JAVASCRIPT

ОБЩИЕ СВЕДЕНИЯ

Задачи JavaScript

- Динамическое добавление, редактирование и удаление HTML-элементов и их значений;
- Проверка содержимого web-форм перед отправкой на сервер;
- Создание на компьютере клиента cookie-файлов для сохранения и получения данных при последующих визитах.

Подключение JS

- Теги script и атрибут type необходимы для добавления JavaScript на HTML-страницу

  <script type="text/javascript">
  </script>

- Подключение внешнего JS-файла к HTML-странице

  <script type="text/javascript"
  src="path/to/javascript.js"> </script>

Переменные

- Локальные
  - объявляются с помощью ключевого слова *var* *(пример var num = 10;)*
  - доступны только в той области, где были объявлены
- Глобальные
  - объявляются без ключевого слова *var*
  - доступны по всему сценарию

Операторы

- Арифметические
- Присваивания
- Сравнения

  == Равенство

  === Равенство по значению и типу объекта
- Логические

Арифметические операторы

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Вычисление остатка от деления
++	Инкремент
--	Декремент

Операторы присваивания

Оператор	Описание
=	Равно
+=	Присвоить переменной результат сложения
num += 5;	*num = (num + 5);*
-=	Присвоить переменной результат вычитания
*=	Присвоить переменной результат умножения
/=	Присвоить переменной результат деления
%=	Присвоить переменной результат вычисления остатка от деления

Операторы сравнения

Оператор	Описание
==	Равенство
===	Равенство по значению и типу объекта
!=	Неравенство
>	Больше чем
<	Меньше чем
>=	Больше или равно
<=	Меньше или равно

Логические операторы

Оператор	Описание
&&	И
||	ИЛИ
!	НЕ

True
False

- True && True = true
- True && False= false
- False && True = false
- False && False= false

- True || True = true
- True || False= true
- False || True = true
- False || False= false

- 1 && 0 || 1 = 1
- (0 && 1) || 0 = 0 && (1 || 0)= 0

Массивы

- Массивы похожи на переменные, но отличаются от них тем, что могут хранить несколько значений и выражений под одним именем.
- Возможность хранения нескольких значений в одной переменой – это главное преимущество массива.
- В JavaScript для массивов не существует ограничений на количество или тип данных, которые будут в нем храниться, пока эти данные находятся в области видимости массива.
- Доступ к значению любого элемента массива можно получить в любой момент времени после объявления массива в сценарии.

Хранение однородных значений в массиве

- *var colors = new Array("orange", "blue", "red", "brown");*
- *var shapes = new Array("circle", "square", "triangle", "pentagon");*
- Массив всегда начинается с 0-го, а не первого элемента, что поначалу может смущать.
- Нумерация элементов начинается с 0, 1, 2, 3 и т.д.
- Для доступа к элементу массива необходимо использовать его идентификатор, соответствующий позиции элемента в массиве.
- *document.write("Orange: "+ colors[0]);*

Присваивание значений элементам

```
var colors = new Array();
colors[0] = "orange";
colors[1] = "blue";
colors[2] = "red";
colors[3] = "brown";
document.write("Blue: "+ colors[1]);
//изменить значение blue на purple
colors[1] = "purple";
document.write("Purple: "+ colors[1]);
```

Условные выражения

- *if* Используется для выполнения сценария, если определенное условие истинно (равно true).
- *if...else* Используется для выполнения одного сценария, если условие истинно, или другого сценария, если данное условие ложно (равно false).
- *if...else*

  if...else
- Используется для выполнения одного сценария, если одно из множества условий истинно, или другого сценария, когда все условия ложны.
- *switch* Используется для выполнения одного из нескольких сценариев.

```
var num = 10;
if(num == 5)
{
document.write("num равно 5");
}
```

```
var num = 10;
if(num == 5)
{
document.write("num равно 5");
}
else
{
document.write("num НЕ равно 5, num равно: "+
num);
}
```

```
var num = 10;
if(num == 5)
{
document.write("num равно 5");
}
else if(num == 10)
{
document.write("num равно 10");
}
else
{
document.write("num равно: "+ num);
}
```

```
var num = 10;
switch(num)
{
case 5:
        document.write("num равно 5");
        break;
case 10:
        document.write("num равно 10");
        break;
default:
        document.write("num равно: "+ num);
}
```

- Выражение break прерывает или останавливает выполнение последующих выражений в конструкции switch.

```
var num = 10;
switch(num)
{
case 5:
document.write("num равно 5");
break;
case 10:
document.write("num равно 10");
default:
document.write("num равно: "+ num);
}
```

Циклы

- Циклы for и while предоставляют средства для итерирования по этим массивам, доступа к их значениям и использования их в сценариях.

```
var i = 0;
while(i<10)
{
document.write(i +"<br/>");
i++;
}
```

```
var colors = new Array("orange", "blue", "red",
                                        "brown");
for(var i=0; i<colors.length; i++)
{
document.write("цвет: "+ colors[i] +"<br/>");
}
```

Создание всплывающих окон

- Окно для вывода информации

  alert("здесь может быть переменная или строка");

- Окно для подтверждения действий

  if(confirm("нажмите для продолжения работы"))

  { alert('вы нажали ОК'); }

  else{ alert('вы нажали Отмена'); }

- Окно для ввода информация (prompt)

Функции в JavaScript. Базовые понятия.

- Предназначение функции – хранить код для выполнения определенной задачи, чтобы его можно было вызвать в любое время.
- Функция оформляется в коде очень просто: она начинается с ключевого слова function, за которым следует пробел и название функции.

```
var num = 10;
function changeVariableValue()
{
num = 11;
}
changeVariableValue();
document.write("num равно: "+ num);
```

- В представленном фрагменте можно изменить значение переменной, так как она объявлена в области видимости основного сценария, как и сама функция, поэтому функция знает о существовании переменной.
- Но если объявить переменную внутри функции, то к ней нельзя будет получить доступ за пределами функции.

- Функции также могут принимать данные через входные параметры.
- Функция может иметь один или несколько параметров, и в вызове функции будет передаваться столько параметров, сколько объявлено в сигнатуре функции.
- Важно не путать понятие "параметр" с "аргументом".
- Параметр – это часть определения (или сигнатуры) функции, а аргумент – это выражение, используемое при вызове функции.

```
var num, num2;
num=num2=10;
function incDec(x, y)
{
    x++;
    y--;
}
incDec(num, num2);
document.write("num равно: "+ num);
```

- В данном случае аргумент – это ранее объявленная переменная.

- В функциях также используются выражения *return*. Эти выражения возвращают значение, полученное после выполнения сценария в функции.
- Например, можно присвоить переменной значение, возвращенное функцией.

```
function add(_num1, _num2)
{
return _num1+_num2;
}
var num = add(10, 10);
document.write("num равно: "+ num);
```

Особенности функций в JavaScript

- В JavaScript функция является значением.
- Объявление создает функцию и присваивает в переменную.
- Как и любое значение, функцию например можно вывести

```
function sayHi() {
 alert('Привет'); }
alert(sayHi); // выведет код функции
```

- Функцию также можно скопировать. Копируется не сама функция, а ссылка на неё.

