружен именно с этого сервера. В конце урока мы более подробно поговорим о кросс-доменных ограничениях, влияющих на эту особенность JavaScript.
- Командами JavaScript нельзя закрыть или запустить браузер, свернуть его главное окно или изменить его размеры.

Взаимодействие с браузером и системой всё же необходимо для хорошо построенных сайтов. Например, могут возникнуть задачи вернуться на предыдущую просмотренную страницу, открыть новую страницу (или новый сайт) в текущей или новой вкладке браузера, а также получить данные о типе браузера или операционной системе, в которой он запущен. Для мобильных устройств могут понадобиться сведения о состоянии источника питания (аккумулятора) или данные о геолокации устройства.

Возможности, которые предоставляет браузер для использования кодами JavaScript, составляют основу Браузерной Объектной Модели (*Browser Object Model*, BOM). Согласно с этой моделью, работа с браузером и, через него, с операционной системой заменяется работой со специаль-

ным объектом «window». Из этого объекта можно получить (считать) информацию о параметрах и свойствах браузера, системы или устройства, а также установить или изменить (записать) некоторые величины, влияющие на их работу.

Следует отметить, что на данный момент для BOM не существует официального стандарта. Связано это с постоянной конкуренцией на рынке браузеров — так называемой «войной браузеров». Для завоевания лидирующих позиций браузеры предоставляют новые возможности, отличающие их от других конкурентов. Вероятно, что стандарт BOM вообще не будет принят в окончательном виде, так как появление такого стандарта, во-первых, сделает браузеры «одинаковыми» и они тут же начнут снова расширять свои возможности сверх стандарта. И мы снова будем говорить об отсутствии стандарта, но уже на другие группы функций. Во-вторых, сама идея стандарта касается не столько языка JavaScript, сколько его взаимодействия с браузером и системой. В этом аспекте нужно принять решение, что какой-то или какие-то браузеры являются «образцовыми», а остальные должны переделать свои функции под них. Такие предпосылки могут завершиться неудачей из-за правил честной конкуренции.

Таким образом, складывается ситуация в которой разные браузеры содержат собственные, уникальные возможности, не повторяющиеся в других браузерах. Более того, эти возможности постоянно обновляются и добавляются. Когда Вы будете читать этот урок перечень возможностей различных браузеров, наверняка, будет отличаться от тех, которые существуют сейчас, на момент написания урока. В то же время, большинство современных браузеров

реализуют определенный набор одинаковых методов, позволяющих скриптам на веб-страницах управлять одними и теми же свойствами. Его можно считать определенным неофициальным стандартом, позволяющим вести разговор о браузерной модели вообще, а не о функциях каждого отдельного браузера по отдельности. Исходя из сказанного, при использовании возможностей BOM в реальных проектах желательно проверить их поддержку и правильность работы в различных браузерах, причем не только в их новейших выпусках, а и в прошлых версиях, которые могут использовать поклонники данного семейства браузеров.

Ранее мы упоминали об еще одной роли объекта «window» — обеспечении глобальной области видимости для переменных (см. урок 1). Напомним, что именно благодаря этому объекту реализуется возможность обмена данными между различными объектами JavaScript.

Сам объект «window», в свою очередь, является составным. В его структуру входит несколько дочерних объектов, отвечающих за различные задачи (см. рис. 30)

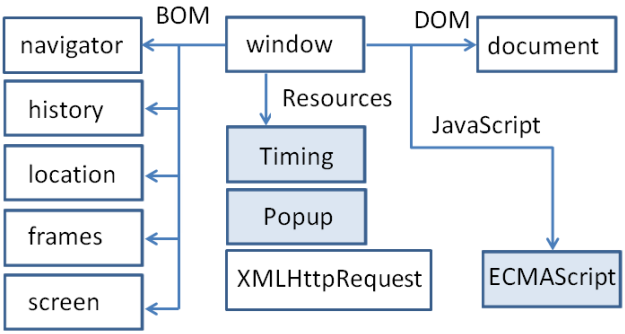


Рисунок 29

Среди дочерних элементов объекта «window» можно выделить несколько групп. Группу BOM составляют объекты, отвечающие за описанное выше взаимодействие с браузером и операционной системой. Чуть позже мы рассмотрим их более детально.

Группу ресурсов (Resources) формируют объекты и функции, обеспечивающие вспомогательные задачи, такие как:

- таймер и отложенный запуск функций (Timing);
- диалоговые окна (Popup), рассмотренные на первом уроке (alert, prompt, confirm);
- асинхронный обмен данными через объект «XMLHttpRequest».

Объект «document» является основой объектной модели документа (*Document Object Model*, DOM). В нем собирается информация о структуре HTML страницы. С его помощью можно управлять этой структурой посредством команд JavaScript. С этим объектом мы детальнее познакомимся во второй части урока.

Стандартизированные элементы языка JavaScript можно выделить в группу «ECMAScript». Здесь мы увидим, например, типы данных, такие как «Object» или «String», основные функции и определения языка. Эта группа обеспечивает работу JavaScript в соответствии с требованиями стандарта языка — ECMAScript.

Элементы, указанные на рисунке 31 как Timing, Popup и ECMAScript не являются непосредственными дочерними узлами объекта «window», а обозначают группы дочерних функций, свойств и объектов, объединенных

по своему предназначению. К остальным элементам можно получить доступ, набрав их имя после имени объекта «window.» (добавив после него точку), либо непосредственно указав их имя, опустив запись «window.». Убедитесь в этом, включив консоль браузера:

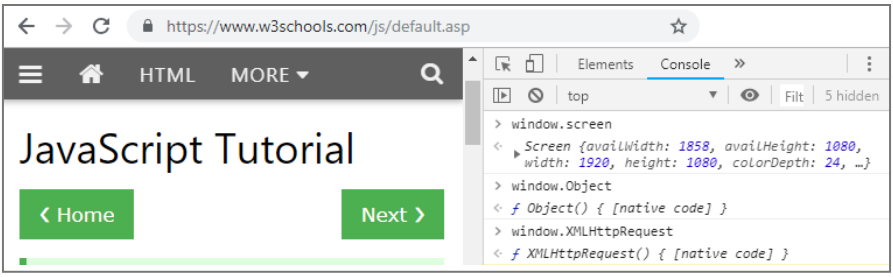


Рисунок 30

Объекты Browser Object Model

Объект Window. Открытие, перемещение и изменение размера окон

Объект «window» является основой браузерной объектной модели (BOM). Этот объект «представляет» окно браузера как главный контейнер для веб-страницы. Всё, что присутствует на этой веб-странице, так или иначе, попадает в объект «window» и становится доступным для программного управления. Кроме HTML элементов страницы в объекте «window» также собираются все переменные и функции, объявленные в различных скриптах страницы. Объект «window» поддерживается всеми видами браузеров.

Наряду с тем, что «window» является контейнером для других элементов, сам объект «window» также со-

держит методы и свойства. В основном, они нацелены на возможность получить информацию о параметрах окна браузера, а также на управление окнами. Основные свойства объекта «window», определяющие размеры и положение текущего окна (вкладки) браузера приведены в следующей таблице:

Свойство	Описание
innerHeight	Высота контента окна (без панелей браузера)
innerWidth	Ширина контента окна
outerHeight	Высота окна браузера (включая панели)
outerWidth	Ширина окна браузера
screenLeft, screenX	Отступ от левого края окна браузера до начала содержимого страницы (контента)
screenTop, screenY	Отступ от верхнего края окна браузера до начала содержимого страницы (контента)
scrollX, scrollY	Величина (в пикселях) сдвига контента за счет полос прокрутки

Все эти свойства доступны только для чтения, то есть можно узнать их величину из JavaScript, но изменения их значений не будут иметь эффекта на отображении окна.

Основные методы объекта «window», позволяющие управлять самими окнами, приведены в следующей таблице:

Метод	Описание
open()	Открывает новое пустое окно браузера
open(addr,id,attr)	Открывает новое окно и загружает страницу с адресом addr. Окну присваивается имя «id» и применяются атрибуты «attr»
stop()	Прекращает загрузку окна
close()	Закрывает окно

Метод	Описание
moveTo(X,Y)	Перемещает окно в точку экрана с заданными координатами
moveBy(dX,dY)	Сдвигает позицию окна по каждой координате
resizeTo(W,H)	Устанавливает размеры окна на заданные
resizeBy(dW,dH)	Прибавляет к размерам окна переданные значения

Метод «open» после создания нового окна возвращает ссылку на это новое окно. При помощи этой ссылки можно управлять окном, вызывая остальные методы из таблицы.

Рассмотрим пример. Откройте в браузере произвольную страницу (в текущем примере это страница с адресом «https://www.w3schools.com») и вызовите консоль разработчика. Напишите в консоли инструкцию по созданию нового окна:

```
window.open()
```

После выполнения команды (нажатия кнопки «Enter») в браузере появится новая пустая вкладка. Очевидно, что изменить размеры отдельной вкладки невозможно, т.к. она встроена в браузер наравне с остальными вкладками.

