Лабораторна робота №12

Тема. Розробка динамічних елементів **Web** – сайтів з використанням JavaScript – сценаріїв**.**

Мета. Ознайомлення із засобами обробки подій в переглядачі(браузері) та моделю DOM(BOM) переглядача. Застосування JavaScript – програм для обробки подій. Застосування регулярних виразів при обробці даних.

Завдання до лабораторної роботи №12

Розроблений Web- проект в ***лабораторних робота №1-11***, модифікувати згідно варіанту.

В Web – проекті можлива та модифікація (варіанти модифікації):

1. Для головного вікна проекту тега <BODY> запрограмувати обробку таких подій:
   1.1. onload()
   1.2. onunload()
   1.3. onClick()
2. У головному вікні для тега <H1>(<H2>..<H6>) із назвою фірми (організації) запрограмувати обробку таких подій:
   2.1. onClick()
   2.2. onMouseOver()
   2.3. onMouseOut()
3. Для головного вікна проекту передбачити обробку методів
   3.1. alert()
   3.2. confirm()
   3.3. open() та close()
   3.4. focus()
   3.5. setTimeout() та clearTimeout()
   3.6. blur()
4. На форму проекту добавити кнопку **Run Task**, при натисненні якої викликався *JavaScript* сценарій, що необхідно розробити. Сценарій повинен записувати в один із фреймів головного вікна наступну інформацію
   4.1. тип поточного браузера, яким використовує користувач Web – проекту;
   4.2. історію звернень до сторінок;
   4.3. властивість поточної сторінки;
   4.4. час звернення до сторінки;
   4.5. всі властивості об’єкта документ;
   4.6. кількість форм на сторінці;
   4.7. змінював вміст двох другорядних списків;
   4.8. вміст масиву links однієї із форм;
   4.9. вміст масиву images однієї із форм;
   4.10. розробника Web – проекту;
   4.11. імена фреймів;
   4.12. сценарій, закривав Web – проекту.
5. До форми на сторінці введення інформації (в попередніх лабораторних роботах, якщо там такої немає створити нову) добавити елементи управління :
   5.1. options
   5.2. radiobutton
   5.3. text
   5.4. кнопку очищення форми;
   5.5. кнопку виведення значень за замовченням.

   Для 5.1- 5.3 написати сценарій обробки події onChange() та onFocus().

6. Для малюнків на сайті запрограмувати обробку таких подій:
   6.1. onClick()
   6.2. onMouseOver()
   6.3. onMouseOut()
7. Створити сторінку з формою в якій розмісти елемент для введення вхідного тексту та кнопку «пошуку інформації», використовуючи **регулярні вирази** в тексті знайти:
   7.1. кількість слів;
   7.2. слова які починаються з букви та закінчуються буквою відповідно :
      7.2.1. p, s
      7.2.2. a , s
      7.2.3. e, a
      7.2.4. y, s
      7.2.5. k, o
      7.2.6. e, p
      7.2.7. b, c
   7.3. цифр
   7.4. цілих чисел

Варіанти завдань

Таблиця 2

	Вид діяльності	Обов'язкові завдання модифікації Web - проекту
1	Продаж комп'ютерів	1.1; 2.1;3.1;3.5;4.1;4.10;5.1;6.3;7.1;7.2.1;
2	Продаж комп'ютерів	1.2; 2.2;3.1;3.6;4.2;4.11;5.2;6.2;7.3;7.2.2
3	Продаж комп'ютерів	1.3; 2.3;3.1;3.4;4.3;4.12;5.3;6.1;7.4;7.2.3
4	Комп'ютерні аксесуари	1.1; 2.1;3.1;3.3;4.4;4.9;5.4;6.3;7.4;7.2.4
5	Комп'ютерні аксесуари	1.2; 2.3;3.2;3.6;4.5;4.10;5.5;6.2;7.1;7.2.5
6	Продаж книг	1.3; 2.2;3.1;3.5;4.6;4.11;5.1;6.1;7.3;7.2.6
7	Виробництво молочних виробів	1.1; 2.3;3.1;3.4;4.7;4.12;5.2;6.3;7.1;7.2.7
8	Продаж сигарет	1.2; 2.1;3.1;3.6;4.8;4.10;5.3;6.2;7.3;7.2.1
9	Продаж косметики	1.3; 2.2;3.1;3.2;4.1;4.9;5.4;6.1;7.1;7.2.2
10	Продаж алкогольних напоїв	1.1; 2.2;3.1;3.6;4.2;4.10;5.5;6.3;7.4;7.2.3
11	Продаж слабо - алкогольних напоїв	1.2; 2.3;3.2;3.5;4.3;4.11;5.1;6.2;7.4;7.2.4
12	Продаж меблів	1.3; 2.2;3.1;3.2;4.4;4.12;5.2;6.1;7.3;7.2.5
13	Продаж автомашин	1.1; 2.1;3.1;3.4;4.5;4.9;5.3;6.3;7.1;7.2.6
14	Виробництво слабо - алкогольних напоїв	1.2; 2.2;3.4;3.5;4.6;4.10;5.4;6.2;7.3;7.2.7
15	Виробництво меблів	1.3; 2.3;3.3;3.5;4.7;4.11;5.5;6.1;7.3;7.2.1

Приклад сторінки.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1251">
<meta name="Author" lang="ua" content="Гаштемеленко Яваскриптик Перлович">
<meta http-equiv="Content-Script-Type" content="text/javascript">
<meta name="Description" content="Приклади Web – сторінок. HTML Reference.">
```

```
<meta name="Keywords" content="Web - сторінка,  audio  video ">
<title>Приклад. Сторінка  audio  та  video </title>
<script>
function fnd1()
{
document.frmfnd.txta1.value="";
var result;
    str=document.frmfnd.txtf1.value;
    mask=new RegExp("0|1|2|3|4|5|6|7|8|9","g");
result=str.match(mask);
document.frmfnd.txta1.value=(result.length);
}

function fnd2()
{
document.frmfnd.txta2.value="";
var result;
    str=document.frmfnd.txtf2.value;
    mask=new RegExp("k[a-z]*o","g");
result=str.match(mask);
document.frmfnd.txta2.value=(result);
}
function myTime()
{
document.getElementById('myctime').innerHTML=new Date().toLocaleTimeString();
}
function LmyTime()
{
document.getElementById('myctime').innerHTML= " Час завантаження " + new
Date().toLocaleTimeString();
}
</script>
</head>
<body onunload ="javascript:alert('Спацював обробник!  onunload... ')"  onload
="LmyTime()" >
<h2> Лабораторна робота №13. Приклад. </h2>
<br>
<div id = "myctime">
Час завантаження
</div>
<a href="JavaScript:myTime()">
    Яка година?
</a>
</div>
<div  style = "width: 600px">
<h1 align="center">Пошук</h1>
<form id="frmfnd" name="frmfnd" method="post" action="">
 <fieldset  style = "border-style:double;  border-color: red green;" >
   <legend>Работа з текстом </legend>

 <p align="left">Введіть текст:</p>
 <input name="txtf1" type="text" size="85%"/>
 <p align="left">Результат: кількість цифр</p>
 <p>
   <input name="txta1" type="text" size="85%"/>
  </p>
 <p align="left">
   <input name="btn1" type="button" onclick="javascript:fnd1()" value="Пошук" />
  </p>
  <hr>
 <input type="submit" value="Очистити форму">
```

```
 <hr color='yellow'>

 <p align="left">Введіть текст:</p>
 <input name="txtf2" type="text" size="85%"/>
 <p align="left">Результат: слова що починаються на k і закінчуються на o</p>
 <p>
  <input name="txta2" type="text" size="85%"/>
  </p>
 <p align="left">
   <input name="btn2" type="button" onclick="javascript:fnd2()" value="Пошук" />
  </p>
 </fieldset>
</form>

<br>
</div>
<hr>
</body>
</html>
```

Додаток.

1. DOM(BOM) переглядача.

Об'єктна модель документа (англ. Document Object Model, DOM) — специфікація прикладного програмного інтерфейсу для роботи зі структурованими документами (як правило, документів XML). Визначається ця специфікація консорціумом W3C.

З точки зору об'єктно-орієнтованого програмування, DOM визначає класи, методи та атрибути цих методів для аналізу структури документів та роботи із представленям документів у вигляді дерева. Все це призначено для того, аби надати можливість комп'ютерній програмі доступу та динамічної модифікації структури, змісту та оформлення документа.

Разом із поширенням та розвитком веб-технологій і веб-переглядачів почали з'являтись різні, часто несумісні інтерфейси роботи із HTML документами в інтерпретаторах JavaScript вбудованих в веб-переглядачі. Це спонукало World Wide Web Consortium (W3C) узгодити та визначити низку стандартів, які отримали назву W3C Document Object Model (W3C DOM). Специфікації W3C не залежать від платформи або мови програмування.

Через те, що структура документа представляється у вигляді дерева, повний зміст документа аналізується та зберігається в пам'яті комп'ютера. Тому, DOM підходить для застосувань в програмах, які вимагають багаторазовий доступ до елементів документа в довільному порядку. В разі, якщо треба лише послідовний або одноразовий доступ до елементів документа, рекомендується, для пришвидшення переробки та зменшення обсягів необхідної пам'яті комп'ютера, використовувати послідовну модель роботи зі структурованими документами (SAX).

2. Реалізація DOM у веб-браузерах

Враховуючи існуючі суттєві відмінності у реалізації DOM у веб-браузерах, серед програмістів розповсюджена звичка перевіряти дієздатність тих чи інших можливостей DOM для кожного з браузерів, і тільки потім використовувати їх. Код нижче ілюструє можливість перевірки стандартів W3CDOM перед тим як запускати код, що залежить від результату перевірки.