Объявление Function Declaration

function func(параметры) { код }

func – название переменной, в которую будет помещена функция (она может быть затем удалена, скопирована в другую).

- Функции, объявленные как Function Declaration, создаются в момент входа в область видимости (можно вызывать до объявления).
- Условно объявить функцию через Function Declaration нельзя.

Объявление Function Expression

var f = function(параметры) { код }

- Создается «анонимная функция», которая затем присваивается переменной.
- Function Expression создают функцию, когда до них доходит выполнение.
- Поэтому и пользоваться ими можно только после объявления.
- Function Expression можно использовать для условного объявления функции.

Вызов «на месте»

- Можно создать функцию при помощи Function Expression и не присваивать ее переменной, а тут же вызвать:

```
(function() {
 var a, b; // локальные переменные
// код
})();
var res = function(a,b) { return a+b }(2,2);
alert(res); // 4
```

Именованные функциональные выражения

- Обычно у функций в JavaScript нет имени. Функция является всего лишь значением, которое присваивается переменной.
- Но в JavaScript возможность указать имя, действительно привязанное к функции.
- Она называется Named Function Expression (NFE) или именованное функциональное выражение.

```
var f = function sayHi(...) { /*тело функции*/ };
```

- Имя функционального выражения имеет особый смысл.
- Оно доступно только изнутри самой функции.
- Цель — позволить функции вызвать саму себя.
- Оно работает всегда, даже если переменная, в которой изначально содержалась функция, была перезаписана, а сама функция — перемещена в другое место.

Аргументы функций

- В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.
- Отсутствующие аргументы становятся undefined.
- В JavaScript нет «полиморфизма функций» или «перегрузок функций».
- Второе объявление функции просто переопределит первое.

Доступ к «лишним» аргументам

- Доступ к значения аргументов, которых нет в списке параметров осуществляется через «псевдо-массив» arguments (не массив Array, обычный Object).
- Он содержит список аргументов по номерам: arguments[0], arguments[1]…, а также свойство length.

```
function sayHi() {
for (var i=0; i<arguments.length; i++) {
alert("Привет, " + arguments[i]); }}
```

Значения по умолчанию

- Значения по умолчанию нужны там, где аргумент может отсутствовать. Такие аргументы желательно располагать после обязательных.
- Например, функция показа сообщения showMessage может быть вызвана как

  showMessage(text, title) или *showMessage(text)*.
- Если title не передан, то по умолчанию title = "Сообщение".

- Первый способ указать значение по умолчанию для title — явно проверить на undefined и переназначить:

```
function showMessage(text, title) {
if (title === undefined) title = 'Сообщение';
                ...     }
```

- Второй — использовать оператор ||:

```
function showMessage(text, title) {
title = title || 'Сообщение';
                ...     }
```

Преобразование типов

В JavaScript есть три преобразования, которые зависят от контекста:

- Строковое: происходит обычно при выводе объекта, использует toString.
- Численное: его делают математические операторы и функции, а также сравнения и проверки равенства, кроме строгих === и !==. Использует valueOf, если есть, а если нет — toString.
- Логическое. Неявное преобразование в логический тип данных происходит при использовании любого типа данных в условиях.

Автопреобразование типов не осуществляется в следующих случаях:

- При сравнении объектов. Объекты равны только когда это один и тот же объект.
- При сравнении двух строк (отдельный алгоритм сравнения).
- При проверке равенства с null и undefined (они равны друг другу, но не равны чему бы то ни было ещё).

Явные преобразования

- **Преобразование к числу**

  +value или Number(value)

- **Преобразование к строке**

  '' + value (если value — объект,
  кроме Date, то сработает valueOf) или
  String(value)

- **Преобразование к логическому значению**

  !!value или Boolean(value)

Результаты преобразования к значению типа String

Тип	Конвертируется в строку
Неопределенный (Undefined)	"undefined"
Пустой (Null)	"null"
Логический (Boolean)	"true", если это true, " f a 1 se ", если это false
Числовой (Number)	"NaN", " 0 " или строка, представляющая соответствующее числовое значение
Другие объекты	Значение, возвращаемое объектом с помощью метода toStr ing (), если таковой имеется, иначе "undefined"

Результаты преобразования к значению типа Number

Тип	Конвертируется в числовое значение
Неопределенный (Undefined)	NaN
Пустой (Null)	0
Логический (Boolean)	1,если это true, 0, если это false
Строковый (String)	Соответствующее числовое значение, если строка похожа на число, иначе NaN
Другие объекты	NaN

Результаты преобразования к значению типа Boolean

Тип	**Конвертируется в булево значение**
Неопределенный (Undefined)	False
Пустой (Null)	False
Числовой (Number)	False, если это 0 или NaN, иначе true
Строковый (String)	False, если это длина строки равна 0, иначе true
Другие объекты	True

Преобразование объектов

- Для преобразования объекта к строке используется метод toString(), для преобразования объекта в число нужно переопределить метод valueOf().

```
var student = {
Name : Alex,
Age : 18,
toString: function() {
  return "Студент "+ this.Name;
 },
 valueOf: function() {
 return this.Age;
}
}
```

- Результат toString() может иметь любой тип, кроме объекта. То есть toString() не обязан возвращать именно строку.
- Метод valueOf() обязан возвращать примитивное значение, иначе его результат будет проигнорирован. При этом — не обязательно числовое.
- У большинства встроенных объектов такого valueOf() нет, поэтому численное и строковое преобразования для них работают одинаково, так ка вызывается toString().
- Исключением является объект Date, который поддерживает оба типа преобразований

Использование cookie-файлов

- Cookie-файлы используются для хранения данных на стороне клиента
- Cookie – это текстовый файл, хранящийся в Web-браузере посетителя, состоящий из:
  - пары имя–значение
  - срока годности
  - имени домена
  - пути на сервере

- Создание cookie
  - *document.cookie = 'cookiename=cookievalue; expires=Sat, 3 Nov 2001 12:30:10 UTC; path=/'*
- Извлечение данных из cookie
  - document.cookie – это массив
  - определенный cookie необходимо правильно выбрать

Функция принимает имя cookie и возвращает его значение.

```
function getCookie(c_name) {
var i,x,y;
var cookieArray = document.cookie.split(";");
for (i=0;i<cookieArray.length;i++)    {
  x = cookieArray[i].substr(0,cookieArray[i].indexOf("="));
  y = cookieArray[i].substr(cookieArray[i].indexOf("=")+1);
  x = x.replace(/^\s+|\s+$/g,"");
  if(x == c_name)
        { return unescape(y);  }
} }
alert(getCookie('cookiename'));
```

Работа с таймерами

- setInterval – для многократного выполнения сценария без участия пользователя
  - *setInterval(code, milliseconds, argument)*
- clearInterval – для остановки выполнения
  - *clearInterval(myInterval)*

  Однако при запуске серии необходимо указать имя переменной, чтобы потом использовать её в качестве параметра при вызове clearInterval.

```
var myInterval = setInterval(myFunction,
10000, 'sample');
function myFunction(myArg)
{   alert('аргумент: '+ myArg);
clearInterval(myInterval); }
```