Для того чтобы иметь возможность управлять размерами и положениями нового окна необходимо создавать не вкладку, а именно отдельное окно. Это достигается путем передачи атрибута «resizable» при вызове метода создания окна. Выполним в консоли следующую команду:

```
newWin=window.open("https://itstep.org", "StepWin",
                   "resizable");
```

Как результат ее выполнения появится новое окно (а не вкладка), в котором начнется загрузка страницы «https://itstep.org». Второй аргумент «StepWin», переданный в функцию создания окна, задает имя нового окна как программного объекта. По этому имени окно может быть найдено среди множества других окон. Это имя не является названием вкладки или окна и непосредственно нигде не отображается. Для того чтобы закрыть новое окно введем в консоли команду:

```
newWin.close()
```

Окно, которое было ранее создано, закроется и исчезнет с экрана компьютера без каких-либо предупреждений.

Задавать положение и размер окна необходимо при его создании. После того как новое окно будет создано и управление вернется в консоль старого окна, возможность их взаимного влияния будет ограничена. Как мы уже видели, закрыть новое окно из старого можно, поскольку оно его и создавало. А вот поменять размеры и положение может оказаться затруднительным. Попробуйте ввести в консоли команду изменения размеров «newWin.resizeTo(500,500)». В результате будет «выброшено» исключение, запрещающее прямой доступ сайтов одного домена к сайтам другого домена (*cross-origin access*):

Рисунок 31

Задать положение и размер нового окна все-таки можно, только делать это необходимо сразу после его создания, не ожидая возврата в консоль разработчика. Команды нужно передать в одной инструкции, разделив их точкой с запятой. Введите или скопируйте в консоль следующую инструкцию:

```
newWin=window.open("https://itstep.org",
                   "StepWin",
                   "resizable");
newWin.resizeTo(500,500);
newWin.moveTo(50,50);
```

После нажатия «Enter» появится новое окно с заданными размерами (500×500 пикселей) в левом верхнем углу экрана (точнее, с отступами 50×50 пикселей от левого верхнего угла). Как уже отмечалось, возможность закрыть новое окно сохраняется и после возврата в консоль предыдущего окна браузера.

Попробуйте самостоятельно создавать новые окна с другими адресами. Используйте методы относительного характера «moveBy» и «resizeBy» для управления положением и размерами новых окон.

Объект Navigator. Параметры браузера

Объект «navigator» используется для сбора информации от операционной системы, параметрах, настройках и состоянии браузера. Этот объект доступен только для чтения, установка (запись) новых параметров не будет иметь эффекта на дальнейшей работе браузера или системы. Отметим еще раз, что объект «navigator» не яв-

ляется полностью стандартизированным, тем не менее, большинство браузеров обеспечивают его поддержку.

Информацию, содержащуюся в объекте «navigator», не следует использовать как однозначно достоверную. В силу отсутствия стандартов многие браузеры заполняют поля объекта одинаковыми значениями, просто чтобы не оставлять их пустыми. Также, браузеры могут неправильно подавать сведения об операционной системе, если она обновлялась позже, чем был установлен браузер.

Полезную информацию в объекте «navigator» содержат поля, отвечающие за настройки браузера такие, как предпочтительный язык, наличие сетевого подключения или установки разрешений на использование Cookie.

Основные свойства и методы объекта «navigator» приведены в следующей таблице

Свойство / метод	Описание	Особенности
appCodeName	Кодовое имя браузера	Содержит «Mozilla» для браузеров Chrome, Edge, Firefox, IE, Safari, и Opera
appName	Имя браузера	Содержит «Netscape» для браузеров Chrome, Edge, Firefox, IE11 и Safari
appVersion	Данные о версии браузера	Включает совместимые версии
battery или getBattery()	Данные об аккумуляторе	Возвращает объект BatteryManager. Поддерживается не всеми браузерами
connection	Данные о сетевом подключении	Содержит объект Connection
cookieEnabled	Установки разрешений Cookie	true или false
geolocation	Данные геолокации	Содержит объект Geolocation

Свойство / метод	Описание	Особенности
language	Основной язык браузера	Кодовое имя языка
languages	Допустимые языки	Массив кодовых имен
onLine	Наличие подключения к сети	true или false
platform	Платформа (операционная система) браузера	Строка с кодовым названием системы
product	Имя ядра браузера	Все браузеры содержат «Gecko»
userAgent	Данные заголовка «User-Agent», отсылаемого браузером	Объединяет appCodeName и appVersion
getUserMedia()	Доступ к медиаресурсам	Получить доступ к микрофону или камере (если они есть)
vibrate()	Управление устройством вибрирования (если есть)	Включает вибрацию на заданное время

Некоторые свойства объекта «navigator» рассчитаны на использовании в составе мобильных устройств. Например, свойство «battery» или метод «getBattery()» (в зависимости от типа браузера) позволяет получить данные о состоянии аккумулятора устройства и прогнозированном времени его работы.

Для стационарных ПК, подключенных к сети, можно увидеть, например, полный заряд «level: 1» и бесконечное время разряда «dischargingTime: Infinity». Для устройств, работающий от батареи, свойство «dischargingTime» позволяет определить прогнозируемое время работы до полного разряда (в секундах), а свойство «level» содержит данные об уровне оставшегося заряда батареи. Также свойство «charging» уведомляет о наличии (или отсутствии) подключения к зарядному устройству (рис. 33, 34).

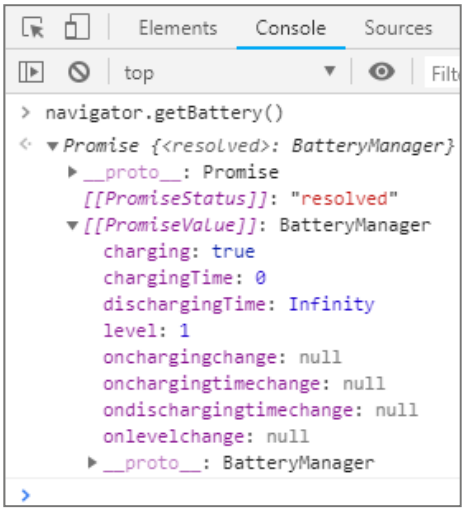


Рисунок 32

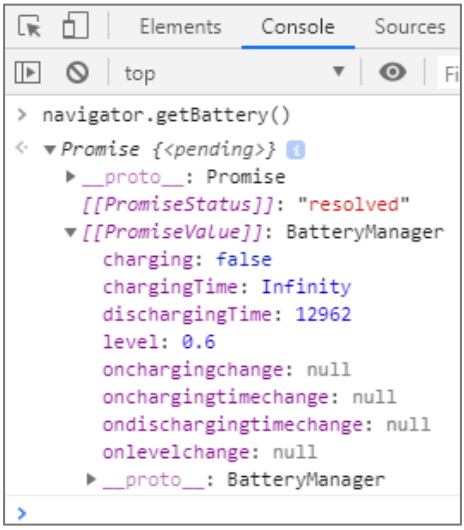


Рисунок 33

Свойство «connection» может быть использовано для получения информации о типе сетевого подключения.

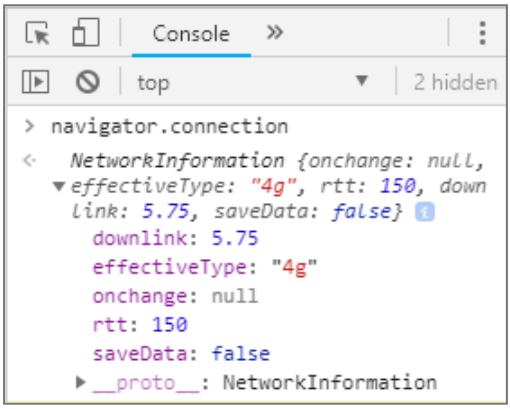


Рисунок 34

В составе объекта, возвращенного свойством «connection», присутствует информация о скорости подключения (downlink) в Мб за секунду, эквивалентный тип подключения (effectiveType) — "slow-2g", "2g", "3g" или "4g", а также время приема-передачи (round-trip time, rtt), выраженное в миллисекундах.

Свойство «geolocation» предоставляет данные о геолокации устройства, на котором установлен браузер. Данные не передаются, если на устройстве нет модуля геолокации или этот модуль отключен.

Методы объекта «navigator», такие как «getUserMedia()» или «vibrate()» дают возможность получить доступ к видеокамере и микрофону устройства, а также к средствам вибрации, если они, конечно, в устройстве присутствуют. В основном, подобные устройства распространены в мобильных телефонах или планшетах. Хотя и в стационарных ПК видеокамера может быть установлена и зарегистрирована в операционной системе.