```
if (document.getElementById && document.getElementsByTagName) {
```

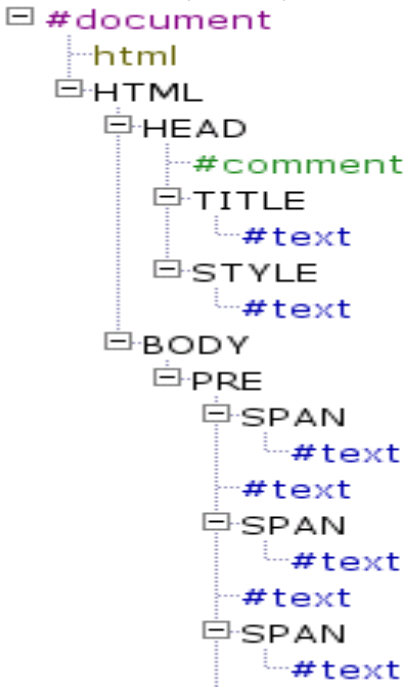
```
    // якщо методи getElementById та getElementsByTagName
    // існують, то можна з майже впевнено сподіватись на підтримку
W3CDOM.

    obj = document.getElementById("navigation")
    // далі йде інший код з використанням можливостей W3CDOM.
    // .....
}
```

Ще один фрагмент коду JavaScript, що дозволяє перевірити заявлену підтримку різних доповнень DOM у відповідному браузері.

```
<html>
<head>
<title>Test DOM Implementation</title>
<meta http-equiv="Content-Type" content="text/html; charset=windows-
1251">
<script type="text/javascript">
function domImplementationTest() {
  var featureArray = ['HTML', 'XML', 'Core', 'Views',
            'StyleSheets', 'CSS', 'CSS2', 'Events',
            'UIEvents', 'MouseEvents', 'HTMLEvents',
            'MutationEvents', 'Range', 'Traversal'];
  var versionArray = ['1.0', '2.0', '3.0'];
  var i;
  var j;
  if (document.implementation && document.implementation.hasFeature){
    document.write('<table border="1" cellpadding="2" style="border-
collapse:collapse;">');

    // header of table
    document.write('<tr>');
    document.write('<td>' + 'Підтримка доповнення ' + '</td>')
    for (j = 0; j < versionArray.length; j++) {
      document.write('<td>'+ 'версія ' + versionArray[j] + '</td>');
    }
    document.write('</tr>');

    // content of table
    for (i = 0; i < featureArray.length; i++){
      document.write('<tr>');
      document.write('<td>' + featureArray[i] + '</td>');

      for (j = 0; j < versionArray.length; j++) {
        var res = document.implementation.hasFeature(featureArray[i],
versionArray[j]);
        document.write('<td style="background-color:' + (res ? 'blue' :
'red') + '; color:white;">'+ res + '</td>');
      }

      document.write('</tr>');
    }

    document.write('</table>');
  }
}
</script>
</head>
<body>
<h1>Перевірка доповнень DOM</h1>

<script type="text/javascript">
  domImplementationTest();
```

```
</script>

</body>
</html>
```

Модель документу



Приклад відображення фрагменту DOM дерева HTML документа в програмі DOM Inspector із Mozilla Suite

Після аналізу структурованого документа, будується його представлення у вигляді дерева. Дерево, в моделі DOM, складається із множини зв'язних *вузлів* (Node) різних типів. Як правило, розрізняють вузли наступних типів:

- Документ (*Document*) — корінь дерева, представляє цілий документ.
- Фрагмент документа (*DocumentFragment*) — вузол, який є корнем піддерева основного документа.
- Елемент (*Element*) — представляє окремий елемент HTML або XML документа.
- Атрибут (*Attr*) — представляє атрибут елемента.
- Текст (*Text*) — представляє текстові дані, які містяться в елементі або атрибуті.

Стандартом визначаються і деякі інші типи вузлів у моделі документа.

Вузли деяких типів можуть мати гілки, інші ж можуть бути лише листами дерева. Спеціальні методи об'єктів вузлів дають можливість обходу дерева.

http://professorweb.ru/my/javascript/js_theory/level2/2_1.php

Обработчики событий в HTML

JavaScript-код, расположенный в теге <script>, исполняется один раз, когда содержащий его HTML-файл считывается в веб-браузер. Для обеспечения интерактивности программы на языке JavaScript должны определять обработчики событий - JavaScript-функции, которые регистрируются в веб-браузере и автоматически вызываются веб-браузером в ответ на определенные события (такие как ввод данных пользователем).

JavaScript-код может регистрировать обработчики событий, присваивая функции свойствам объектов Element (таким как onclick или onmouseover), представляющих HTML-элементы в документе.

Свойства обработчиков событий, такие как onclick, отражают HTML-атрибуты с теми же именами, что позволяет определять обработчики событий, помещая JavaScript-код в HTML-атрибуты. Например:

```
<button onClick="alert('Привет')">Щелкните меня!</button>
```

Обратите внимание на атрибут onClick. JavaScript-код, являющийся значением этого атрибута, будет выполняться всякий раз, когда пользователь будет щелкать на кнопке.

Атрибуты обработчиков событий, включенных в разметку HTML, могут содержать одну или несколько JavaScript-инструкций, отделяемых друг от друга точками с запятой. Эти инструкции будут преобразованы интерпретатором в тело функции, которая в свою очередь станет значением соответствующего свойства обработчика события.

Однако обычно в HTML-атрибуты обработчиков событий включаются простые инструкции присваивания или простые вызовы функций, объявленных где-то в другом месте. Это позволяет держать большую часть JavaScript-кода внутри сценариев и ограничивает степень взаимопроникновения JavaScript и HTML-кода. На практике многие веб-разработчики считают плохим стилем использование HTML-атрибутов обработчиков событий и предпочитают отделять содержимое от поведения.

JavaScript в URL

Еще один способ выполнения JavaScript-кода на стороне клиента - включение этого кода в URL-адресе вслед за спецификатором псевдопротокола javascript:. Этот специальный тип протокола обозначает, что тело URL-адреса представляет собою произвольный JavaScript-код, который должен быть выполнен интерпретатором JavaScript. Он интерпретируется как единственная строка, и потому инструкции в ней должны быть отделены друг от друга точками с запятой, а для комментариев следует использовать комбинации символов /* */, а не //.

«Ресурсом», который определяется URL-адресом javascript:, является значение, возвращаемое этим программным кодом, преобразованное в строку. Если программный код возвращает значение undefined, считается, что ресурс не имеет содержимого.

URL вида javascript: можно использовать везде, где допускается указывать обычные URL: в атрибуте href тега <a>, в атрибуте action тега <form> и даже как аргумент метода, такого как window.open(). Например, адрес URL с программным кодом на языке JavaScript в гиперссылке может иметь такой вид:

```
<a href="JavaScript:new Date().toLocaleTimeString();">
    Который сейчас час?
</a>
```

Некоторые браузеры (такие как Firefox) выполняют программный код в URL и используют возвращаемое значение в качестве содержимого нового отображаемого документа. Точно так же, как при переходе по ссылке http:, браузер стирает текущий документ и отображает новое содержимое. Значение, возвращаемое примером выше, не содержит HTML-теги, но если бы они имелись, браузер мог бы отобразить их точно так же, как любой другой HTML-документ, загруженный в браузер.

Другие браузеры (такие как Chrome и Safari) не позволяют URL-адресам, как в примере выше, затирать содержимое документа - они просто игнорируют возвращаемое значение. Однако они поддерживают URL-адреса вида:

```
<a href="JavaScript:alert(new Date().toLocaleTimeString());">
    Узнать время, не затирая документ
</a>
```

Когда загружается такой URL-адрес, браузер выполняет JavaScript-код, но, т.к. он не имеет возвращаемого значения (метод alert() возвращает значение undefined), такие

браузеры, как Firefox, не затирают текущий отображаемый документ. (В данном случае URL-адрес javascript: служит той же цели, что и обработчик события onclick. Ссылку выше лучше было бы выразить как обработчик события onclick элемента <button> - элемент <a> в целом должен использоваться только для гиперссылок, которые загружают новые документы.)

Если необходимо гарантировать, что URL-адрес javascript: не затрет документ, можно с помощью оператора void обеспечить принудительный возврат значения undefined:

```
<a href="javascript:void window.open('about:blank');">Открыть окно</a>
```

Без оператора void в этом URL-адресе значение, возвращаемое методом Window.open(), было бы преобразовано в строку и (в некоторых браузерах) текущий документ был бы затерт новым документом.

Подобно HTML-атрибутам обработчиков событий, URL-адреса javascript: являются пережитком раннего периода развития Веб и не должны использоваться в современных HTML-страницах. URL-адреса javascript: могут сослужить полезную службу, если использовать их вне контекста HTML-документов. Если потребуется проверить работу небольшого фрагмента JavaScript-кода, можно ввести URL-адрес javascript: непосредственно в адресную строку браузера. Другое узаконенное применение URL-адресов javascript: - создание закладок в браузерах.

Объект Window

Объект Window является средоточием всех особенностей и прикладных интерфейсов клиентского JavaScript. Он представляет окно веб-браузера или фрейм, а сослаться на него можно с помощью идентификатора **window**. Объект Window определяет свойства, такие как location, которое ссылается на объект Location, определяющий URL текущего окна и т.п.

Кроме того, объект Window определяет методы, такие как alert(), который отображает диалог с сообщением, и setTimeout(), который регистрирует функцию для вызова через указанный промежуток времени.

Объект Window в клиентском JavaScript является глобальным объектом. Это означает, что объект Window находится на вершине цепочки областей видимости и что его свойства и методы фактически являются глобальными переменными и функциями. Объект Window имеет свойство window, которое всегда ссылается на сам объект. Это свойство можно использовать для ссылки на сам объект, но обычно в этом нет необходимости, если требуется просто сослаться на свойство глобального объекта окна.

Объект Window определяет также множество других важных свойств, методов и конструкторов, которое мы и рассмотрим ниже.

Таймеры

Функции setTimeout() и setInterval() позволяют зарегистрировать функцию, которая будет вызываться один или более раз через определенные интервалы времени. Это очень важные функции для клиентского JavaScript, и поэтому они были определены как методы объекта Window, несмотря на то что являются универсальными функциями, не выполняющими никаких действий с окном.

Метод **setTimeout()** объекта Window планирует запуск функции через определенное число миллисекунд. Метод setTimeout() возвращает значение, которое может быть передано методу *clearTimeout()*, чтобы отменить запланированный ранее запуск функции.

Метод **setInterval()** похож на setTimeout(), за исключением того, что он автоматически заново планирует повторное выполнение через указанное количество миллисекунд:

```
setInterval(updateClock, 60000); // Вызывать updateClock() через каждые 60 сек
```

Подобно setTimeout(), метод setInterval() возвращает значение, которое может быть передано методу *clearInterval()*, чтобы отменить запланированный запуск функции.

В следующем примере определяется вспомогательная функция, которая ожидает указанный интервал времени, многократно вызывает указанную функцию и затем отменяет запланированные вызовы по истечении другого заданного интервала времени. Этот пример демонстрирует использование методов setTimeout(), setInterval() и clearInterval():

```
/*
*     Планирует вызов или вызовы функции f() в будущем.
*     Ожидает перед первым вызовом start миллисекунд, затем вызывает f()
*     каждые interval миллисекунд и останавливается через start+end
миллисекунд.
*     Если аргумент interval указан, а аргумент end нет, повторяющиеся
вызовы функции f
*     никогда не прекратятся. Если отсутствуют оба аргумента, interval и
end,
*     тогда функция f будет вызвана только один раз, через start
миллисекунд.
*     Если указан только аргумент f, функция будет вызвана немедленно, как
если бы
*     в аргументе start было передано число 0. Обратите внимание, что вызов
invoke()
*     не блокируется: она сразу же возвращает управление.
*/
function invoke(f, start, interval, end) {
      if (!start) start = 0;                // По умолчанию через 0 мс
      if (arguments.length <= 2)            // Случай однократного вызова
            setTimeout(f, start);           // Вызвать 1 раз через start мс
      else {      // Случай многократного вызова
            setTimeout(repeat, start); // Начать вызовы через start мс

            function repeat() {           // Планируется на вызов выше
                  var h = setInterval(f, interval);     // Вызывать f через
interval мс

                  // Прекратить вызовы через end мс, если значение end
определено
                  if (end)
                        setTimeout(function() {
                              clearInterval(h); }, end);
            }
      }
}
```

По исторически сложившимся причинам в первом аргументе методам setTimeout() и setInterval() допускается передавать строку. В этом случае строка будет интерпретироваться (как с применением функции eval()) через указанный интервал времени. Спецификация HTML5 (и все браузеры, кроме IE) допускает передавать методам setTimeout() и setInterval() дополнительные аргументы после первых двух. Все эти дополнительные аргументы будут передаваться функции, вызов которой планируется этими методами. Однако если требуется сохранить совместимость с IE, эту возможность использовать не следует.

Если методу setTimeout() указать величину интервала 0 миллисекунд, указанная функция будет вызвана не сразу, а «как только такая возможность появится», т.е. как только завершат работу все обработчики событий.

Адрес документа и навигация по нему

Свойство **location** объекта Window ссылается на объект **Location**, представляющий текущий URL-адрес документа, отображаемого в окне и определяющий методы, инициирующие загрузку нового документа в окно.

Свойство location объекта Document также ссылается на объект Location:

```
window.location === document.location;      // всегда верно
```

Кроме того, объект Document имеет свойство URL, хранящее статическую строку с адресом URL документа. При перемещении по документу с использованием идентификаторов фрагментов (таких как «#table-of-contents») внутри документа объект Location будет обновляться, отражая факт перемещения, но свойство document.URL останется неизменным.

Анализ URL

Свойство location окна является ссылкой на объект Location и представляет URL-адрес документа, отображаемого в данный момент в текущем окне. Свойство **href** объекта Location - это строка, содержащая полный текст URL-адреса. Метод *toString()* объекта Location возвращает значение свойства href, поэтому в контекстах, где неявно подразумевается вызов метода toString(), вместо конструкции location.href можно писать просто location.

Другие свойства этого объекта, такие как *protocol*, *host*, *hostname*, *port*, *pathname*, *search* и *hash*, определяют отдельные части URL-адреса. Они известны как свойства «декомпозиции URL» и также поддерживаются объектами Link (которые создаются элементами <a> и <area> в HTML-документах).

Свойства hash и search объекта Location представляют особый интерес. Свойство hash возвращает «идентификатор фрагмента» из адреса URL, если он имеется: символ решетки (#) со следующим за ним идентификатором. Свойство search содержит часть URL-адреса, следующую за вопросительным знаком, если таковая имеется, включая сам знак вопроса. Обычно эта часть URL-адреса является строкой запроса. В целом эта часть URL-адреса используется для передачи параметров и является средством встраивания аргументов в URL-адрес.

Хотя эти аргументы обычно предназначены для сценариев, выполняющихся на сервере, нет никаких причин, по которым они не могли бы также использоваться в страницах, содержащих JavaScript-код. В примере ниже приводится определение универсальной функции urlArgs(), позволяющей извлекать аргументы из свойства search URL-адреса. В примере используется глобальная функция decodeURIComponent(), имеющаяся в клиентском JavaScript:

```
/*
*     Эта функция выделяет в URL-адресе разделенные амперсандами
*     пары аргументов имя/значение из строки запроса. Сохраняет эти пары
*     в свойствах объекта и возвращает этот объект. Порядок использования:
*
*     var args = urlArgs(); // Извлечь аргументы из URL
*     var q = args.q || ""; // Использовать аргумент, если определен, или
значение по умолчанию
*     var n = args.n ? parseInt(args.n) : 10; */

function urlArgs() {
      var args = {};                              // Создать пустой объект
      var query = location.search.substring(1);   // Строка запроса без '?'
      var pairs = query.split("&");               // Разбить по амперсандам

      for(var i = 0; i < pairs.length; i++) {     // Для каждого фрагмента
            var pos = pairs[i].indexOf('=');// Отыскать пару имя/значение
            if (pos == -1) continue;                      // Не найдено -
пропустить
            var name = pairs[i].substring(0,pos);         // Извлечь имя
            var value = pairs[i].substring(pos+1);        // Извлечь значение
            value = decodeURIComponent(value);            // Преобразовать
значение
            args[name] = value;                           // Сохранить в виде
свойства
      }
```

```
    return args;                                    // Вернуть полученные
аргументы
}
```

Загрузка нового документа

Метод **assign()** объекта Location заставляет окно загрузить и отобразить документ по указанному URL-адресу. Метод **replace()** выполняет похожую операцию, но перед открытием нового документа он удаляет текущий документ из списка посещавшихся страниц.

Когда сценарию просто требуется загрузить новый документ, часто предпочтительнее использовать метод replace(), а не assign(). В противном случае кнопка Back (Назад) браузера вернет оригинальный документ и тот же самый сценарий снова загрузит новый документ. Метод location.replace() можно было бы использовать для загрузки версии веб-страницы со статической разметкой HTML, если сценарий обнаружит, что браузер пользователя не обладает функциональными возможностями, необходимыми для отображения полноценной версии:

```
// Если браузер не поддерживает объект XMLHttpRequest, выполнить
// переход к статической странице, которая не использует его
if (!XMLHttpRequest)
    location.replace("static_page.html");
```

Примечательно, что строка URL-адреса в этом примере, переданная методу replace(), представляет относительный адрес. Относительные URL-адреса интерпретируются относительно страницы, в которой они появляются, точно так же, как если бы они использовались в гиперссылке.

Кроме методов assign() и replace() объект Location определяет также метод **reload()**, который заставляет браузер перезагрузить документ. Однако более традиционный способ заставить браузер перейти к новой странице заключается в том, чтобы просто присвоить новый URL-адрес свойству location:

```
location = "http://www.professorweb.ru";
```

История посещений

Свойство **history** объекта Window ссылается на объект **History** данного окна. Объект History хранит историю просмотра страниц в окне в виде списка документов и сведений о них. Свойство length объекта History позволяет узнать количество элементов в списке, но по причинам, связанным с безопасностью, сценарии не имеют возможности получить хранящиеся в нем URL-адреса. (Иначе любой сценарий смог бы исследовать историю посещения веб-сайтов.)

Методы *back()* и *forward()* действуют подобно кнопкам Back (Назад) и Forward (Вперед) браузера: они заставляют браузер перемещаться на один шаг назад и вперед по истории просмотра данного окна. Третий метод, *go()*, принимает целочисленный аргумент и пропускает заданное число страниц, двигаясь вперед (если аргумент положительный) или назад (если аргумент отрицательный) в списке истории:

```
// Переход назад на 2 элемента, как если бы пользователь
// дважды щелкнул на кнопке Back (Назад)
history.go(-2);
```

Если окно содержит дочерние окна (такие как элементы <iframe>), истории посещений в дочерних окнах хронологически чередуются с историей посещений в главном окне. То есть вызов history.back() (например) в главном окне может вызвать переход назад, к ранее отображавшемуся документу, в одном из дочерних окон, оставив главное окно в текущем состоянии.

Современные веб-приложения способны динамически изменять содержимое страницы без загрузки нового документа. Приложениям, действующим подобным образом, может потребоваться предоставить пользователям возможность использовать кнопки Back и Forward для перехода между этими динамически созданными состояниями приложения.

Управление историей посещений до появления стандарта HTML5 представляло собой довольно сложную задачу. Приложение, управляющее собственной историей, должно было создавать новые записи в списке истории окна, связывать эти записи с информацией о состоянии, определять момент щелчка на кнопке Back, чтобы перейти к другой записи в списке, получать информацию, связанную с этой записью и воссоздавать предыдущее состояние приложения в соответствии с этой информацией.

Один из приемов реализации такого поведения опирается на использование скрытого элемента <iframe>, в котором сохраняется информация о состоянии и создаются записи в списке истории просмотра. Чтобы создать новую запись, в этот скрытый фрейм динамически записывается новый документ, с помощью методов open() и write() объекта Document. Документ должен включать всю информацию, необходимую для воссоздания соответствующего состояния приложения. Когда пользователь щелкнет на кнопке Back, содержимое скрытого фрейма изменится. До появления стандарта HTML5 не предусматривалось никаких событий, которые извещали бы об этом изменении, поэтому, чтобы определить момент щелчка на кнопке Back, приходилось использовать функцию setInterval() и с ее помощью 2-3 раза в секунду проверять наличие изменений в скрытом фрейме.

Однако на практике разработчики, когда требуется реализовать подобное управление историей просмотра, предпочитают использовать готовые решения. Многие фреймворки JavaScript включают такие решения. Например, для библиотеки jQuery существует расширение history. Существуют также автономные библиотеки управления историей. Например, одной из наиболее популярных является библиотека RSH (Really Simple History - действительно простое управление историей).

Информация о браузере и об экране

Иногда сценариям бывает необходимо получить информацию о веб-браузере, в котором они выполняются, или об экране, на котором отображается браузер. В этом разделе описываются свойства navigator и screen объекта Window. Эти свойства ссылаются, соответственно, на объекты Navigator и Screen, содержащие информацию, которая дает возможность подстроить поведение сценария под существующее окружение.

Объект Navigator

Свойство **navigator** объекта Window ссылается на объект Navigator, содержащий общую информацию о номере версии и о производителе браузера. Объект Navigator назван «в честь» браузера Netscape Navigator, но он также поддерживается во всех других браузерах. (Кроме того, IE поддерживает свойство clientInformation как нейтральный синоним для navigator. К сожалению, другие браузеры свойство с таким именем не поддерживают.)

В прошлом объект Navigator обычно использовался сценариями для определения типа браузера - Internet Explorer или Netscape. Однако такой подход к определению типа браузера сопряжен с определенными проблемами, т.к. требует постоянного обновления с появлением новых браузеров или новых версий существующих браузеров. Ныне более предпочтительным считается метод на основе проверки функциональных возможностей. Вместо того чтобы делать какие-либо предположения о браузерах и их возможностях, гораздо проще прямо проверить наличие требуемой функциональной возможности (например, метода или свойства).

Однако иногда определение типа браузера может представлять определенную ценность. Один из таких случаев - возможность обойти ошибку, свойственную определенному типу браузера определенной версии. Объект Navigator имеет четыре свойства, предоставляющих информацию о версии работающего браузера, и вы можете использовать их для определения типа браузера:

navigator.appName

Название веб-браузера. В IE это строка «Microsoft Internet Explorer». В Firefox значением этого свойства является строка «Netscape». Для совместимости с существующими реализациями определения типа браузера значением этого свойства в других браузерах часто является строка «Netscape».

navigator.appVersion

Обычно значение этого свойства начинается с номера версии, за которым следует другая информация о версии браузера и его производителе. Обычно в начале строки указывается номер 4.0 или 5.0, свидетельствующий о совместимости с четвертым или пятым поколением браузеров. Формат строки в свойстве appVersion не определяется стандартом, поэтому невозможно организовать разбор этой строки способом, не зависящим от типа браузера.

navigator.userAgent

Строка, которую браузер посылает в http-заголовке USER-AGENT. Это свойство обычно содержит ту же информацию, что содержится в свойстве appVersion, а также может включать дополнительные сведения. Как и в случае со свойством appVersion, формат представления этой информации не стандартизован. Поскольку это свойство содержит больше информации, именно оно обычно используется для определения типа браузера.

navigator.platform

Строка, идентифицирующая операционную систему (и, возможно, аппаратную платформу), в которой работает браузер.

Сложность свойств объекта Navigator делает невозможной универсальную реализацию определения типа браузера. На раннем этапе развития Всемирной паутины было написано немало программного кода, зависящего от типа браузера, проверяющего свойства, такие как navigator.appName. Создавая новые браузеры, производители обнаружили, что для корректного отображения содержимого существующих веб-сайтов они должны устанавливать значение «Netscape» в свойстве appName. По тем же причинам потерял свою значимость номер в начале значения свойства appVersion, и в настоящее время реализация определения типа браузера должна опираться на строку в свойстве navigator.userAgent, имеющую более сложный формат, чем ранее.

Пример ниже демонстрирует, как с помощью регулярных выражений можно получить из свойства navigator.userAgent название браузера и номер версии:

```
// Определяет свойства browser.name и browser.version, позволяющие выяснить
// тип клиента. Оба свойства, name и version, возвращают строки, и
// в обоих случаях значения могут отличаться от фактических
// названий браузеров и версий. Определяются следующие названия браузеров:
// "webkit": Safari или Chrome; version содержит номер сборки WebKit
// "opera": Opera; version содержит фактический номер версии браузера
// "mozilla": Firefox или другие браузеры, основанные на механизме gecko;
//    version содержит номер версии Gecko
// "msie": IE; version содержит фактический номер версии браузера
var browser = (function() {
      var s = navigator.userAgent.toLowerCase();
      var match = /(webkit)[ \/]([\w.]+)/.exec(s) ||
                        /(opera)(?:.*version)?[ \/]([\w.]+)/.exec(s) ||
                        /(msie) ([\w.]+)/.exec(s) ||
                        !/compatible/.test(s) && /(mozilla)(?:.*?
rv:([\w.]+))?/.exec(s) || [];
      return { name: match[1] || "", version: match[2] || "0" };
      }
());