- setTimeout
  - Основное отличие setTimeout – функция выполняется только один раз, а не серийно.
- clearTimeout
  - Для отмены срабатывание функции setTimeout
- clearTimeout(myTimeout)

Document Object Model

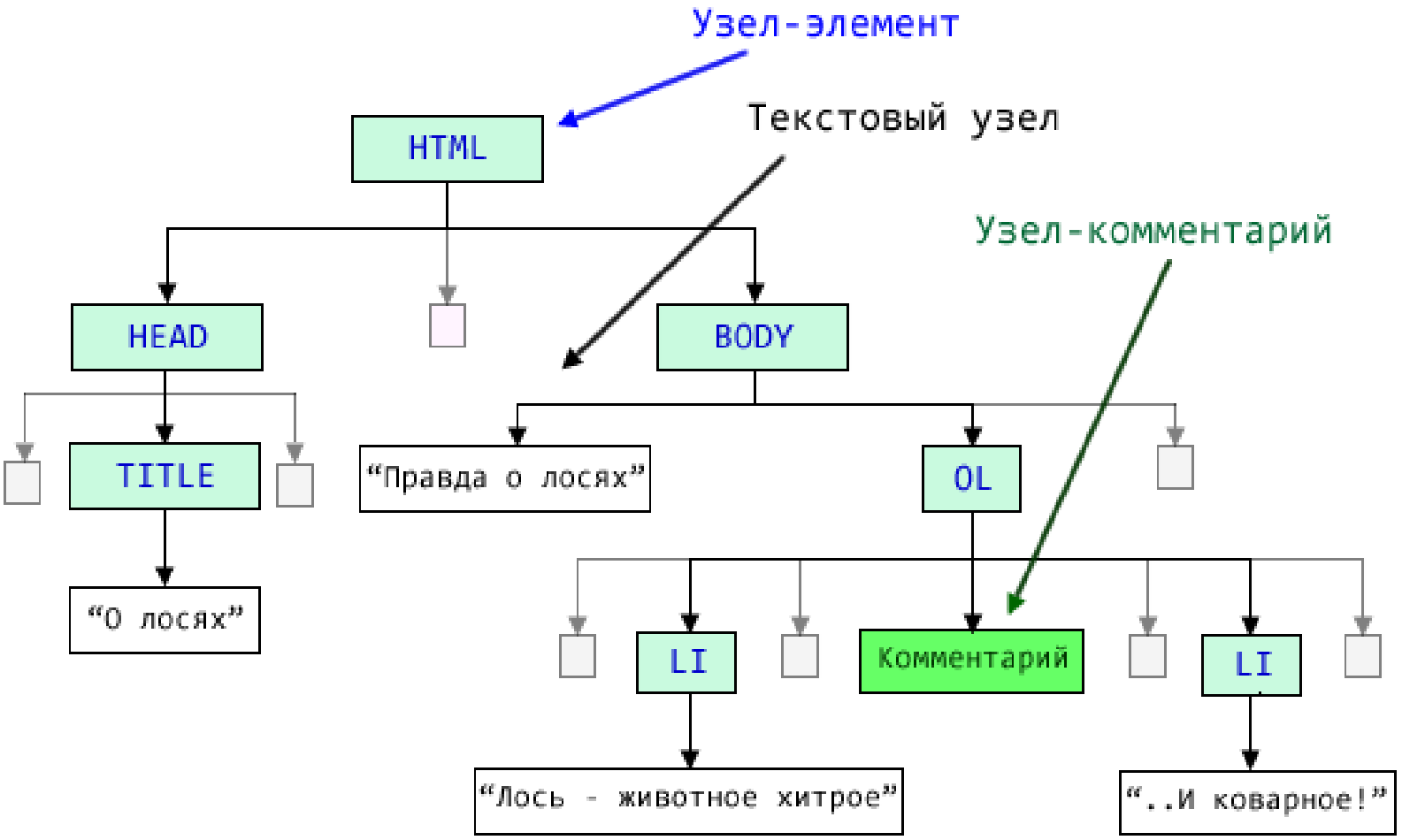
Дерево DOM

- Согласно DOM-модели, документ является иерархией, деревом.
- Каждый HTML-тег образует узел дерева с типом «элемент».
- Вложенные в него теги становятся дочерними узлами.
- Для представления текста создаются узлы с типом «текст». Текстовые узлы не могут иметь потомков.
- Узлы-комментарии

Пример. HTML код.

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

Пример. DOM представление.



Узлы DOM

- В этом дереве выделено три типа узлов.

1. Теги образуют узлы-элементы (element node) DOM-дерева. Естественным образом одни узлы вложены в другие.
2. Текст представлен текстовыми узлами. Текстовые узлы не могут иметь потомков.
3. Узел-комментарий. На отображение-то он не влияет. Но так как он есть в HTML — обязан присутствовать в DOM-дереве. В инструментах они не отображаются.

Навигация в DOM, свойства-ссылки

- Войти в «корень» дерева можно двумя путями.
  - document.documentElement. Это свойство ссылается на DOM-объект для тега HTML.
  - document.body, который соответствует тегу BODY (может быть равен null).
- Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.
- Для свойств-ссылок на узлы в качестве значения «нет такого элемента» или «узел не найден» используется не undefined, а null.