Сведения о языках, используемых браузером, и о предпочтительном языке содержат в себе массив «languages» и строка «language» (рис. 36):

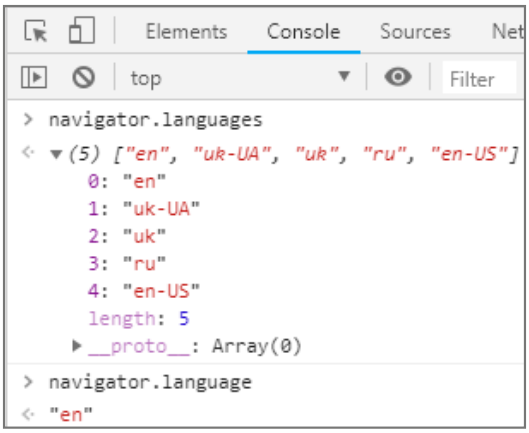


Рисунок 35

В массиве «languages» содержится набор строк, каждая их которых обозначает кодовое название языка. Порядок элементов соответствует приоритету языков, установленному пользователем в настройках операционной системы. Строка «language», по сути, является копией первого (нулевого) элемента массива «languages», то есть содержит кодовое название основного предпочитаемого языка.

Объект Screen. Свойства экрана

Информация, содержащаяся в объекте «window.screen», определяет параметры экрана монитора, такие как глубина цвета и ориентация, а также его размеры в пикселях.

Основные свойства и методы объекта «screen» приведены в следующей таблице:

Свойство, метод	Описание	Особенности
availTop	Первый доступный пиксель от верхнего края экрана	Поддерживается не всеми браузерами. Обычно содержит «0»
availLeft	Первый доступный пиксель от левого края экрана	
availHeight	Доступная высота экрана	Без учета нижней (и/или верхней) строки состояния
height	Полная высота экрана	Заданная монитором
availWidth	Доступная ширина экрана	Без учета боковых панелей
width	Полная ширина экрана	Заданная монитором
colorDepth	Глубина цвета активной палитры (бит на пиксель)	Некоторые браузеры, независимо от настроек, всегда содержат «24» (только для совместимости)
pixelDepth	Глубина цвета монитора (бит на пиксель)	
orientation / msOrientation	Ориентация экрана	В браузере Edge используется свойство msOrientation
lockOrientation()	Закрепление ориентации экрана	Устаревшие. Исключаются из современных браузеров.
unlock-Orientation()	Освобождение закрепления	

Объект «screen» также не имеет стандартизированного описания, в следствие чего различные браузеры по-разному реализуют его свойства и методы. В частности, в целях совместимости скриптов некоторые браузеры задают значения свойств объекта «screen» независимо

от реальных показателей экрана. До введения стандартов на объекты BOM не стоит полагаться на эти значения при разработке программ на JavaScript.

Наиболее достоверными и часто применимыми свойствами объекта «screen» являются те, которые отвечают за размеры экрана монитора. Среди них можно выделить две группы свойств: «width» и «height» — определяющие полные размеры экрана (ширину и высоту, соответственно), а также «availWidth» и «availHeight» — отвечающие за доступные (*от англ. available*) размеры, то есть за свободную часть окна, без учета панели задач, строк состояния, док-панелей и т.п.

Например, если на экране монитора есть одна панель задач, и она размещена по левому краю, то можно получить следующее соотношение параметров (см. рис. 37).

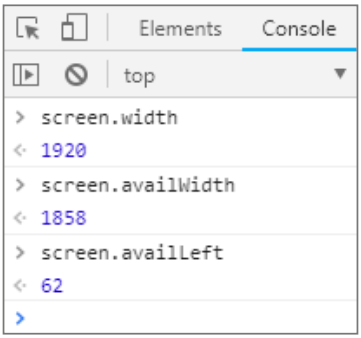


Рисунок 36

Полная ширина экрана соответствует его техническим характеристикам и составляет 1920 пикселей (screen.width). Свободная (доступная) ширина равна 1858 пикселей (screen.availWidth), поскольку часть экрана занимает панель задач. Первый доступный пиксель имеет отступ 62

от левого края (screen.availLeft), что равно ширине панели задач. Несложно убедиться, что 1858+62=1920.

Последний использованный параметр (screen.availLeft) не имеет гарантированной поддержки во всех браузерах, поэтому его применение должно сопровождаться дополнительными проверками на возможность чтения.

Полностью аналогично можно получить данные о высоте экрана — полной и доступной.

Объекты Location и History. Перемещение по страницам

Объекты «location» и «history» в структуре BOM обеспечивают управление адресами веб-страниц, которые отображаются на данной вкладке браузера. При помощи этих объектов можно перемещаться по различным страницам.

Объект «location» собирает в себе данные об веб-адресе текущей страницы, открытой в данный момент в данной вкладке. С его помощью можно узнать об этом адресе и его составных частях, а также загрузить в текущей вкладке страницу с другим адресом.

Перед тем как приступить к рассмотрению свойств и методов объекта «location», напомним о том, что такое веб-адрес и из чего он состоит.

В сети Интернет существует множество разнообразных сайтов. Мы уже знаем, что веб-сайт представляет собой некоторую программу, которая хранится на определенном компьютере (сервере). Для того чтобы браузер отобразил нам нужный сайт, он должен найти сервер, на котором этот сайт располагается и запустить ответственную за сайт программу.

При этом программа должна понимать, что от нее требуется — передать файл, выдать определенные данные или вернуть HTML код страницы. Необходимое действие задается протоколом (или схемой) — первой составной частью адреса сайта. HTTP протокол предусматривает, что от программы требуется HTML код. В таком случае, адрес сайта будет начинаться с «http://». Если при этом необходимо обеспечить шифрование данных, указывают протокол «https://» (s означает «secure» — защищенный). Если требуется получить (загрузить) сам файл, а не HTML код, используют файловый протокол FTP, а адрес будет начинаться как «ftp://». Когда мы выполняем упражнения и открываем локальные html-файлы браузер использует протокол «file://», ссылаясь на локальную файловую систему, а не к Интернет-ресурсам. Есть и другие виды протоколов.

Далее за протоколом следует доменное имя сайта. Как правило, именно эту часть адреса называют «именем сайта» или «адресом сайта». Примерами являются «itstep.org», «www.google.com» или «D:/Sources/js4_1.html».

Если на одном компьютере (сервере) обслуживаются несколько сайтов, то они могут быть разделены при помощи разных портов. Порт (сетевой порт) — это целое число, определяющее программу, которой адресуется сетевой запрос. При помощи различных портов можно запускать несколько программ на одном сетевом узле, и они не будут мешать друг другу при работе с сетью.

Как правило, порт для веб-ресурсов имеет значение «80» и явно в адресе не указывается. Однако некоторые сайты специально размещают на нестандартном пор-

ту. Существует мнение, что это более безопасно, но это утверждение спорно. В таком случае после доменного имени необходимо указать номер сетевого порта, отделив его от имени двоеточием. Следующий сайт использует 8080-й порт: «http://portquiz.net:8080»

В составе адресной строки могут также передаваться дополнительные параметры. При помощи параметров можно уточнить номер или название товара, который запрашивает пользователь, желаемый язык, день, начиная с которого нужно вывести данные об оплатах, поисковый запрос и множество других данных. Параметры позволяют создать одну веб-страницу, в которой они будут анализироваться, вместо создания множества страниц под каждый отдельный товар, язык или опорный день.

Существует два основных способа передачи параметров для веб-страницы. Метод «POST» скрывает данные в составе HTTP пакета, тогда как метод «GET» добавляет их непосредственно в адрес ресурса. Детальнее методы передачи данных мы рассмотрим в следующем уроке «Формы», пока остановимся на том, что параметры могут являться частью адреса веб-страницы и, как следствие, будут частью объекта «location» BOM.

Параметры отделяются от остального адреса знаком вопроса «?». Каждый из параметров задается по схеме «имя=значение». Если требуется передать несколько параметров, то их определения разделяются знаком «&». Например, в следующем адресе передается один параметр «page»: «http://portquiz.net:8080/?page=1». Пример передачи двух параметров «q» и «oq» иллюстрирует такой адрес «https://www.google.com/search?q=itstep&oq=itstep».

Для того чтобы при переходе на страницу была возможность ее «прокрутить» к определенному элементу, используются ссылки-якоря (anchors). Имена этих ссылок также добавляются к полному адресу страницы, отделяясь от него символом «#». Например, при переходе по адресу «https://www.w3schools.com/js/#demo» страница будет отображена, начиная от блока с именем «demo».

Полный адрес сайта также называется термином URI (*англ. Uniform Resource Identifier*) — унифицированный идентификатор ресурса. Он стандартизирован документом RFC3986, который доступен в электронном виде по ссылке https://tools.ietf.org/html/rfc3986. В нем можно более подробно прочитать о принципах формирования адресов, видах протоколов и примерах адресов.

Как уже отмечалось, для получения информации об адресе страницы и его составных частях в браузерной модели (BOM) предназначен объект «location». Его поля (свойства) заполняются при загрузке страницы в зависимости от наличия или отсутствия отдельных частей адреса.

Для примера рассмотрим адреса двух страниц, включающих в себя различные элементы:

1) «https://www.w3schools.com/js/#main» и

2) «http://portquiz.net:8080/?page=1».

Свойства объекта «location» с описаниями собраны в следующей таблице. В колонке «Значение» под номером «1» указано значение свойства, которое будет установлено при загрузке первой из указанных выше страниц, «2» — при загрузке второй.