window.onload = function() {
      console.log('Версия браузера %s %s', browser.name, browser.version);
};
```

В дополнение к свойствам с информацией о версии и производителе браузера, объект Navigator имеет еще несколько свойств и методов. В число стандартных и часто реализуемых нестандартных свойств входят:

navigator.onLine

Свойство navigator.onLine (если существует) определяет, подключен ли браузер к сети. Приложениям может потребоваться сохранять информацию о состоянии локально, если браузер не подключен к сети.

navigator.geolocation

Объект Geolocation, определяющий API для выяснения географического положения пользователя.

navigator.javaEnabled()

Нестандартный метод, который должен возвращать true, если браузер способен выполнять Java-апплеты.

navigator.cookiesEnabled()

Нестандартный метод, который должен возвращать true, если браузер способен сохранять cookies. Если браузер настроен на сохранение cookies только для определенных сайтов, этот метод может возвращать некорректное значение.

Объект Screen

Свойство **screen** объекта Window ссылается на объект Screen, предоставляющий информацию о размере экрана на стороне пользователя и доступном количестве цветов. Свойства *width* и *height* возвращают размер экрана в пикселах. Свойства *availWidth* и *availHeight* возвращают фактически доступный размер экрана; из них исключается пространство, требуемое для таких графических элементов, как панель задач. Свойство *colorDepth* возвращает количество битов на пиксел, определяющих цвет. Типичными значениями являются 16, 24 и 32.

Свойство window.screen и объект Screen, на который оно ссылается, являются нестандартными, но они реализованы практически во всех браузерах. Объект Screen можно использовать, чтобы определить, не выполняется ли веб-приложение на устройстве с маленьким экраном, таком как нетбук. При ограниченном пространстве экрана, например, можно было бы использовать шрифты меньшего размера и маленькие изображения.

Диалоговые окна

Объект Window обладает тремя методами для отображения простейших диалоговых окон. Метод **alert()** выводит сообщение и ожидает, пока пользователь закроет диалоговое окно. Метод **confirm()** предлагает пользователю щелкнуть на кнопке OK или Cancel (Отмена) и возвращает логическое значение. Метод **prompt()** выводит сообщение, ждет ввода строки пользователем и возвращает эту строку. Ниже демонстрируется пример использования всех трех методов:

```
do {
var name = prompt("Введите ваше имя");          // Вернет строку
var correct = confirm("Вы ввели '" + name + "'.\n" +// Вернет логич. знач.
"Щелкните OK, чтобы продолжить, или Отмена, чтобы повторить ввод.");
} while(!correct)

alert("Привет, " + name);           // Выведет простое сообщение
```

Методы alert(), confirm() и prompt() чрезвычайно просты в использовании, но правила хорошего дизайна требуют, чтобы они применялись как можно реже. Диалоги, подобные этим, нечасто используются в Веб, и большинство пользователей сочтет диалоговые окна, выводимые этими методами, выпадающими из обычной практики. Единственный вариант, когда имеет смысл обращаться к этим методам - это отладка. JavaScript-программисты часто вставляют вызов метода alert() в программный код, пытаясь диагностировать возникшие проблемы.

Обратите внимание, что текст, отображаемый методами alert(), confirm() и prompt() в диалогах - это обычный неформатированный текст. Его можно

форматировать только пробелами, переводами строк и различными знаками пунктуации.

Методы confirm() и prompt() являются блокирующими, т.е. они не возвращают управление, пока пользователь не закроет отображаемые ими диалоговые окна. Это значит, что, когда выводится одно из этих окон, программный код прекращает выполнение, и текущий загружаемый документ, если таковой существует, прекращает загружаться до тех пор, пока пользователь не отреагирует на запрос. В большинстве браузеров метод alert() также является блокирующим и ожидает от пользователя закрытия диалогового окна, но это не является обязательным требованием.

В дополнение к методам alert(), confirm() и prompt() в объекте Window имеется более сложный метод, **showModalDialog()**, отображающий модальный диалог, содержащий разметку HTML, и позволяющий передавать аргументы и получать возвращаемое значение.

Метод showModalDialog() выводит модальный диалог в отдельном окне браузера. Первым аргументом методу передается URL, определяющий HTML-содержимое диалога. Во втором аргументе может передаваться произвольное значение (допускается передавать массивы и объекты), которое будет доступно сценарию в диалоге, как значение свойства window.dialogArguments. Третий аргумент - нестандартный список пар имя/значение, разделенных точками с запятой, который, если поддерживается, может использоваться для настройки размеров и других атрибутов диалогового окна. Для определения размеров окна диалога можно использовать параметры «dialogwidth» и «dialogheight», а чтобы позволить пользователю изменять размеры окна, можно определить параметр «resizable=yes».

Окно, отображаемое эти методом, является модальным, и метод showModalDialog() не возвращает управление, пока окно не будет закрыто. После закрытия окна значение свойства **window.returnValue** становится возвращаемым значением метода. Обычно разметка HTML диалога должна включать кнопку OK, которая записывает желаемое значение в свойство returnValue и вызывает window.close().

В примере ниже приводится разметка HTML для использования с методом showModalDialog(). Комментарий в начале примера включает пример вызова showModalDialog(), а на рисунке показано диалоговое окно, созданное вызовом из примера. Обратите внимание, что большая часть текста, отображаемого в окне, передается методу showModalDialog() во втором аргументе, а не является жестко определенной частью разметки HTML:

```
<!--
      Это не самостоятельный HTML-файл. Он должен вызываться
    методом showModalDialog() и ожидает получить в свойстве
    window.dialogArguments массив строк. Первый элемент массива -
    строка, отображаемая в верхней части диалогового окна. Все остальные элементы -
    метки для однострочных текстовых полей ввода. Возвращает массив
    значений полей ввода после щелчка на кнопке Ok.
    Этот файл используется следующим образом:

    var p = showModalDialog("modal_file.html",
       ["Вставте координаты 3х мерного объекта", "x", "y", "z"],
        "dialogwidth:400; dialogheight:300; resizable:yes");

-->
<meta charset="utf-8">
<form>
      <fieldset id="fields"></fieldset>                    <!-- Тело, заполняемое
сценарием ниже -->
      <div style="text-align:center">                      <!-- Кнопки закрытия окна
-->
            <button onclick="ok()">Ok</button>             <!-- Устанавливает
возвращаемое значение и закрывает окно -->
            <button onclick="cancel()">Отмена</button>    <!-- Закрывает диалоговое
окно, не возвращая ничего-->
```

```
        </div>

        <script>
        // Создает разметку HTML диалогового окна и отображает ее в элементе fieldset
        var args = dialogArguments;
        var text = "<legend>" + args[0] + "</legend>";

        for(var i = 1; i < args.length; i++)
               text += "<label>" + args[i] + ": <input id='f" + i + "'></label><br>";

        document.getElementById("fields").innerHTML = text;

        // Закрывает диалоговое окно без установки возвращаемого значения
        function cancel() { window.close(); }

        // Читает значения полей ввода и устанавливает возвращаемое значение,
        // затем закрывает окно
        function ok() {
               window.returnValue = [];   // Возвращаемый массив
               for(var i = 1; i < args.length; i++) // Значения элементов из полей
ввода
                      window.returnValue[i-1] = document.getElementById("f" +
i).value;

               window.close(); // Закрыть диалоговое окно. Это заставит
showModalDialog() вернуть управление.
}