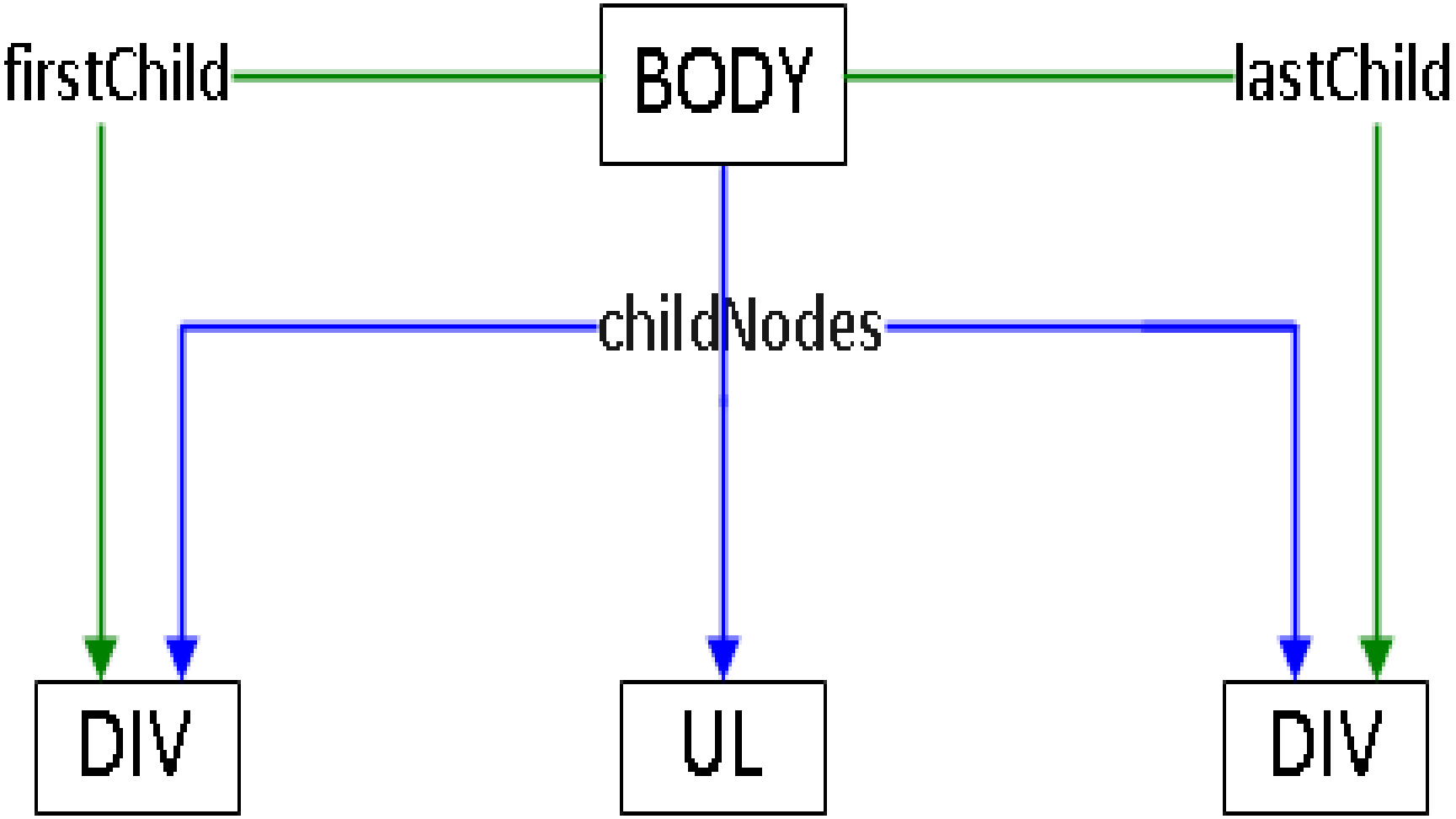
Дочерние элементы

- Из узла-родителя можно получить все дочерние элементы. Для этого есть несколько способов.
- Псевдо-массив childNodes хранит все дочерние элементы, включая текстовые.
- Свойство children перечисляет только дочерние узлы, соответствующие тегам.

Ссылки вверх и вниз

- Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему потомку. Они соответствуют индексам childNodes:
  - *body.firstChild === body.childNodes[0]*
  - *body.lastChild === body.childNodes[body.childNodes.length-1]*
- firstElementChild

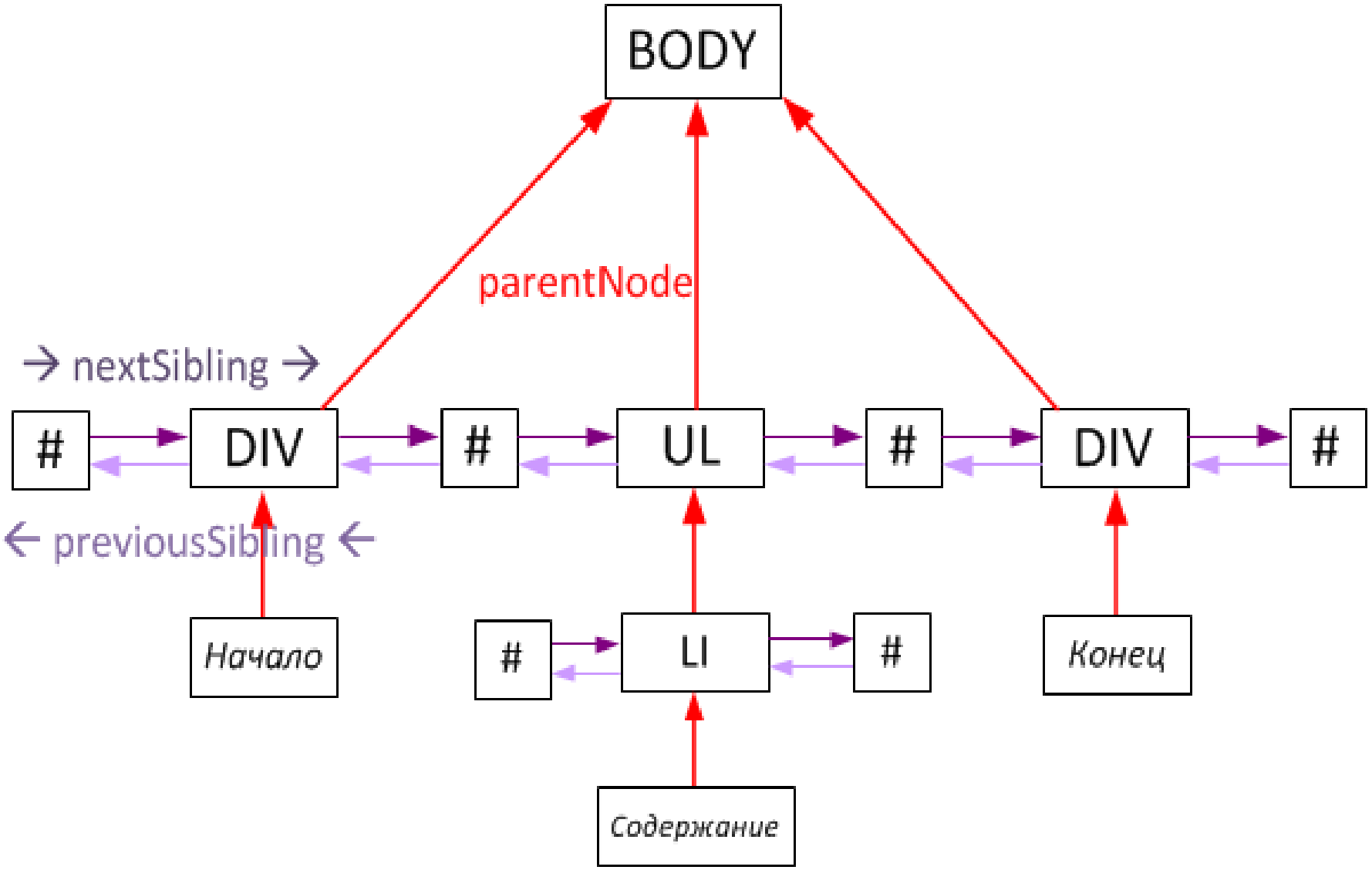
Ссылки вниз



Ссылки вверх и вниз

- Свойство parentNode ссылается на родительский узел. Оно равно null для корневого элемента document.documentElement.
- Свойства previousSibling и nextSibling дают доступ к левому и правому соседу.
- Все навигационные ссылки — только для чтения. При изменениях DOM, добавлении или удалении элементов они обновляются автоматически.