Свойство	Описание	Значение
hash	часть адреса, начиная с символа '#'	1) "#test" 2) "" (пустая строка)
host	имя сайта и порт (если указан)	1) "www.w3schools.com" 2) "portquiz.net:8080"
hostname	имя сайта (без порта)	1) "www.w3schools.com" 2) "portquiz.net"
href	весь адрес	1) "https://www.w3schools.com/js/#main" 2) "http://portquiz.net:8080/?page=1"
pathname	строка пути (от имени сайта)	1) "/js/" 2) "/"
port	номер порта (если указан)	1) "" (пустая строка) 2) "8080"
protocol	имя протокола	1) "https:" 2) "http:"
search	часть адреса начиная с символа '? '	1) "" (пустая строка) 2) "?page=1"

Перейдите по указанным адресам, включите консоль разработчика в браузере и введите запросы на свойства объекта «location». Проверьте их значения в зависимости от модификаций адреса (рис. 38).

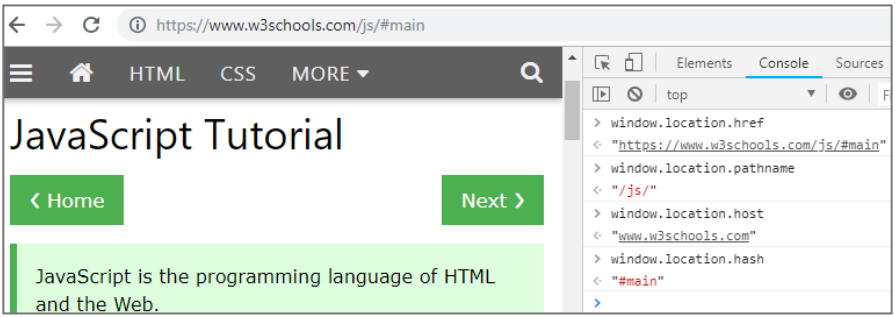


Рисунок 37

У объекта «location» существует несколько методов, предназначенных для управления адресом отображаемой страницы. Они используются для перезагрузки (обновления) страницы или для переходов на другие адреса. Основные методы объекта «location» приведены в таблице:

Метод	Описание	Пример
assign(addr)	Перейти по адресу addr	window.location. assign("https://itstep.org")
reload()	Обновить (перезагрузить) страницу	reload(true) – перезагрузить с сервера reload(false) или reload() – из кеша браузера
replace(addr)	Сменить адрес на addr	window.location. replace("https://itstep.org")

Методы «assign» и «replace» производят практически одинаковые действия с тем отличием, что при использовании метода «assign» сохраняется история переходов между страницами, тогда как применение «replace» не будет фиксироваться в истории и браузер будет запоминать лишь последний адрес в данной вкладке.

Объект «location» предусматривает возможность сокращенной формы вызова метода «assign» при помощи оператора присваивания:

```
window.location = "https://itstep.org"
```

После указанной инструкции произойдет переход на страницу с адресом «https://itstep.org» так же, как если бы был вызван метод «assign».

Также следует отметить, что изменение любого из свойств объекта «location» (кроме «hash») будет сопро-

вождаться обновлением адреса путем вызова метода «assign». Например, инструкция

```
window.location.protocol = "https:"
```

после присваивания также приведет к перезагрузке страницы с учетом нового протокола.

Якорь страницы (или фрагмент, согласно RFC3986) хранится в свойстве «location.hash». Поскольку якоря обращаются к различным элементам одной страницы, смена якоря не требует перезагрузки ее содержимого. Этим часто пользуются для создания «одностраничных приложений», в которых смена страниц происходит без их реальной загрузки. Все данные обо всех страницах загружаются один раз при первом обращении к ресурсу. При необходимости отображения новых данных или новых страниц меняется их видимость, то есть старые данные становятся невидимыми, а новые — видимыми. Для предотвращения перезагрузки страниц их адреса отличаются лишь после символа «#».

Объект браузерной модели «history» хранит в себе историю посещенных ранее страниц в данной вкладке. При помощи этого объекта можно программным способом управлять переходами к ранее просмотренным страницам или возвратами от них к тем, что были просмотрены позже. Другими словами, этот объект позволяет «листать» историю просмотров. Также, объект «history» позволяет менять историю — добавлять или модифицировать записи в истории просмотров.

Основные свойства и методы объекта «history» приведены в следующей таблице:

Свойство / метод	Описание	Особенности
length	Количество страниц в истории текущей вкладки	
state	Объект состояния, установленный для данной страницы	Задается методами pushState или replaceState
back()	Переход к предыдущей странице в истории просмотров	Аналогичен go(-1)
forward()	Переход к следующей странице в истории просмотров	Аналогичен go(1)
go(n)	Переместиться на n шагов в истории просмотров	n>0 — вперед по истории n<0 — назад по истории
pushState(state, title, addr)	Добавить запись в историю	После добавления происходит переход на добавленную страницу
replaceState(state, title, addr)	Заменить запись в истории	Заменяет текущее положение

Для иллюстрации работы методов объекта «history» рассмотрим следующую программу. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл js4_1.html*)

```
<!DOCTYPE HTML>
<html>
<head>
    <title>History object</title>
</head>
<body>
  <button onclick="history.back()">Back</button>
```

```
  <button onclick="history.forward()">Forward</button>
  <script>
     window.history.replaceState({'record':'0'},
                                 "page 0", "?record=0");
     window.history.pushState({'record':'1'},
                                 "page 1", "?record=1");
     window.history.pushState({'record':'2'},
                                 "page 2", "?record=2");
     window.history.pushState({'record':'3'},
                                 "page 3", "?record=3");
     window.history.back();
  </script>

</body>
</html>
```

В теле документа создаются две кнопки «Back» и «Forward», в обработчиках нажатия которых применены соответствующие функции объекта «history». Далее в скриптовой части производятся манипуляции с историей браузера.

Во-первых, подменяется запись текущей страницы командой «replaceState». В качестве аргументов указываем:

1. Объект-состояние с одним полем «{‹record':'0'}»
2. Название страницы «page 0». Это название используется только для журнала переходов и не является заголовком страницы, не отображается на вкладке браузера. Более того, многими браузерами это название игнорируется, пока стандарт для объекта «history» не утвержден окончательно.
3. Адрес страницы, хранимый в журнале. В качестве адреса используем добавочные данные с указанием условного номера страницы «?record=0». В журнале

истории разрешаются замены только на адрес того же домена, что и был в записи.

Во-вторых, в журнал истории добавляются три новые записи вызовами метода «pushState». Аргументы в этот метод передаются по тому же шаблону, что и для метода «replaceState».

В-третьих, вызывается метод «back()», который должен совершить переход на предыдущую страницу из журнала истории данной вкладки.

Сохраните файл и откройте его в браузере. Обратите внимание на адресную строку — в ней будет указано «js4_1.html?record=2». Из этого мы делаем вывод, что при добавлении записей в журнал истории методом «pushState», происходит автоматический переход на эту новую запись. Добавленные записи «page 1», «page 2» и «page 3» сопровождались сменой адреса текущей страницы на новый. Последняя команда «back()» вернула предпоследнюю запись из журнала, которую мы и увидели при загрузке страницы.

Проверьте работоспособность кнопок «Back» и «Forward». При движении назад по истории крайней записью (первой страницей в истории) будет страница с адресом «js4_1.html?record=0», а не «js4_1.html». Хотя первым мы открывали именно файл «js4_1.html», данная запись в журнале истории была заменена командой «replaceState» в скриптовой части страницы.

Небольшое замечание: если при работе с примером будут допущены ошибки, то после их исправления необходимо заново открыть файл в новой вкладке. Обновление текущей вкладки исправит ошибки, но не сотрет

историю. При этом работа кнопок может отличаться от описанной выше.

Коллекция Frames. Управление фреймами

В своем составе HTML-страницы могут содержать фреймы — элементы, отображающие другие страницы.

При помощи фреймов на веб-странице часто реализуется взаимодействие со сторонними ресурсами, например, в отдельных фреймах размещается видео из ресурса «YouTube» или средства картографии «Google Maps» (детальнее о размещении плееров или карт можно узнать по следующим ссылкам https://developers.google.com/youtube/player_parameters, https://developers.google.com/maps/documentation/embed/start). На рисунке 39 приведен пример размещения фрейма с видео в составе файла *js4_3.html* (*файл доступен в папке Sources*).