</script> </form>
```

Обработка ошибок

Свойство **onerror** объекта Window - это обработчик событий, который вызывается во всех случаях, когда необработанное исключение достигло вершины стека вызовов и когда браузер готов отобразить сообщение об ошибке в консоли JavaScript. Если присвоить этому свойству функцию, функция будет вызываться всякий раз, когда в окне будет возникать ошибка выполнения программного кода JavaScript: присваиваемая функция станет обработчиком ошибок для окна.

Исторически сложилось так, что обработчику события onerror объекта Window передается три строковых аргумента, а не единственный объект события, как в других обработчиках. (Другие объекты в клиентском JavaScript также имеют обработчики onerror, обрабатывающие различные ошибочные ситуации, но все они являются обычными обработчиками событий, которым передается единственный объект события.) Первый аргумент обработчика window.onerror - это сообщение, описывающее произошедшую ошибку. Второй аргумент - это строка, содержащая URL-адрес документа с JavaScript-кодом, приведшим к ошибке. Третий аргумент - это номер строки в документе, где произошла ошибка.

Помимо этих трех аргументов важную роль играет значение, возвращаемое обработчиком onerror. Если обработчик onerror возвращает true, это говорит браузеру о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, браузер не должен выводить собственное сообщение об ошибке. К сожалению, по историческим причинам в Firefox обработчик ошибок должен возвращать true, чтобы сообщить о том, что ошибка обработана.

Обработчик onerror является пережитком первых лет развития JavaScript, когда в базовом языке отсутствовала инструкция try/catch обработки исключений. В современном программном коде этот обработчик используется редко.

Элементы документа как свойства окна

Если для именования элемента в HTML-документе используется атрибут id и если объект Window еще не имеет свойства, имя которого совпадает со значением этого атрибута, объект Window получает неперечислимое свойство с именем,

соответствующим значению атрибута id, значением которого становится объект HTMLElement, представляющий этот элемент документа.

Как вы уже знаете, объект Window играет роль глобального объекта, находящегося на вершине цепочки областей видимости в клиентском JavaScript. Таким образом, вышесказанное означает, что атрибуты id в HTML-документах становятся глобальными переменными, доступными сценариям. Если, например, документ включает элемент <button id="ok"/>, на него можно сослаться с помощью глобальной переменной ok.

Однако важно отметить, что этого не происходит, если объект Window уже имеет свойство с таким именем. Элементы с атрибутами id, имеющими значение «history», «location» или «navigator», например, не будут доступны через глобальные переменные, потому что эти имена уже используются. Аналогично, если HTML-документ включает элемент с атрибутом id, имеющим значение «x» и в сценарии объявляется и используется глобальная переменная x, явно объявленная переменная скроет неявную переменную, ссылающуюся на элемент.

Если переменная объявляется в сценарии, который в документе находится выше именованного элемента, наличие переменной будет препятствовать появлению нового свойства окна. А если переменная объявляется в сценарии, который находится ниже именованного элемента, первая же инструкция присваивания значения этой переменной затрет значение неявно созданного свойства.

Неявное использование идентификаторов элементов в качестве глобальных переменных - это пережиток истории развития веб-браузеров. Эта особенность необходима для сохранения обратной совместимости с существующими веб-страницами, но использовать ее сейчас не рекомендуется - в любой момент производители браузеров могут определить новое свойство в объекте Window, что нарушит работу любого программного кода, использующего неявно определяемое свойство с этим именем. Поиск элементов лучше выполнять явно, с помощью метода document.getElementById().

Открытие и закрытие окон

Открыть новое окно веб-браузера (или вкладку, что обычно зависит от настроек браузера) можно с помощью метода **open()** объекта Window. Метод Window.open() загружает документ по указанному URL-адресу в новое или в существующее окно и возвращает объект Window, представляющий это окно. Он принимает четыре необязательных аргумента:

Первый аргумент open() - это URL-адрес документа, отображаемого в новом окне. Если этот аргумент отсутствует (либо является пустой строкой), будет открыт специальный URL пустой страницы about:blank.

Второй аргумент open() - это строка с именем окна. Если окно с указанным именем уже существует (и сценарию разрешено просматривать содержимое этого окна), используется это существующее окно. Иначе создается новое окно и ему присваивается указанное имя. Если этот аргумент опущен, будет использовано специальное имя «_blank», т.е. будет открыто новое неименованное окно.

Обратите внимание, что сценарии не могут просто так указывать имена окон и не могут получать контроль над окнами, используемыми другими веб-приложениями: они могут указывать имена только тех существующих окон, которыми им «разрешено управлять» (термин взят из спецификации HTML5). Проще говоря, сценарий может указать имя существующего окна, только если это окно содержит документ, происходящий из того же источника, или если это окно было открыто самим сценарием (или рекурсивно открытым окном, которое открыло это окно).

Кроме того, если одно окно является фреймом, вложенным в другое окно, сценарии в любом из них получают возможность управлять другим окном. В этом случае можно использовать зарезервированные имена «_top» (для обозначения

вмещающего окна верхнего уровня) и «_parent» (для обозначения ближайшего вмещающего окна).

Третий необязательный аргумент open() - это список параметров, определяющих размер и видимые элементы графического пользовательского интерфейса нового окна. Если опустить этот аргумент, окно получает размер по умолчанию и полный набор графических элементов: строку меню, строку состояния, панель инструментов и т.д. В браузерах, поддерживающих вкладки, это обычно приводит к созданию новой вкладки. Этот третий аргумент является нестандартным, и спецификация HTML5 требует, чтобы браузеры игнорировали его.

Указывать четвертый аргумент open() имеет смысл, только если второй аргумент определяет имя существующего окна. Этот аргумент - логическое значение, определяющее, должен ли URL-адрес, указанный в первом аргументе, заменить текущую запись в истории просмотра окна (true) или требуется создать новую запись (false). Если этот аргумент опущен, используется значение по умолчанию false.

Значение, возвращаемое методом open(), является объектом Window, представляющим вновь созданное окно. Этот объект позволяет сослаться в JavaScript-коде на новое окно так же, как исходный объект Window ссылается на окно, в котором выполняется сценарий:

```
window.onload = function() {
  var w = window.open();                    // Открыть новое пустое окно
  w.alert("Будет открыт сайт http://professorweb.ru");              // Вызвать
его метод alert()
  w.location = "http://professorweb.ru";               // Установить св-во
location
};
```

В окнах, созданных методом window.open(), свойство **opener** ссылается на объект Window сценария, открывшего его. В других случаях свойство opener получает значение null:

```
w.opener !== null;       // Верно для любого окна w, созданного
методом open()
w.open().opener === w;                  // Верно для любого окна
```

Метод Window.open() часто используется рекламодателями для создания «всплывающих окон» с рекламой, когда пользователь путешествует по Всемирной паутине. Такие всплывающие окна могут раздражать пользователя, поэтому большинство веб-браузеров реализуют механизм блокирования всплывающих окон. Обычно вызов метода open() преуспевает, только если он производится в ответ на действия пользователя, такие как щелчок мышью на кнопке или на ссылке. Попытка открыть всплывающее окно, которая производится, когда браузер просто загружает (или выгружает) страницу, как в приведенном примере, обычно оканчивается неудачей.

Новое окно открывается при помощи метода open() и закрывается при помощи метода **close()**. Если объект Window был создан сценарием, то этот же сценарий сможет закрыть его следующей инструкцией:

```
w.close();
```

Отношения между фреймами

Мы уже видели, что метод open() объекта Window возвращает новый объект Window, свойство opener которого ссылается на первоначальное окно. Таким образом, два окна могут ссылаться друг на друга, и каждое из них может читать свойства и вызывать методы другого. То же самое возможно для фреймов. Сценарий, выполняющийся в окне или фрейме, может ссылаться на объемлющее или вложенное окно или фрейм при помощи свойств, описываемых ниже.

Как вы уже знаете, сценарий в любом окне или фрейме может сослаться на собственное окно или фрейм с помощью свойства window или self. Фрейм может сослаться на объект Window вмещающего окна или фрейма с помощью свойства **parent**.

Если фрейм находится внутри другого фрейма, содержащегося в окне верхнего уровня, то он может сослаться на окно верхнего уровня так: parent.parent. Однако в качестве универсального сокращения имеется свойство **top**: независимо от глубины вложенности фрейма его свойство top ссылается на содержащее его окно самого верхнего уровня.

Каждый объект Window имеет свойство **document**, ссылающееся на объект Document. Этот объект Document не является автономным объектом. Он является центральным объектом обширного API, известного как объектная модель документа (DOM), который определяет порядок доступа к содержимому документа.

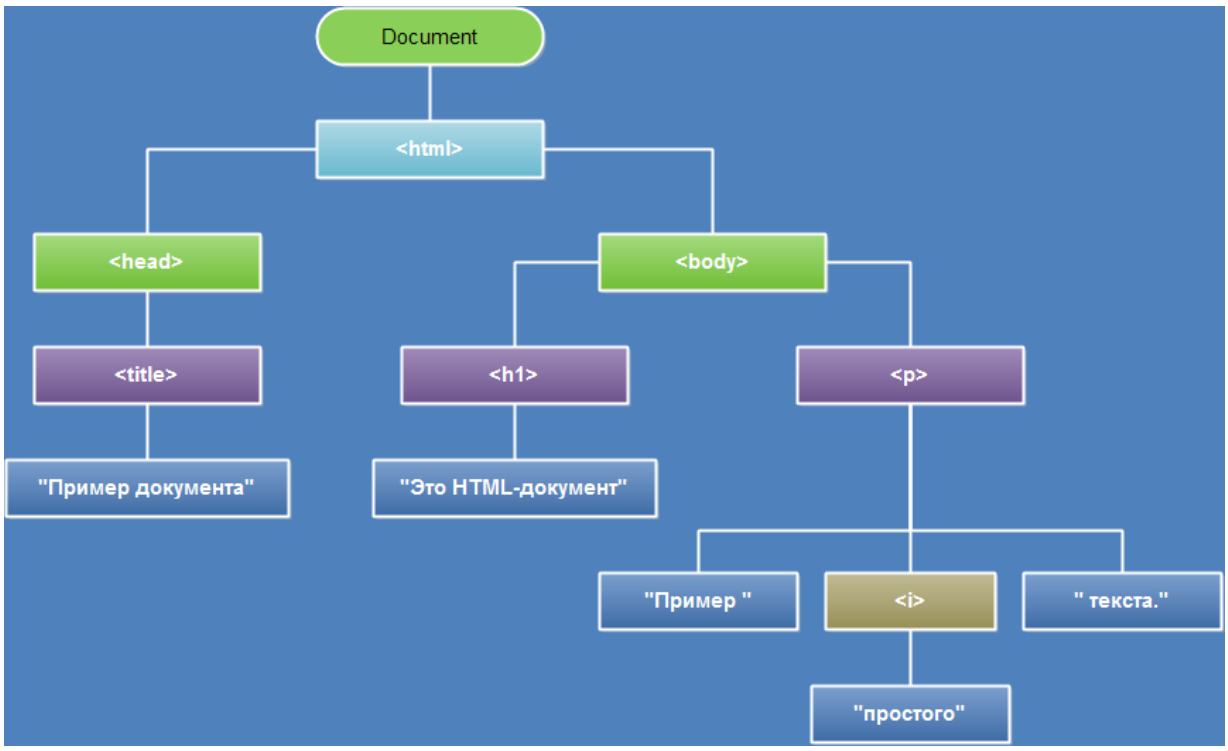
Обзор модели DOM

Объектная модель документа (Document Object Model, DOM) - это фундаментальный прикладной программный интерфейс, обеспечивающий возможность работы с содержимым HTML и XML-документов. Прикладной программный интерфейс (API) модели DOM не особенно сложен, но в нем существует множество архитектурных особенностей, которые вы должны знать.

Прежде всего, следует понимать, что вложенные элементы HTML или XML-документов представлены в виде дерева объектов DOM. Древовидное представление HTML-документа содержит узлы, представляющие элементы или теги, такие как <body> и <p>, и узлы, представляющие строки текста. HTML-документ также может содержать узлы, представляющие HTML-комментарии. Рассмотрим следующий простой HTML-документ:

```
<html>
  <head>
        <title>Пример документа</title>
  </head>
  <body>
        <h1>Это HTML-документ</h1>
        <p>Пример <i>простого</i> текста.</p>
  </body>
</html>
```

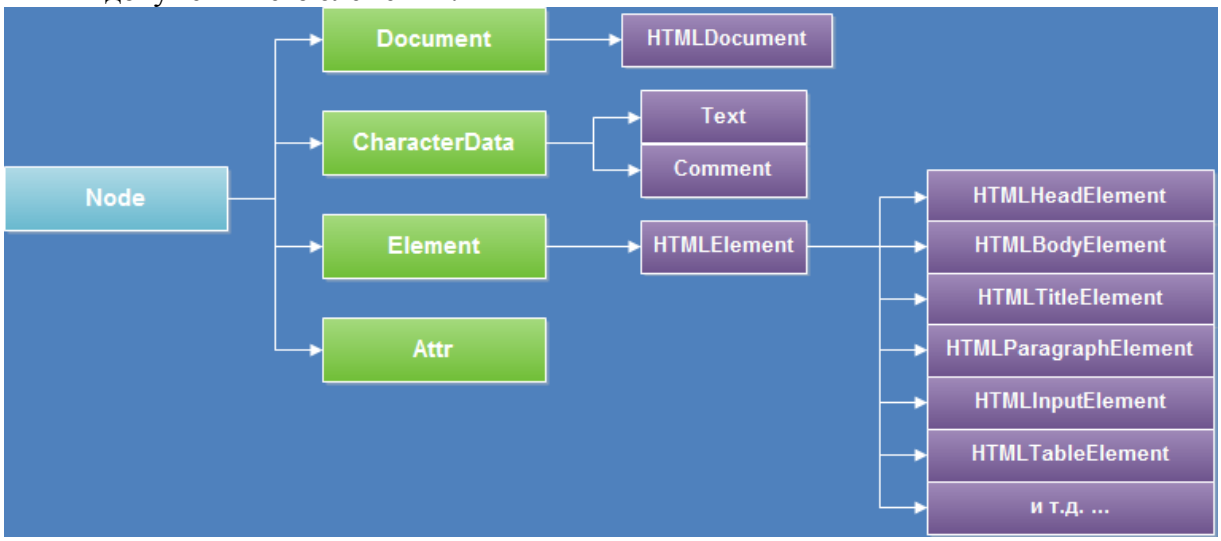
DOM-представление этого документа приводится на следующей диаграмме:



Тем, кто еще не знаком с древовидными структурами в компьютерном программировании, полезно узнать, что терминология для их описания была заимствована у генеалогических деревьев. Узел, расположенный непосредственно над данным узлом, называется *родительским* по отношению к данному узлу. Узлы, расположенные на один уровень ниже другого узла, являются *дочерними* по отношению к данному узлу. Узлы, находящиеся на том же уровне и имеющие того же родителя, называются *сестринскими*. Узлы, расположенные на любое число уровней ниже другого узла, являются его потомками. Родительские, прародительские и любые другие узлы, расположенные выше данного узла, являются его предками.

Каждый прямоугольник на этой диаграмме является узлом документа, который представлен объектом **Node**. Обратите внимание, что на рисунке изображено три различных типа узлов. Корнем дерева является узел Document, который представляет документ целиком. Узлы, представляющие HTML-элементы, являются узлами типа Element, а узлы, представляющие текст, - узлами типа Text. Document, Element и Text - это подклассы класса Node. Document и Element являются двумя самыми важными классами в модели DOM.

Тип Node и его подтипы образуют иерархию типов, изображенную на диаграмме ниже. Обратите внимание на формальные отличия между обобщенными типами Document и Element, и типами HTMLDocument и HTMLElement. Тип Document представляет HTML и XML-документ, а класс Element представляет элемент этого документа. Подклассы HTMLDocument и HTMLElement представляют конкретно HTML-документ и его элементы:



На этой диаграмме следует также отметить наличие большого количества подтипов класса HTMLElement, представляющих конкретные типы HTML-элементов. Каждый из них определяет JavaScript-свойства, отражающие HTML-атрибуты конкретного элемента или группы элементов. Некоторые из этих специфических классов определяют дополнительные свойства или методы, которые не являются отражением синтаксиса языка разметки HTML.

Выбор элементов документа

Работа большинства клиентских программ на языке JavaScript так или иначе связана с манипулированием элементами документа. В ходе выполнения эти программы могут использовать глобальную переменную document, ссылающуюся на объект Document. Однако, чтобы выполнить какие-либо манипуляции с элементами документа, программа должна каким-то образом получить, или выбрать, объекты Element, ссылающиеся на эти элементы документа. Модель DOM определяет несколько способов выборки элементов. Выбрать элемент или элементы документа можно:

по значению атрибута id;
по значению атрибута name;
по имени тега;
по имени класса или классов CSS;
по совпадению с определенным селектором CSS.

Все эти приемы выборки элементов описываются в следующих подразделах.

Выбор элементов по значению атрибута id

Все HTML-элементы имеют атрибуты id. Значение этого атрибута должно быть уникальным в пределах документа - никакие два элемента в одном и том же документе не должны иметь одинаковые значения атрибута id. Выбрать элемент по уникальному значению атрибута id можно с помощью метода **getElementById()** объекта Document:

```
var section1 = document.getElementById("section1");
```

Это самый простой и самый распространенный способ выборки элементов. Если сценарию необходимо иметь возможность манипулировать каким-то определенным множеством элементов документа, присвойте значения атрибутам id этих элементов и используйте возможность их поиска по этим значениям.

В версиях Internet Explorer ниже IE8 метод getElementById() выполняет поиск значений атрибутов id без учета регистра символов и, кроме того, возвращает элементы, в которых будет найдено совпадение со значением атрибута name.

Выбор элементов по значению атрибута name

HTML-атрибут name первоначально предназначался для присваивания имен элементам форм, и значение этого атрибута использовалось, когда выполнялась отправка данных формы на сервер. Подобно атрибуту id, атрибут name присваивает имя элементу. Однако, в отличие от id, значение атрибута name не обязано быть уникальным: одно и то же имя могут иметь сразу несколько элементов, что вполне обычно при использовании в формах радиокнопок и флажков. Кроме того, в отличие от id, атрибут name допускается указывать лишь в некоторых HTML-элементах, включая формы, элементы форм и элементы <iframe> и <img>.

Выбрать HTML-элементы, опираясь на значения их атрибутов name, можно с помощью метода **getElementsByName()** объекта Document:

```
var radiobuttons = document.getElementsByName("favorite_color");
```

Метод getElementsByName() определяется не классом Document, а классом HTMLDocument, поэтому он доступен только в HTML-документах и не доступен в XML-документах. Он возвращает объект **NodeList**, который ведет себя, как доступный только для чтения массив объектов Element.

В IE метод getElementsByName() возвращает также элементы, значения атрибутов id которых совпадает с указанным значением. Чтобы обеспечить совместимость с разными версиями браузеров, необходимо внимательно подходить к выбору значений атрибутов и не использовать одни и те же строки в качестве значений атрибутов name и id.

Выбор элементов по типу

Метод **getElementsByTagName()** объекта Document позволяет выбрать все HTML или XML-элементы указанного типа (или по имени тега). Например, получить подобный массиву объект, доступный только для чтения, содержащий объекты Element всех элементов в документе, можно следующим образом:

```
var spans = document.getElementsByTagName("span");
```

Подобно методу getElementsByName(), getElementsByTagName() возвращает объект NodeList. Элементы документа включаются в массив NodeList в том же порядке, в каком они следуют в документе, т.е. первый элемент <p> в документе можно выбрать так:

```
var firstParagraph = document.getElementsByTagName("p")[0];
```

Имена HTML-тегов не чувствительны к регистру символов, и когда getElementsByTagName() применяется к HTML-документу, он выполняет сравнение с именем тега без учета регистра символов. Переменная spans, созданная выше, например, будет включать также все элементы , которые записаны как <SPAN>.

Можно получить NodeList, содержащий все элементы документа, если передать методу getElementsByTagName() шаблонный символ «*».

Кроме того, классом Element также определяет метод getElementsByTagName(). Он действует точно так же, как и версия метода в классе Document, но выбирает только элементы, являющиеся потомками для элемента, относительно которого вызывается метод. То есть отыскать все элементы внутри первого элемента <p> можно следующим образом:

```
var firstParagraph = document.getElementsByTagName("p")[0];
var firstParagraphSpans = firstParagraph.getElenentsByTagName("span");
```

По историческим причинам класс HTMLDocument определяет специальные свойства для доступа к узлам определенных типов. Свойства *images*, *forms* и *links*, например, ссылаются на объекты, которые ведут себя как массивы, доступные только для чтения, содержащие элементы <img>, <form> и <a> (но только те теги <a>, которые имеют атрибут href). Эти свойства ссылаются на объекты HTMLCollection, которые во многом похожи на объекты NodeList, но дополнительно могут индексироваться значениями атрибутов id и name.

Объект HTMLDocument также определяет свойства-синонимы **embeds** и **plugins**, являющиеся коллекциями HTMLCollection элементов <embed>. Свойство **anchors** является нестандартным, но с его помощью можно получить доступ к элементам <a>, имеющим атрибут name, но не имеющим атрибут href. Свойство **scripts** определено стандартом HTML5 и является коллекцией HTMLCollection элементов <script>.

Кроме того, объект HTMLDocument определяет два свойства, каждое из которых ссылается не на коллекцию, а на единственный элемент. Свойство **document.body** представляет элемент <body> HTML-документа, а свойство **document.head** - элемент <head>. Эти свойства всегда определены в документе: даже если в исходном документе отсутствуют элементы <head> и <body>, браузер создаст их неявно. Свойство **documentElement** объекта Document ссылается на корневой элемент документа. В HTML-документах он всегда представляет элемент <html>.

Выбор элементов по классу CSS

Значением HTML-атрибута class является список из нуля или более идентификаторов, разделенных пробелами. Он дает возможность определять множества связанных элементов документа: любые элементы, имеющие в атрибуте class один и тот же идентификатор, являются частью одного множества. Слово class зарезервировано в языке JavaScript, поэтому для хранения значения HTML-атрибута class в клиентском JavaScript используется свойство className.

Обычно атрибут class используется вместе с каскадными таблицами стилей CSS, с целью применить общий стиль отображения ко всем членам множества. Однако кроме этого, стандарт HTML5 определяет метод **getElementsByClassName()**, позволяющий выбирать множества элементов документа на основе идентификаторов в их атрибутах class.

Подобно методу getElementsByTagName(), метод getElementsByClassName() может вызываться и для HTML-документов, и для HTML-элементов, и возвращает «живой» объект NodeList, содержащий все потомки документа или элемента, соответствующие критерию поиска.

Метод getElementsByClassName() принимает единственный строковый аргумент, но в самой строке может быть указано несколько идентификаторов, разделенных пробелами. Соответствующими будут считаться все элементы, атрибуты class которых

содержат все указанные идентификаторы. Порядок следования идентификаторов не имеет значения. Обратите внимание, что и в атрибуте class, и в аргументе метода getElementsByClassName() идентификаторы классов разделяются пробелами, а не запятыми.

Ниже приводится несколько примеров использования метода getElementsByClassName():

```
// Отыскать все элементы с классом "warning"
var warnings = document.getElementsByClassName("warning");

// Отыскать всех потомков элемента с идентификатором "log"
// с классами "error" и "fatal"
var log = document.getElementById("log");
var fatal = log.getElementsByClassName("fatal error");
```

Выбор элементов с использованием селекторов CSS

Каскадные таблицы стилей CSS имеют очень мощные синтаксические конструкции, известные как селекторы, позволяющие описывать элементы или множества элементов документа. Наряду со стандартизацией селекторов CSS3, другой стандарт консорциума W3C, известный как **Selectors API**, определяет методы JavaScript для получения элементов, соответствующих указанному селектору.

Ключевым в этом API является метод **querySelectorAll()** объекта Document. Он принимает единственный строковый аргумент с селектором CSS и возвращает объект NodeList, представляющий все элементы документа, соответствующие селектору.

В дополнение к методу querySelectorAll() объект документа также определяет метод **querySelector()**, подобный методу querySelectorAll(), - с тем отличием, что он возвращает только первый (в порядке следования в документе) соответствующий элемент или null, в случае отсутствия соответствующих элементов.

Эти два метода также определяются классом Elements. Когда они вызываются относительно элемента, поиск соответствия заданному селектору выполняется во всем документе, а затем результат фильтруется так, чтобы в нем остались только потомки использованного элемента. Такой подход может показаться противоречащим здравому смыслу, так как он означает, что строка селектора может включать предков элемента, для которого выполняется сопоставление.

Структура документа и навигация по документу

После выбора элемента документа иногда бывает необходимо отыскать структурно связанные части документа (родитель, братья, дочерний элемент). Объект Document можно представить как дерево объектов Node. Тип Node определяет свойства, позволяющие перемещаться по такому дереву. Существует еще один прикладной интерфейс навигации по документу, как дерева объектов Element.

Документы как деревья узлов

Объект Document, его объекты Element и объекты Text, представляющие текстовые фрагменты в документе - все они являются объектами Node. Класс Node определяет следующие важные свойства:

parentNode

Родительский узел данного узла или null для узлов, не имеющих родителя, таких как Document.

childNodes

Доступный для чтения объект, подобный массиву (NodeList), обеспечивающий представление дочерних узлов.

firstChild, lastChild

Первый и последний дочерние узлы или null, если данный узел не имеет дочерних узлов.

nextSibling, previousSibling

Следующий и предыдущий братские узлы. Братскими называются два узла, имеющие одного и того же родителя. Порядок их следования соответствует порядку следования в документе. Эти свойства связывают узлы в двусвязный список.

nodeType

Тип данного узла. Узлы типа Document имеют значение 9 в этом свойстве. Узлы типа Element - значение 1. Текстовые узлы типа Text - значение 3. Узлы типа Comments - значение 8 и узлы типа DocumentFragment - значение 11.

nodeValue

Текстовое содержимое узлов Text и Comment.

nodeName

Имя тега элемента Element, в котором все символы преобразованы в верхний регистр.

С помощью этих свойств класса Node можно сослаться на второй дочерний узел первого дочернего узла объекта Document, как показано ниже:

```
document.childNodes[0].childNodes[1] ==
    document.firstChild.firstChild.nextSibling
```

Допустим, что рассматриваемый документ имеет следующий вид:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Тогда вторым дочерним узлом первого дочернего узла будет элемент <body>. В свойстве nodeType он содержит значение 1 и в свойстве nodeName - значение «BODY».

Однако, обратите внимание, что этот прикладной интерфейс чрезвычайно чувствителен к изменениям в тексте документа. Например, если в этот документ добавить единственный перевод строки между тегами <html> и <head>, этот символ перевода строки станет первым дочерним узлом (текстовым узлом Text) первого дочернего узла, а вторым дочерним узлом станет элемент <head>, а не <body>.

Документы как деревья элементов

Когда основной интерес представляют сами элементы документа, а не текст в них (и пробельные символы между ними), гораздо удобнее использовать прикладной интерфейс, позволяющий интерпретировать документ как дерево объектов Element, игнорируя узлы Text и Comment, которые также являются частью документа.

Первой частью этого прикладного интерфейса является свойство **children** объектов Element. Подобно свойству childNodes, его значением является объект NodeList. Однако, в отличие от свойства childNodes, список children содержит только объекты Element.

Обратите внимание, что узлы Text и Comment не имеют дочерних узлов. Это означает, что описанное выше свойство Node.parentNode никогда не возвращает узлы типа Text или Comment. Значением свойства parentNode любого объекта Element всегда будет другой объект Element или корень дерева - объект Document или DocumentFragment.

Второй частью прикладного интерфейса навигации по элементам документа являются свойства объекта Element, аналогичные свойствам доступа к дочерним и братским узлам объекта Node:

firstElementChild, lastElementChild

Похожи на свойства firstChild и lastChild, но возвращают дочерние элементы.

nextElementSibling, previousElementSibling

Похожи на свойства nextSibling и previousSibling, но возвращают братские элементы.

childElementCount

Количество дочерних элементов. Возвращает то же значение, что и свойство children.length.

Эти свойства доступа к дочерним и братским элементам стандартизованы и реализованы во всех текущих браузерах, кроме IE.