Ссылки вниз и на соседей



Дополнительные ссылки

- Дополнительные ссылки для элементов (кроме IE<9)
  - childElementCount — число детей-элементов (=children.length)
  - firstElementChild — первый потомок-элемент (=children[0])
  - lastElementChild — последний потомок-элемент (=children[children.length-1])
  - nextElementSibling — правый брат-элемент
  - previousElementSibling — левый брат-элемент

Основные свойства: тип, тег

- Тип узла содержится в его свойстве nodeType. (Их 12.) Наиболее важные — это ELEMENT_NODE под номером 1 и TEXT_NODE под номером 3.
- Два свойства: nodeName и tagName содержат название (тег) элемента узла. Название HTML-тега всегда находится в верхнем регистре.

Разница между tagName и nodeName

- Свойство nodeName определено для многих типов DOM-узлов.
- Свойство tagName — есть только у элементов (в IE<9 также у комментариев, это ошибка в браузере).
- То есть при помощи tagName мы можем работать только с элементами, а nodeName может что-то сказать и о других типах узлов

Свойство innerHTML

- Свойство innerHTML позволяет получить HTML-содержимое узла в виде строки. В innerHTML можно и читать и писать.
- Добавление innerHTML+= осуществляет перезапись.
- Однако фактически *добавления* не происходит:

1. Удаляется старое содержание
2. На его место становится новое значение innerHTML.

Так как новое значение записывается с нуля, то все изображения и другие ресурсы будут перезагружены.

- Скрипты не выполняются.

- Текстовые узлы и комментарии, имеют содержимое, которое можно читать и обновлять через два свойства: nodeValue и data.
- Узлы DOM также имеют другие свойства, в зависимости от тега. Например, у INPUT есть свойства value и checked, а у A есть href и т.д.

Пользовательские свойства

- Узел DOM – это объект. Как и любой объект JavaScript, он может содержать пользовательские свойства и методы.
- Они видны только в JavaScript и никак не влияют на отображение тега.
- Пользовательские DOM-свойства:
  - Могут иметь любое значение.
  - Названия свойств чувствительны к регистру.
  - Работают за счет того, что DOM-узлы являются объектами JavaScript

Атрибуты

- Узлы DOM являются HTML-элементами, у которых есть атрибуты.
- Доступ к атрибутам осуществляется при помощи стандартных методов:
  - elem.hasAttribute(name) - проверяет наличие
  - elem.getAttribute(name) - получает значение
  - elem.setAttribute(name, value) - устанавливает
  - elem.removeAttribute(name) - удаляет атрибут
- В отличие от свойств, атрибуты:
  - Могут быть только строками.
  - Их имя нечувствительно к регистру (т.к. это HTML)
  - Они отражены в HTML, включая свойство innerHTML (за исключением старых IE)

Синхронизация свойств и атрибутов

- Каждый тип узлов DOM имеет свои стандартные свойства.
- Стандартные свойства DOM синхронизируются с атрибутами.
- Id. Браузер синхронизирует атрибут "id" со свойством id.
- href. Синхронизация не гарантирует одинакового значения.
- value. Синхронизируется в одну сторону.
- Атрибут "class" соответствует свойству className ("class" зарезервированное словом в Javascript).

Создание и клонирование узлов

- Создание элементов: createElement
  - *document.createElement(tag)* – Создает новый элемент с указанным тегом.
  - *document.createTextNode(text*) – Создает новый текстовый узел с данным текстом.
- Новому элементу тут же можно поставить свойства.
- Новый элемент можно также склонировать из существующего (вместе с атрибутами):
  - *newElem = elem.cloneNode(true)* – С вложенными в него подэлементами.
  - *newElem = elem.cloneNode(false)* – Без подэлементов.

Добавление элемента

- Чтобы DOM-узел был показан на странице, его необходимо вставить в документ.
- Добавление элемента: appendChild, insertBefore.
  - *parentElem.appendChild(elem)* – Добавляет elem в конец списка дочерних элементов parentElem.
  - *parentElem.insertBefore(elem, nextSibling)* – Вставляет elem в список дочерних parentElem, перед элементом nextSibling.
- Все методы вставки возвращают вставленный узел

Удаление узлов

- Для удаления узла есть два метода:
  - *parentElem.removeChild(elem)* – Удаляет elem из списка детей parentElem.
  - *parentElem.replaceChild(elem, currentElem)* – Среди детей parentElem заменяет currentElem на elem.
- Эти метода возвращают удаленный узел.
- Если надо *переместить* элемент на новое место — не нужно его удалять со старого. Методы appendChild/insertBefore автоматически удаляют вставляемый элемент со старого места.

Классы в виде строки: className

- Атрибуту "class" соответствует свойство className.
- Например:

```
<body class="main page">
 <script> // прочитать класс элемента
alert( document.body.className ); // main page
// поменять класс элемента
document.body.className = "class1 class2";
</script>
</body>
```

Классы в виде объекта: classList

- Работать с классами как со строкой неудобно.
- **Свойство classList – это объект для работы с классами.**
- Методы *classList: elem.classList.contains("class")* – возвращает true/false, в зависимости от того, есть ли у элемента класс class.
- *elem.classList.add/remove("class")* – добавляет/удаляет класс class
- *elem.classList.toggle("class")* – если класса class нет, добавляет его, если есть – удаляет.
- Кроме того, можно перебрать классы через for, так как classList – это псевдо-массив.

```
<body class="main page">
 <script>
var classList = document.body.classList;
classList.remove('page'); // удалить класс
classList.add('post'); // добавить класс
for (var i = 0; i < classList.length; i++)
{ // перечислить классы
alert( classList[i] ); // main, затем post
}
alert( classList.contains('post') ); // проверить наличие класса
alert( document.body.className ); // main post, тоже работает
</script>
</body>
```

Мультивставка: "insertAdjacentHTML" и "DocumentFragment"