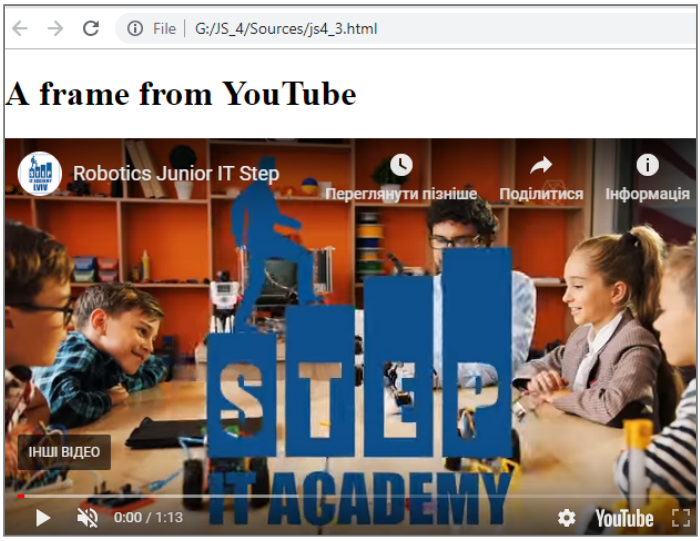


Рисунок 38

Напомним, что фреймы создаются тегом <iframe>, атрибут «src» которого указывает адрес страницы, загружаемой во фрейм. Тег является парным, то есть требует закрывающего тега:

```
<iframe src='https://itstep.org'></iframe>.
```

Фреймы являются «окнами в окне» — для них, так же как и для основной страницы, создается объект «window» со всеми свойствами и методами, со своей моделью BOM.

Все фреймы страницы хранятся в специальном объекте-коллекции «window.frames». Количество фреймов можно узнать из величины «window.frames.length». Доступ к элементам коллекции осуществляется по индексам: «window.frames[0]», «window.frames[1]» и т.д.

При создании фрейма ему может быть задано имя при помощи атрибута «name»:

```
<iframe name='itstepFrame' src='https://itstep.org'>
</iframe>
```

В таком случае доступ к элементу коллекции «window.frames» может быть осуществлен по имени фрейма в объектном стиле «window.frames.itstepFrame». Коллекция «window.frames» доступна только для чтения, изменять ее элементы не разрешается.

Для иллюстрации работы с коллекцией «window.frames» рассмотрим следующую программу. Создайте новый HTML-файл, наберите или скопируйте в него следующее содержимое (*код также доступен в папке Sources, файл js4_2.html*). При наборе обратите внимание

на комбинирование двойных и одинарных кавычек при создании обработчиков «onclick».

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Frames object</title>
    <style>
        iframe{
            border:1px solid navy;
            height:300px;
            margin:30px;
            width:300px;
        }
        button{
            margin:10px 150px;
        }
    </style>
</head>
<body>
    <iframe src="about:blank"></iframe>
    <iframe src="about:blank"></iframe>
    <iframe src="about:blank" name="frame3"></iframe>
    <button onclick="window.frames[0].
            location='https://itstep.org'">
        IT Step
    </button>
    <button onclick="window.frames[1].
            location='https://mystat.itstep.org'">
         Mystat
    </button>
    <button onclick="window.frames.frame3.
            location='https://quiz.itstep.org'">
         Quizes
    </button>
</body>
</html>
```

Откройте созданный файл в браузере. При загрузке страницы создаются три фрейма со специальной «пустой» страницей «src="about:blank"». Не рекомендуется создавать фреймы без указания источника, так как они могут быть проигнорированы браузерами при загрузке страницы.

Для наглядности фреймам установлены одинаковые размеры и рамки при помощи стилевых определений в заголовочной части программы. Ниже фреймов создаются три кнопки. Внешний вид страницы должен соответствовать приведенному на рисунке 40.

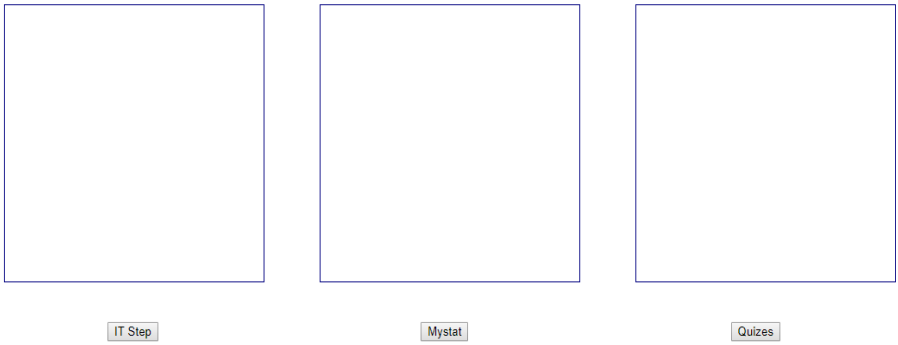


Рисунок 39

При нажатии на кнопки происходит обращение к коллекции «window.frames». Поскольку элементы этой коллекции обладают всеми атрибутами окна, используем их объект «location» для переходов на нужный адрес. Например, при нажатии первой кнопки для первого фрейма устанавливается адрес следующей командой:

```
window.frames[0].location='https://itstep.org'
```

Аналогичные команды созданы и для остальных кнопок. Для третьего фрейма использовано обращение по имени. При создании фрейма ему указан атрибут «name="frame3"», а в обработчике события нажатия кнопки его объект «location» меняется командой:

```
window.frames.frame3.location='https://quiz.itstep.org'
```

Нажмите на кнопки и убедитесь в том, что во фреймы загружаются указанные страницы.

Если есть такая необходимость, можно обеспечить обратный обмен данными: из фрейма к родительскому окну. Для этих целей предназначается объект «window.parent», который указывает на окно, в котором был создан данный фрейм. Таким образом, у всех элементов коллекции «window.frames» родительским элементом является текущее окно. Для проверки этого утверждения откройте консоль разработчика на странице с текущим примером и наберите в ней сравнение:

```
window.frames[1].parent === window
```

Как результат получим ответ «true». Проверьте данное условие для остальных членов коллекции.

В случае, если страницы, открытые в дочерних фреймах, имеют свои собственные фреймы, то реализуется более глубокая, иерархическая вложенность фреймов. Для определения самой первой, корневой страницы в таких структурах может быть использован объект «window.top».

Следует отметить, что не все сайты могут быть открыты во фреймах на других сайтах. Классически, считается что фреймы предназначены для отображения дополни-

тельных страниц одного и того же сайта, то есть страниц с одним и тем же доменным именем. Описанная ситуация называется «кросс-доменными ограничениями» (*англ.* «*same origin policy*»). Подобные ограничения действуют не только на фреймы, но и на некоторые другие способы обмена данными, например, на AJAX-коммуникации, которые мы будем рассматривать в следующих уроках.

Кросс-доменные ограничения вводятся с целью безопасности, чтобы сайты не выдавали себя за другие либо не пользовались их функциями для своих целей. Например, злоумышленник может создать фрейм размером на всю страницу с сайтом некоторого банка. При этом у него появляется возможность узнать о данных, которые пользователь будет вводить в любые поля, поскольку ввод осуществляется на самом деле на поддельной странице, содержащей фрейм.

Менее опасно, но также противоречит авторским правам использование лицензионных функций других сайтов, выдавая их за свои. Например, на сайте могут быть собраны прогнозы погоды, средства поиска, новостные анонсы от разных ресурсов, что будет поднимать популярность такого сайта за счет чужих разработок.

Если какой-либо сайт устанавливает кросс-доменное ограничение, то он не будет отображаться в фреймах сайтов с другими доменными именами. В то же время, как мы видели выше, определенные ресурсы не вводят ограничений и даже предлагают инструкции и указания по созданию фреймов для других сайтов. Перед тем как использовать фрейм необходимо убедиться, что загружаемая в него страница не содержит указанных ограничений.

Добавьте еще один фрейм с кнопкой к нашему примеру. Используйте в качестве источника фрейма пустую страницу, а в кнопке укажите новый адрес «https://www.google.com». После обновления страницы и нажатия на кнопку мы увидим предупреждение в консоли и страницу-ошибку во фрейме (рис. 41).

Из увиденного делаем вывод о том, что сайт «https://www.google.com» устанавливает кросс-доменные ограничения и не может быть помещен во фрейм стороннего сайта.

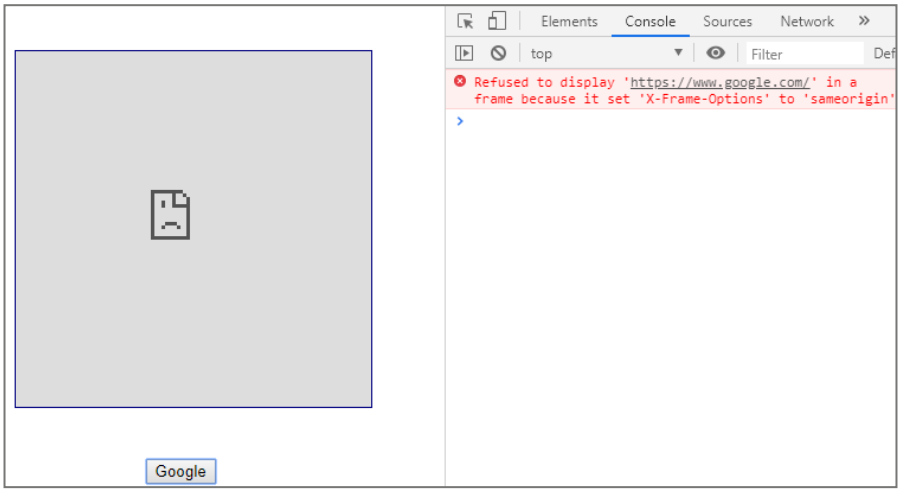


Рисунок 40

Document Object Model

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 отобра-

жается одна и та же страница, ее параметры просто не могут быть различными[1].

С позиций программной архитектуры, DOM является структурной частью BOM. Основной объект BOM «window» является родительским для главного объекта 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

[1] 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.

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, 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

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 папке 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 « » в который вложены:

- блок ;
- заголовок <h1>;
- абзац <p>, в который, в свою очередь, вложен блок ;
- 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 окончанию 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

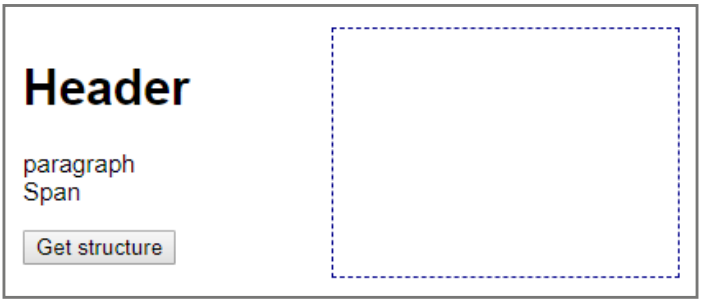


Рисунок 41

Нажмите на кнопку «Get structure». Во внутреннем блоке должно появиться сообщение:

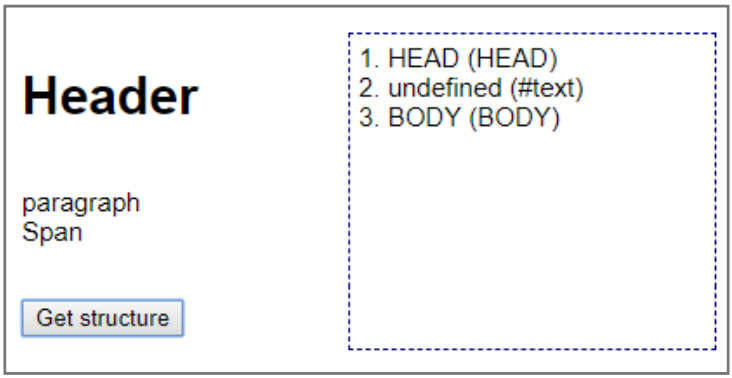


Рисунок 42

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:

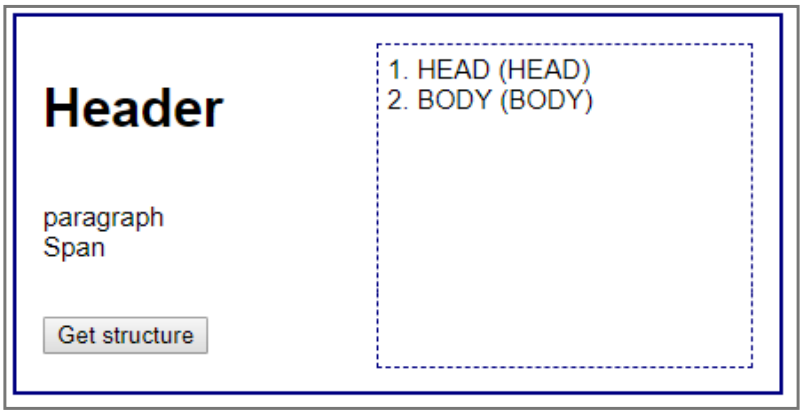


Рисунок 43

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 += "   " + (+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 += "   " + …
```