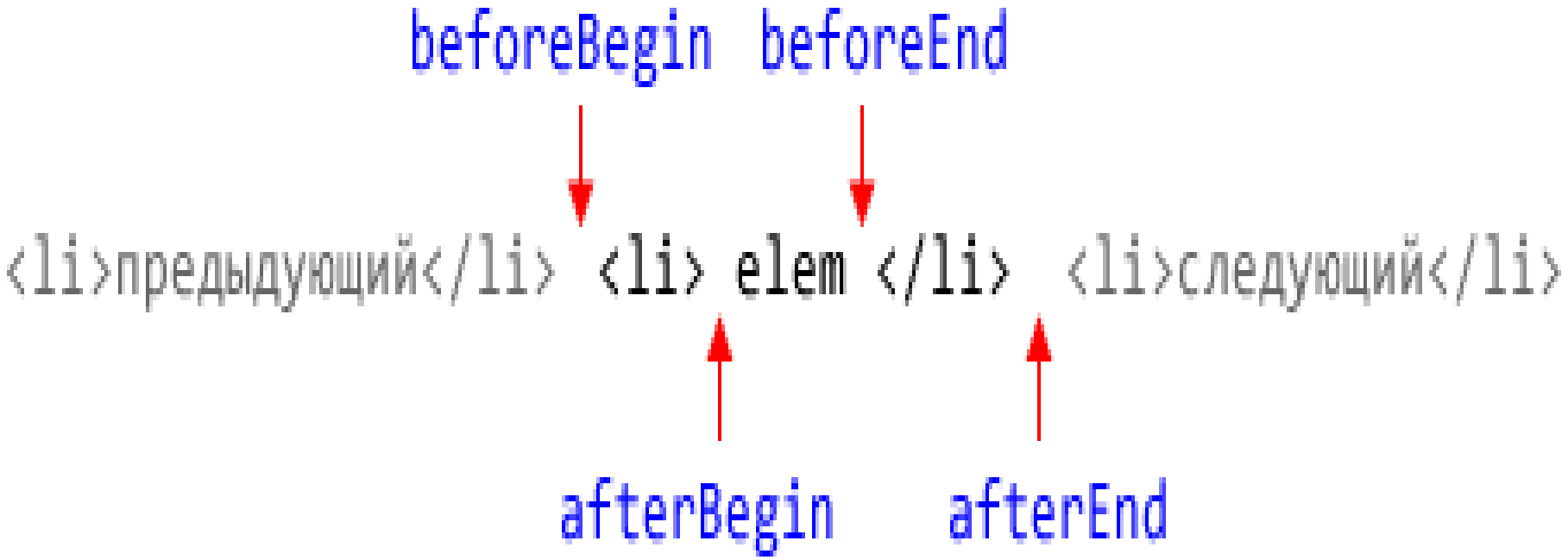
- Обычные методы вставки работают с одним узлом. Но есть и способы вставлять множество узлов одновременно.
- Есть две возможных последовательности (на примере генерации списка UL/LI):
  - Сначала вставить UL в документ, а потом добавить к нему LI.
  - Полностью создать список «вне DOM», а потом — вставить в документ (этот способ быстрее).

insertAdjacentHTML

elem.insertAdjacentHTML(where, html);

- *html* — Строка HTML, которую нужно вставить.
- *where* — Куда по отношению к elem вставлять строку. Всего четыре варианта:
  - *beforeBegin* — перед elem.
  - *afterBegin* — внутрь elem, в самое начало.
  - *beforeEnd* — внутрь elem, в конец.
  - *afterEnd* — после elem.

insertAdjacentHTML



DocumentFragment

- DocumentFragment – особенный кросс-браузерный DOM-объект.
- Синтаксис для его создания:

var fr = document.createDocumentFragment();

- В него можно добавлять другие узлы:

fr.appendChild(node);

- Его можно клонировать:

fr.cloneNode(true); // клонирование с подэлементами

DocumentFragment

- Но у DocumentFragment нет обычных свойств DOM-узлов, таких как innerHTML, tagName и т.п. Это не узел.
- Особенность заключается в том, что когда DocumentFragment вставляется в DOM — то он исчезает, а вместо него вставляются его дети. Это свойство является уникальной особенностью DocumentFragment.

DocumentFragment. Пример

```
var fragment =
document.createDocumentFragment();
for (цикл по li) {
 fragment.appendChild(list[i]); //
вставить каждый LI в DocumentFragment
}
ul.appendChild(fragment);   // одна
операция над живым документом
```

document.write

- Метод document.write — один из наиболее древних методов добавления текста к документу. У него есть существенные ограничения, поэтому он используется редко, но бывает полезен
- Метод *document.write(str)* корректно работает только пока HTML еще не догружен.
- Нет никаких ограничений на содержимое document.write.
- Строка просто пишется в документ, без проверки структуры тегов, как будто она всегда там была.

Поиск элементов в DOM

- Методы поиска элементов
  - document.getElementById(id)
  - elem.getElementsByTagName(tag)
  - elem.getElementsByClassName(className)
  - document.getElementsByName(name)
  - elem.querySelectorAll(css)
  - elem.querySelector(css)
  - elem.matchesSelector(css)

Результаты

- Все DOM-запросы, которые начинаются с getElements.., возвращают не массив, а коллекции, имеющие тип NodeList или HTMLCollection.
- У коллекции есть индексы, длина, но нет push, pop и других свойств массива.
- При обращении к элементам такой коллекции, поиск выполняется каждый раз заново.

События

ОСНОВЫ

Введение в браузерные события

- Виды событий:
- DOM-события, которые инициализируются элементами DOM.
- События для окна браузера.
- Загрузки файла/документа

Назначение обработчиков событий

- Использование атрибута HTML

  <input id="b1" value="Нажми меня" onclick="alert('Спасибо!')" type="button"/>

- Использование свойства DOM-объекта

  elem.onclick = function() {}

- Фундаментальный недостаток описанных способов назначения обработчика — невозможность повесить несколько обработчиков на одно событие.

Специальные методы. (IE<9)

- Назначение обработчика осуществляется вызовом attachEvent:
  - *element.attachEvent("on"+event, handler);*
- Удаление обработчика — вызовом detachEvent:
  - *element.detachEvent("on"+event, handler);*
- Обработчики, назначенные с attachEvent не получают this!

Назначение обработчиков по стандарту

- Назначение обработчика:
  - *element.addEventListener(event, handler, phase);*
- Удаление:
  - *element.removeEventListener(event, handler, phase);*
- Особенности специальных методов:
  - Можно поставить столько обработчиков, сколько вам нужно.
  - Нельзя получить назначенные обработчики из элемента.
  - Браузер не гарантирует сохранение порядка выполнения обработчиков.
  - Кроссбраузерные несовместимости.

Получение объекта события

- Когда обработчик запускается, браузер создает «объект события» с информацией о том, что и где произошло.
- В обработчике его можно получить и прочитать детали произошедшего из его свойств.
- Разные типы событий, предоставляют разные свойства.

Объект события onclick

- event.type = "click" — тип события
- event.target — элемент, по которому кликнули. В IE вместо него используется свойство event.srcElement.
- event.clientX / event.clientY - координаты курсора в момент клика.
- … Также есть информация о том, какой кнопкой был произведен клик, и другие свойства.

Путь W3C. Путь IE<9

- Браузеры, которые следуют стандартам W3C всегда передают объект события первым аргументом в обработчик.
  - element.onclick = function(event) {}
  - function doSomething(event) {}
    element.onclick = doSomething
- IE<9 создает глобальный объект window.event, который ссылается на последнее событие.
  - element.onclick = function() {
    // window.event - объект события }

Кроссбраузерное решение

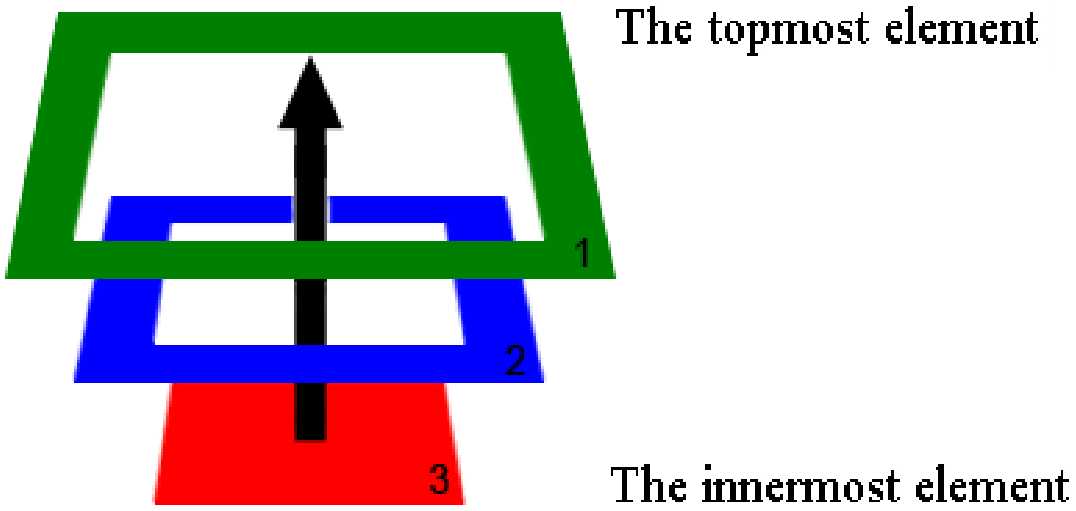
- Универсальное решение для получения объекта события:

```
element.onclick = function(event) {
event = event || window.event; // (*)
// Теперь event - объект события во всех браузерах.
}
```

Всплытие и перехват

- Всплытие:

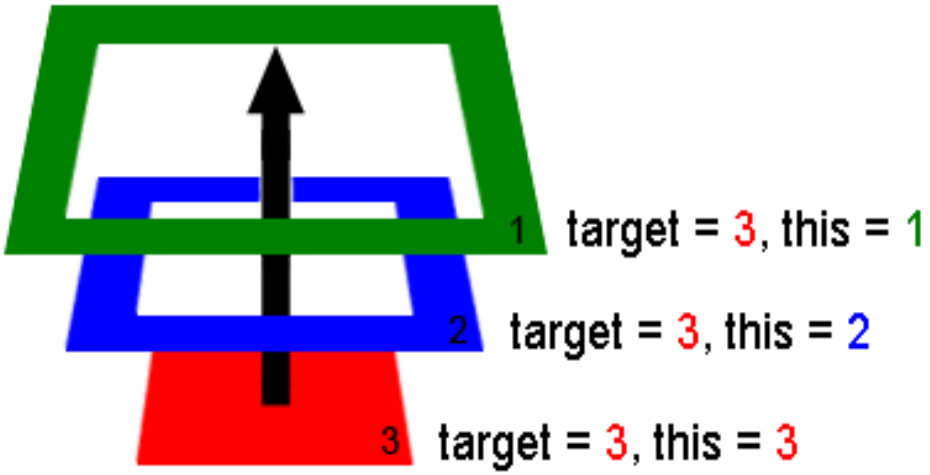
После того, как событие сработает на самом вложенном элементе, оно также сработает на родителях, вверх по цепочке вложенности.



Элементы

- Элемент, на котором сработал обработчик, доступен через this.
- Во всех браузерах, кроме IE<9, существует св-во event.currentTarget, аналогичное this.
- Самый глубокий элемент, который вызывает событие называется целевым, или исходным элементом.
  - event.srcElement (В IE<9)
  - event.target (остальные браузеры)

- event.target/srcElement - означает исходный элемент, на котором произошло событие.
- this - текущий элемент, до которого дошло всплытие и который запускает обработчик.



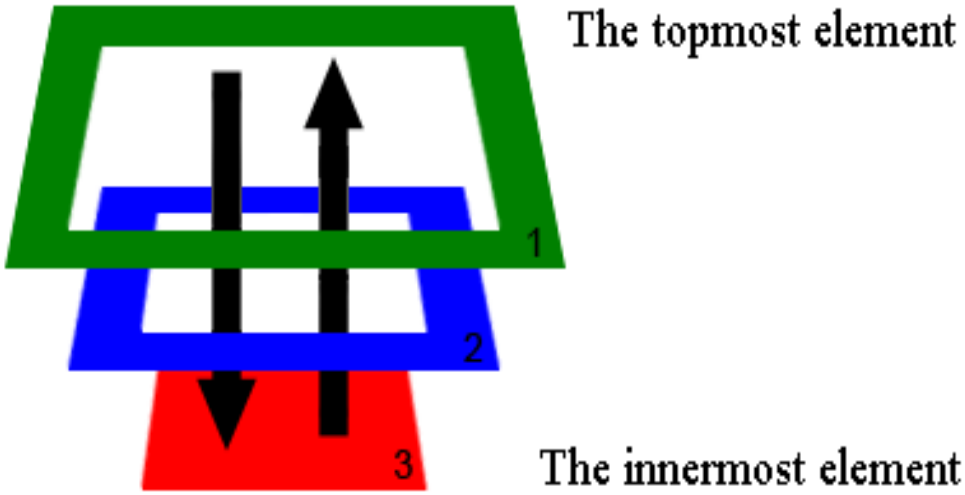
- event.target/srcElement не изменяется по мере всплытия события, а this изменяется.

Прекращение всплытия

- Любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.
  - Стандартный код — это вызов метода: event.stopPropagation()
  - Для IE<9 — это назначение свойства: event.cancelBubble = true
- Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены.

Стадия перехвата

- Перед тем, как всплыть, событие сначала идет сверху вниз. Эта стадия называется стадия перехвата (capturing stage).



- Может быть перехвачено по дороге вниз - через 1 -> 2 -> 3.
- Всплывает вверх - через 3 -> 2 -> 1.

- Единственный способ поймать событие на стадии перехвата — addEventListener с последним аргументом true.

  elem.addEventListener(type, handler, true);
- Смысл последнего аргумента:
  - true — Обработчик ставится на стадию захвата.
  - false — Обработчик ставится на стадию всплытия.
- Есть события, которые не всплывают, но которые можно захватить.

Действия браузера по умолчанию

- Многие события влекут за собой действие браузера.
- Если логика работы обработчика требует отменить действие браузера — это возможно.
  - Первый способ — это воспользоваться объектом события.
    - стандартный метод event.preventDefault()
    - а для IE<9 свойство event.returnValue = false
  - Если обработчик назначен через on..., то return false. Такой способ проще, но не будет работать, если обработчик назначен через addEventListener/attachEvent.

Действия, которые нельзя отменить

- Есть действия браузера, которые происходят до вызова обработчика. Такие действия нельзя отменить.
- Например, при клике по ссылке происходит фокусировка.

Делегирование событий

- Если у вас есть много элементов, события на которых нужно обрабатывать похожим образом, то не стоит присваивать отдельный обработчик каждому.
- Вместо этого, назначьте один обработчик общему родителю.
- Из него можно получить целевой элемент event.target, понять на каком потомке произошло событие и обработать его.