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 (рис. 45):

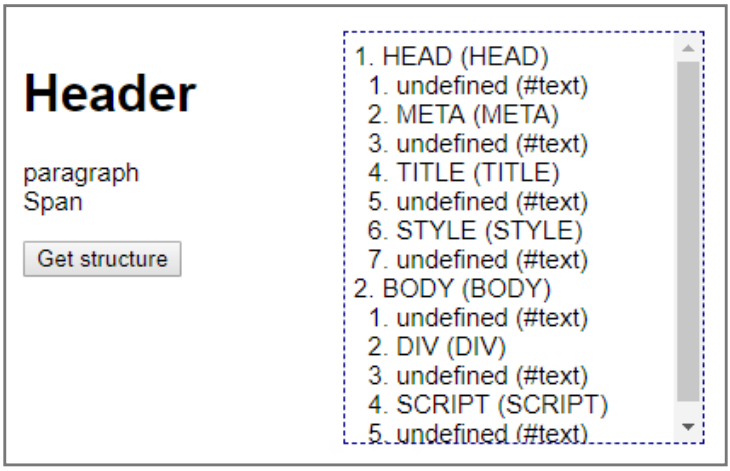


Рисунок 44

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*)

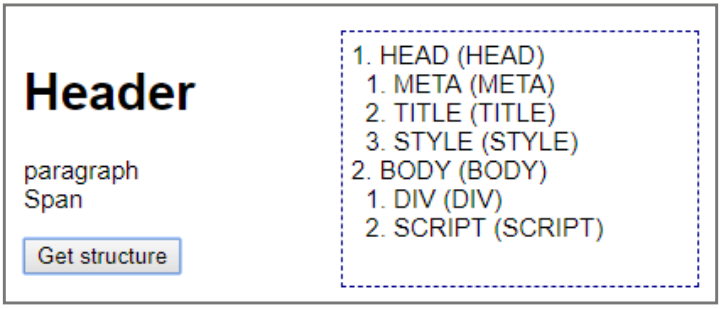


Рисунок 45

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:

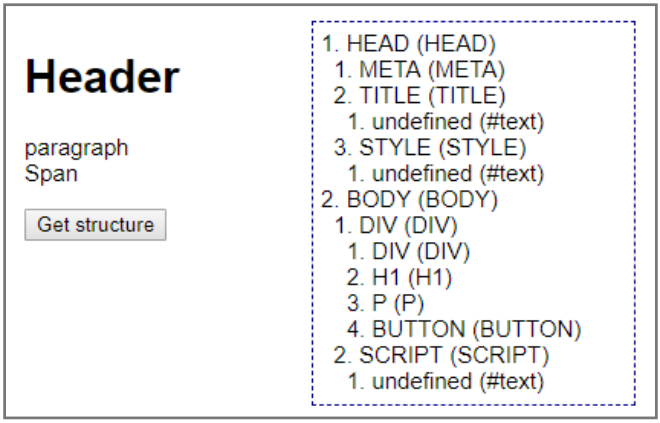


Рисунок 46

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 (рис. 48):

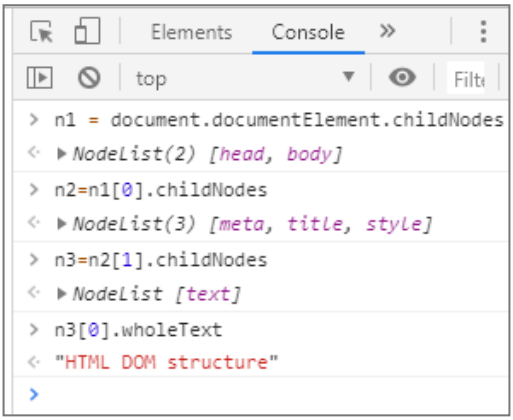


Рисунок 47

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 (ненумерованный) 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[1] (рис. 49).

В отношениях между узлами модели можно выделить три уровня:

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

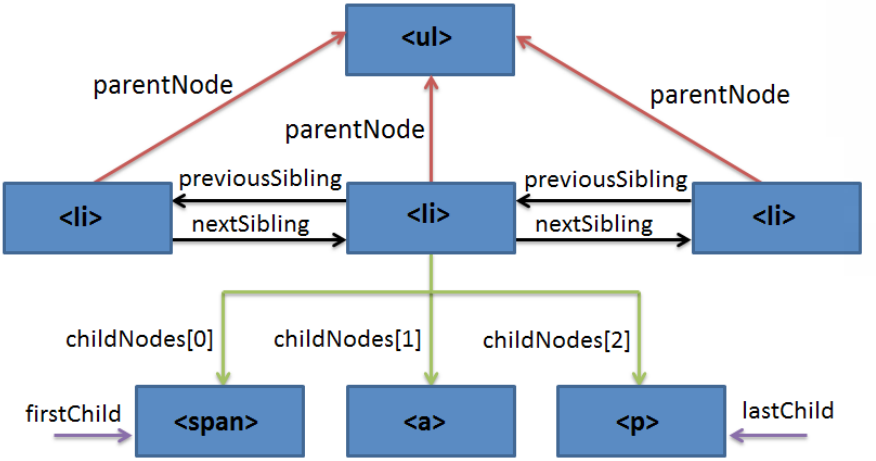


Рисунок 48

- Соседние узлы (siblings) отвечают за элементы одного уровня иерархии. Переход между такими уз-

[1] 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.