Пример

- Есть таблица следующей структуры:

***Bagua* Chart: Direction, Element, Color, Meaning**

Northwest Metal Silver Elders	**North** Water Blue Change	**Northeast** Earth Yellow Direction
West Metal Gold Youth	**Center** All Purple Harmony	**East** Wood Blue Future
Southwest Earth Brown Tranquility	**South** Fire Orange Fame	**Southeast** Wood Green Romance

- В каждой ячейке есть текст и тег STRONG

- Вместо того, чтобы назначать обработчик для каждой ячейки, назначен один *обработчик* для всей таблицы. Он использует event.target, чтобы получить элемент, на котором произошло событие.

```
table.onclick = function(event) {
 event = event || window.event;
var target = event.target || event.srcElement;
 // ... Обработать ...
}
```

- Клик может произойти на любом теге внутри таблицы.
- Для того, чтобы найти ячейку, нам нужно пройти цепочку parentNode

```
while(target != this) { // ( ** )
 if (target.tagName == 'TD') { // ( * )
   toggleHighlight(target); // ФУНКЦИЯ!
}
 target = target.parentNode;
 }
```

Осн. принципы делегирования

- В обработчике можно получить целевой элемент, на котором произошел клик. Он всегда самый вложенный, может быть как TD, так и STRONG внутри него. А еще клик может попасть в область между ячейками, если присутствует атрибут cellspacing. В этом случае целевым элементом будет TR или TABLE.
- Поднимаемся вверх по цепочке родителей *target = target.parentNode*, пока не встретим TD или не дойдем до TABLE.
- (*) Если мы TD - обрабатываем его. (**) Если мы дошли вверх до текущего элемента (таблицы), значит клик каким-то образом попал вне TD, и нам он не интересен.

События мыши

ОСНОВЫ

Типы и порядок событий мыши

- Простые события:

mousedown — Кнопка мыши нажата над элементом.

mouseup — Кнопка мыши отпущена над элементом.

mouseover — Мышь появилась над элементом.

mouseout — Мышь ушла с элемента.

mousemove — Каждое движение мыши над элементом генерирует это событие.

mousewheel — Прокрутка колесика мыши. В Firefox событие называется DOMMouseScroll.

- Комплексные события:

click — Вызывается при клике мышью, то есть при mousedown, а затем mouseup на одном элементе.

contextmenu — Вызывается при клике правой кнопкой мыши на элементе.

dblclick — Вызывается при двойном клике по элементу.

- Порядок срабатывания событий

Одно действие может вызывать несколько событий. Например, клик вызывает сначала mousedown при нажатии, а затем mouseup и click при отпускании кнопки.

В тех случаях, когда одно действие генерирует несколько событий, их порядок фиксирован.

Получение информации о кнопке: which/button

- Стандартные свойства

  which == 1 – левая кнопка

  which == 2 – средняя кнопка

  which == 3 – правая кнопка

- Свойство button для IE<9

  Является 3-х битным числом, в котором каждому биту соответствует кнопка мыши.

  Чтобы его расшифровать — нужна побитовая операция & («битовое И»). При этом мы можем узнать, были ли две кнопки нажаты одновременно.

Правый клик: oncontextmenu

- При клике правой кнопкой мыши, браузер показывает свое контекстное меню. Это является его действием по умолчанию:

<button oncontextmenu="alert('Клик!');">Правый клик сюда</button>

- Но если установлен обработчик события, то он может отменить действие по умолчанию и, тем самым, предотвратить появление встроенного меню.
- В примере ниже меню не будет:

<button oncontextmenu="alert('Клик!');return false">Правый клик сюда</button>

Модификаторы shift, alt, ctrl и meta

- Во всех событиях мыши присутствует информация о нажатых клавишах-модификаторах.
- Соответствующие свойства:

  shiftKey

  altKey

  ctrlKey

  metaKey (для Mac)

Координаты мыши

- Относительно окна: clientX/Y

Свойство clientX/clientY содержит координаты относительно window. Если прокрутить вниз, влево или вверх не сдвигая при этом мышь, то значения clientX/clientY не изменятся.

- Относительно документа: pageX/Y

Пара pageX/pageY содержит координаты относительно левого верхнего угла документа, вместе со всеми прокрутками. Эти свойства поддерживаются всеми браузерами, кроме IE<9 (их можно получить, прибавив к clientX/clientY величину прокрутки страницы).

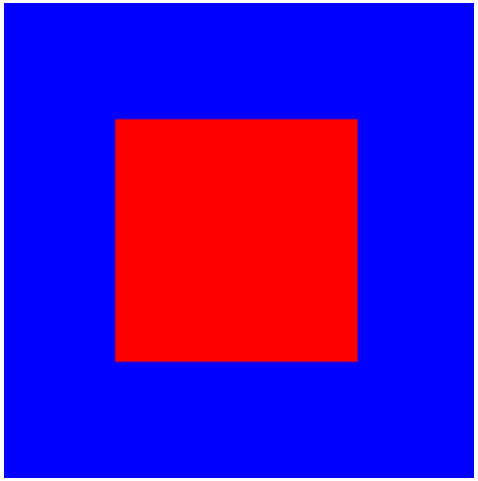
События mouseover/mouseout, свойство relatedTarget

- События mouseover (и mouseout) происходят, когда мышь появляется над элементом (и уходит с него).
- В этих событиях мышь переходит с одного элемента на другой, то есть участвуют два элемента, и оба их можно получить из свойств объекта события.

- mouseover — Элемент под курсором — *event.target (IE:srcElement)*. Элемент, с которого курсор пришел — *event.relatedTarget (IE:fromElement)*.
- mouseout — Элемент, с которого курсор пришел — *event.target (IE:srcElement*). Элемент под курсором — *event.relatedTarget (IE:toElement)*.

«Лишний» mouseout при уходе на потомка

- На синем стоит обработчик, который записывает его mouseover/mouseout.
- При заходе на синий — сработает mouseover.
- При переходе с синего на красный — будут выведены сразу два события:



1. mouseout [target: blue] — уход с родителя.
2. mouseover [target: red] — как ни странно, «обратный переход» на родителя.

События mouseenter и mouseleave

- События mouseenter/mouseleave похожи на mouseover/mouseout. Они тоже срабатывают, когда курсор заходит на элемент и уходит с него, но с двумя отличиями.

1. При переходе на потомка курсор не уходит с родителя.

Курсор заходит на элемент — срабатывает mouseenter, а затем — неважно, куда он внутри него переходит, mouseleave будет, когда курсор окажется за пределами элемента.

2. События mouseenter/mouseleave не всплывают.

- Эти события описаны в спецификации DOM Level 3 и поддерживаются IE6+, а также Opera 11.10+ и Firefox 10+.

Колёсико мыши: "mousewheel"

- **Отличия колёсика от прокрутки**
- При прокрутки срабатывает событие onscroll
- в том числе через клавиатуру, но *только на прокручиваемых элементах*.
- А событие mousewheel является событием только для мыши. Оно генерируется *над любым элементом* при передвижении колеса мыши. При этом не важно, прокручиваемый он или нет.
- **Свойство wheelDelta**
- wheelDelta — условный «размер прокрутки», обычно равен 120 для прокрутки вверх и -120 — вниз.
- В Firefox этого события нет, вместо него есть событие DOMMouseScroll со свойством detail, которое равно ±3.