лами 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»:

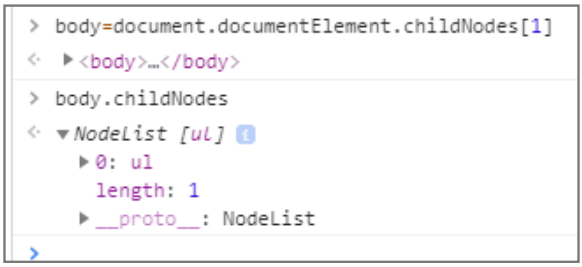


Рисунок 49

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 (рис. 51):

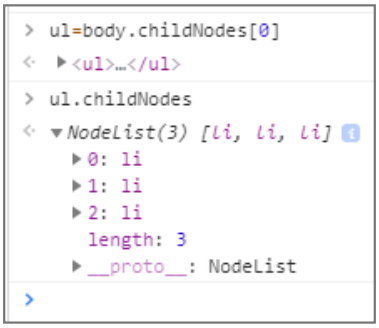


Рисунок 50

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») (рис. 52):

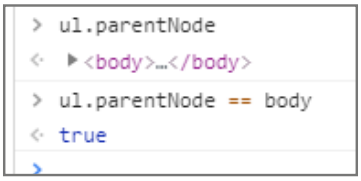


Рисунок 51

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 first element. 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 (рис. 53).

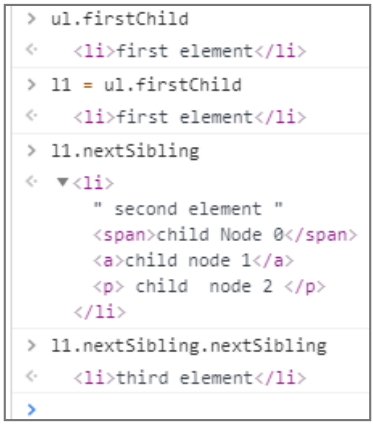


Рисунок 52

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.

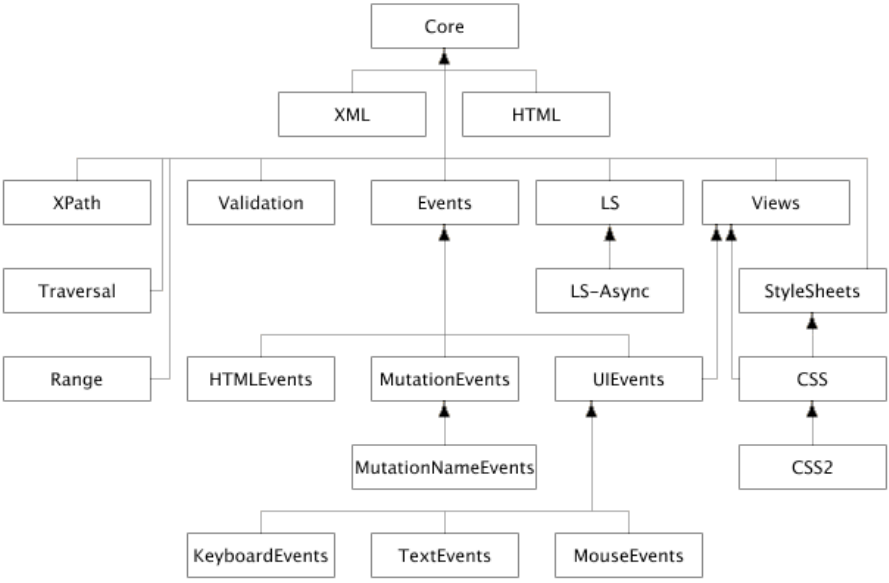


Рисунок 53

Если собрать воедино все требования, которые должны быть сформулированы при построении документа, получится следующая схема, утвержденная стандартом (рис. 54).

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 — платформо- 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 , 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 54

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.

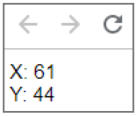


Рисунок 55

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.

Обратите внимание, 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.

Чтобы определить различные действия для различных клавиш мыши необходимо анализировать параметры события. Для этого в определение 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. Убеди-

тесь 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 (рис. 57).

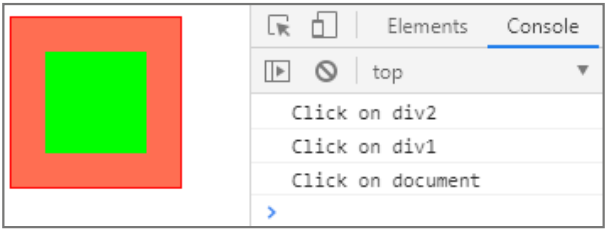


Рисунок 56

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 перемещается, 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 стилевого 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 следующем: 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:

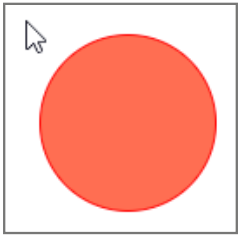


Рисунок 57

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 100×100 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 58

Такое поведение значительно лучше, однако и его можно улучшить. В реализованном способе круг перетягивается 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) следует применить команду

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

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

Продемонстрируем 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.

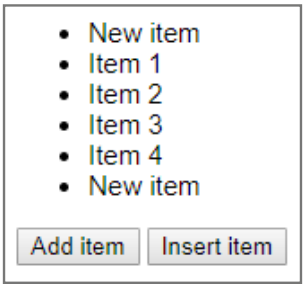


Рисунок 59

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;
      }
      .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 60

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 который является «document»:

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

Следует отметить, что браузеры сами по себе могут поддерживать технологию Drag-and-Drop, позволяя пе-

ретянуть 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 доступу 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 стилевой 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 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 убедится, 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, хранимые 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 эффект дребезга 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 мышь 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 перестановках его дочерних элементов. То есть все описанные выше команды относятся к объекту «window.list».

В завершение, рассмотрим обработчик события отпускания 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 отпускания 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 61

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. Обрабатывая 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 неоднократно 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, созданных 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
P	элементы с тегом <p>, аналог getElementsByTagName("p")
#d1	элемент с id= "d1", аналог getElementById("d1")
.c1	элементы с классом class="c1", аналог getElementsByClassName("c1")
[name="n1"]	элементы с атрибутом "name" равным "n1", в данном случае аналог getElementsByName("n1")

Селектор	описание
[type= "text"]	элементы с атрибутом "type" равным "text"
[selected]	элементы с указанным атрибутом "selected"
p.c1	элементы с тегом и классом "c1":
p, div	элементы с тегом и элементы с тегом (объединение)
p div	элементы с тегом дочерние для элементов с тегом (element)

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">…, 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 «child-

Nodes», 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 63.

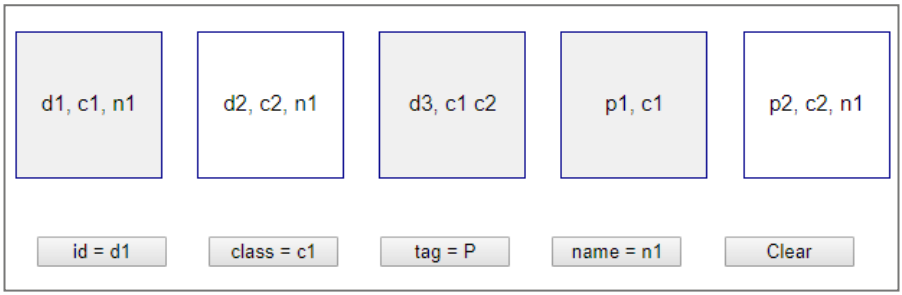


Рисунок 62

Основу страницы составляют пять блоков — три созданы тегом , 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 ( ) 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 убедиться в правильности подключения стилей и упредить 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 HML-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 и всех элементов с тегом <p>: «querySelectorAll("div,p")». Нажмите эту кнопку — цветовое 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 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 указать 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 переданного 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:

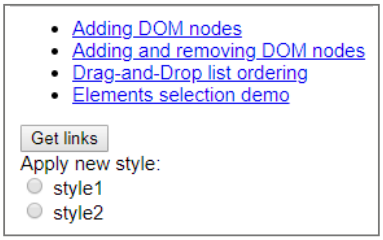


Рисунок 63

Обратите внимание на значки, используемые для маркировки списка, поскольку именно они меняются при помощи внешних стилевых файлов. Нажмите радиокнопку

«style1». Должен подключиться стилевой файл «style1.css», определяющий маркеры списка в виде пустых кружков:

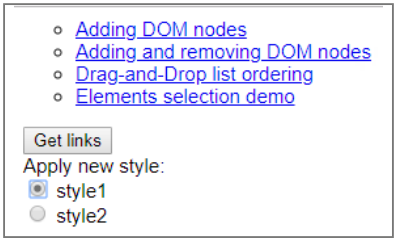


Рисунок 64

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:

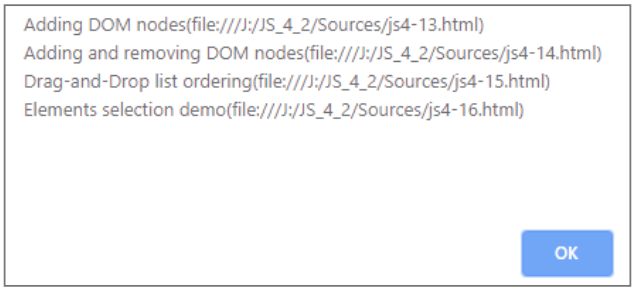


Рисунок 65

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

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».

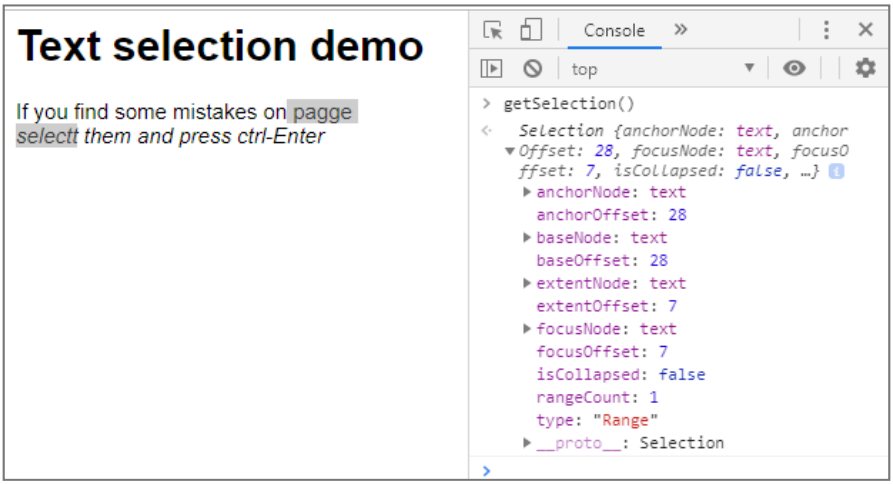


Рисунок 66

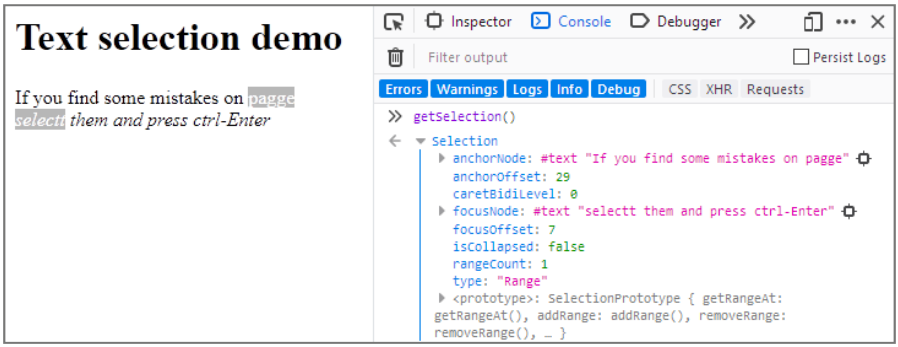


Рисунок 67

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 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:

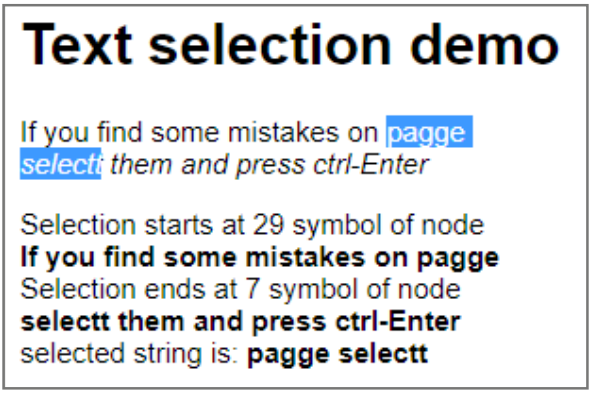


Рисунок 68

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 ☐. 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 выполнятся 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
setStart(elem, offset)	Устанавливает маркер начала в элементе «elem» со смещением «offset» от его начала
setEnd(elem, offset)	Устанавливает маркер конца в элементе «elem» со смещением «offset» от его начала
setStartBefore(elem)	Устанавливает маркер начала перед элементом «elem»
setStartAfter(elem)	Устанавливает маркер начала после элемента «elem»
setEndBefore(elem)	Устанавливает маркер конца перед элементом «elem»
setEndAfter(elem)	Устанавливает маркер конца после элемента «elem»
selectAllChildren(elem)	Устанавливает маркеры начала и конца вокруг элемента «elem»
addRange(range)	Устанавливает маркеры согласно диапазону «range»
removeAllRanges()	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» (рис. 70).

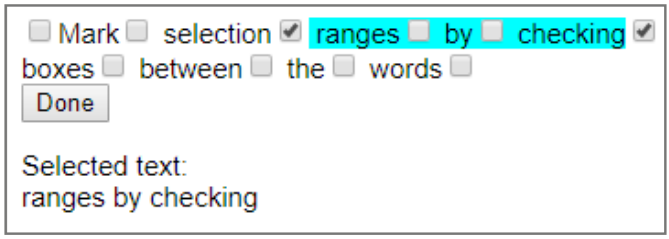


Рисунок 69

Задание для самостоятельной работы: Добавьте в созданную программу кнопку, отменяющую выделение (снимающую цветовое 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 « » 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.

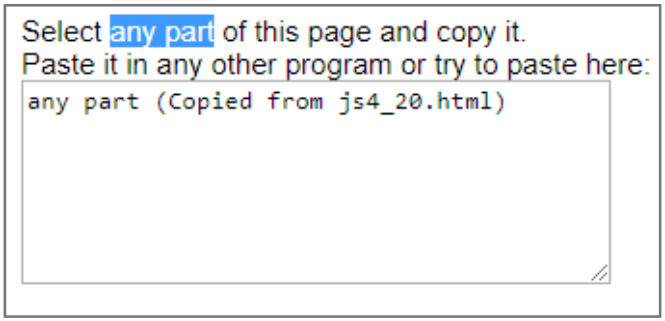


Рисунок 70

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. 71).

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.

Согласно принципам, заложенным 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. 72).

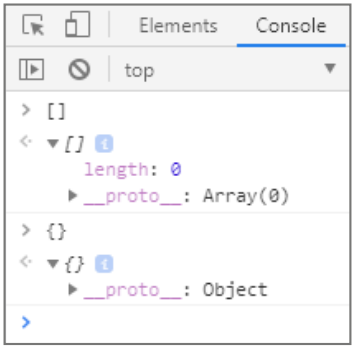


Рисунок 71

Прототипы позволяют обеспечить одинаковую функциональность для различных объектов одного типа. Например, все массивы, и даже пустой, имеют одинаковые наборы методов на подобие «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 является 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 возможность 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» чаше 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.

Домашние задания

Задания по теме: обработка событий

1. Разместите на странице 4 блока, как показано на рисунке. Напишите инструкции, меняющие цвет нижнего блока в зависимости от того, на каком из верхних блоков находится указатель мыши. Если указатель не находится ни над одним из верхних блоков, нижний блок должен приобрести серый цвет.

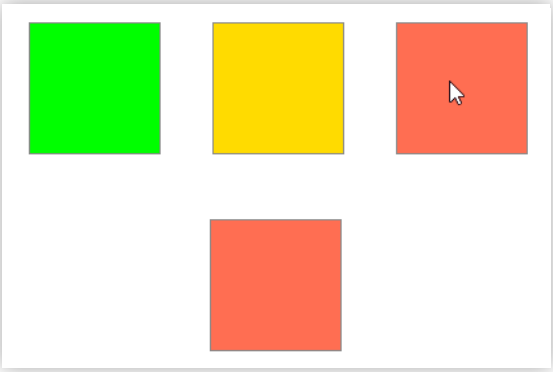


Рисунок 72

2. Разместите на странице один блок. Напишите инструкции, которые обеспечат перемещение блока при нажатии левой клавиши мыши в точку, в которой находится указатель мыши.
3. Разместите на странице 3 блока. Напишите инструкции, которые обеспечат перемещение в точку, в которой находится указатель мыши: первого блока — при нажатии левой клавиши мыши, второго блока — при

нажатии средней клавиши (или колеса), третьего блока — при нажатии правой клавиши мыши. Контекстное меню при нажатии правой клавиши мыши появляться не должно.

Задания по теме: BOM

1. Создайте HTML-страницу, разместите на ней две кнопки. По нажатию на первую должно открываться новое окно с адресом «https://itstep.org». Размеры нового окна должны быть 640×480. По нажатию на вторую кнопку новое окно должно закрываться.
2. Создайте HTML-страницу, при загрузке которой будет отображаться список установленных в системе языков. Предпочтительный язык выделите жирным шрифтом.
3. Создайте HTML-страницу, разместите на ней кнопку. По нажатию на кнопку должно выводиться количество записей в истории браузера. Проверьте работоспособность кнопки, переходя на страницу с различными параметрами (exc3.html?page = 1, exc3.html?page = 2, exc3.html?page = 3).
4. Модифицируйте пример из раздела «Фреймы». Добавьте кнопки управления историей («Назад» и «Вперед») для каждого фрейма под ним.

Задания по теме: DOM

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.

Урок № 3

Обработка событий, Browser Object Model, Document Object Model