JavaScript

Что такое JavaScript

JavaScript предназначен для разработки динамических web приложений.

Программы на этом языке называются скриптами. Они подключаются напрямую к HTML и, как только загружается страничка — тут же выполняются.

Процесс выполнения скрипта называют «интерпретацией».

Во все основные браузеры встроен интерпретатор JavaScript, именно поэтому они могут выполнять скрипты на странице.

JavaScript это полноценный язык, программы на котором можно запускать и на сервере, и на любом устройстве, если в нем установлен соответствующий интерпретатор.

Что умеет JavaScript?

Современный JavaScript — это «безопасный» язык программирования общего назначения. Он не предоставляет низкоуровневых средств работы с памятью, процессором, так как изначально был ориентирован на браузеры, в которых это не требуется.

В браузере JavaScript умеет делать все, что относится к манипуляции со страницей, взаимодействию с посетителем и, в какой-то мере, с сервером:

1) Создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т.п.
2) Реагировать на действия посетителя, обрабатывать клики мыши, перемещение курсора, нажатие на клавиатуру и т.п.
3) Посылать запросы на сервер и загружать данные без перезагрузки страницы(AJAX).
4) Получать и устанавливать cookie, запрашивать данные, выводить сообщения
5) …и многое, многое другое.

Что НЕ умеет JavaScript?

Большинство возможностей JavaScript в браузере ограничено текущим окном и страницей.

1) JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе.

2) Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией — песочницей. Возможности по доступу к устройствам также прорабатываются в современных стандартах и, частично, доступны в некоторых браузерах.

3) JavaScript, работающий в одной вкладке, почти не может общаться с другими вкладками и окнами. За исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол).

4) Есть способы это обойти, но для этого требуется как минимум явное согласие обоих сторон. Просто так взять и залезть в произвольную вкладку с другого домена нельзя.

5) Из JavaScript можно легко посылать запросы на сервер, с которого пришла страничка. Запрос на другой домен тоже возможен, но менее удобен, т.к. и здесь есть ограничения безопасности.

Этих ограничений нет там, где JavaScript используется вне браузера, например на сервере.

В чем уникальность JavaScript?

Есть как минимум три замечательных особенности JavaScript:

1) Полная интеграция с HTML/CSS.
2) Простые вещи делаются просто.
3) Поддерживается всеми распространенными браузерами и включен по умолчанию.

HTML 5

HTML 5 — стандарт HTML, добавляющий новые теги и, что более важно, ряд новых возможностей браузерам.

Вот несколько примеров:

1) Чтение/запись файлов на диск (в специальной «песочнице», то есть не любые).
2) Встроенная в браузер база данных, которая позволяет хранить данные на компьютере пользователя.
3) Многозадачность с одновременным использованием нескольких ядер процессора.
4) Проигрывание видео/аудио, без Flash.
5) 2d и 3d-рисование с аппаратной поддержкой, как в современных играх.

Альтернативные технологии

1) Java
2) ActiveX/NPAPI, плагины и расширения для браузера
3) Adobe Flash
4) Dart

Книги по JS

JavaScript

Полезное чтение о языке, встроенных методах и конструкциях JavaScript:

1) **JavaScript. Подробное руководство.** Дэвид Флэнаган.
2) **JavaScript. Шаблоны.** Стоян Стефанов.

jQuery

Кроме документации:

3) **jQuery. Подробное руководство по продвинутому JavaScript.** Бер Бибо, Иегуда Кац.

Привет, мир!

Программы на языке JavaScript можно вставить в любое место HTML при помощи тега SCRIPT.

```
<!DOCTYPE HTML>
<html>
<head>
  <!-- Тег meta для указания кодировки -->
  <meta charset="utf-8">
</head>
<body>
  <p>Начало документа...</p>
  <script>
    alert('Привет, Мир!');
  </script>
  <p>...Конец документа</p>
</body>
</html>
```

<script> ... </script>

Тег script содержит исполняемый код. Предыдущие стандарты HTML требовали обязательного указания атрибута type, но сейчас он уже не нужен. Достаточно просто <script>.

Браузер, когда видит <script>:

- Начинает отображать страницу, показывает часть документа до script
- Встретив тег script, переключается в JavaScript-режим и не показывает, а исполняет его содержимое.
- Закончив выполнение, возвращается обратно в HTML-режим и отображает оставшуюся часть документа.

alert(...)

Отображает окно с сообщением и ждет, пока посетитель не нажмет «Ок»

Внешние скрипты

Если JavaScript-кода много — его выносят в отдельный файл, который подключается в HTML:

<script src="/path/to/script1.js"></script>

<script src="/path/to/script2.js"></script>

Здесь /path/to/script.js - это абсолютный путь к файлу, содержащему скрипт (из корня сайта).

Браузер сам скачает скрипт и выполнит.

```
<html>
  <head>
    <meta charset="utf-8">
    <script src="rabbits.js"></script>
  </head>
  <body>
    <script>
      count_rabbits();
    </script>
  </body>
</html>
```

```
/* Файл rabbits.js */
function count_rabbits() {
    for(var i=1; i<=3; i++) {
        alert("Кролик номер "+i)
    }
}
```

Команды

Как правило, новая команда занимает отдельную строку,

и заканчивается символом **«;»**

```
alert('Привет');

alert('Мир');
```

Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку. Так тоже будет работать

```
alert('Привет')

alert('Мир')
```

В этом случае JavaScript интерпретирует переход на новую строчку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними.

Однако, внутренние правила по вставке точки с запятой не идеальны. В примере выше они сработали, но в некоторых ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна. Таких ситуаций не так много, но они все же есть, и ошибки, которые при этом появляются, достаточно сложно исправлять.

Поэтому рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт.

prompt()

Функция prompt принимает два аргумента:

```
result = prompt(title, default);
```

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием ESC на клавиатуре.

Вызов prompt возвращает то, что ввел посетитель - строку или специальное значение null, если ввод отменен.

Как и в случае с alert, окно prompt модальное.

```
var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')
```

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.

Он дописывает текст в текущее место HTML.

```
<script>
  document.write('<style> td { color: #F40 } </style>');
 </script>
<table>
  <tr>
  <script> document.write('<td>') </script>
    Текст внутри TD.
  <script> document.write('</td>') </script>
  </tr>
</table>
```

Нет никаких ограничений на содержимое document.write.

Строка просто пишется в HTML-документ без проверки структуры тегов

Также существует метод document.writeln(str) — который добавляет после str символ перевода строки '\n'.

Комментарии

Комментарии могут находиться в любом месте программы и никак не влияют на ее выполнение. Интерпретатор JavaScript попросту игнорирует их.

Однострочные комментарии начинаются с двойного слэша //. Текст считается комментарием до конца строки:

```
// Команда ниже говорит "Привет"

document.write('Привет');

document.write('Мир'); // Второе сообщение выводим отдельно
```

Многострочные комментарии начинаются слешем-звездочкой "/*" и заканчиваются звездочкой-слэшем "*/", вот так:

```
/* Пример с двумя сообщениями.

Это - многострочный комментарий. */

document.write('Привет');

document.write('Мир');
```

Переменные

Переменная состоит из имени и выделенной области памяти, которая ему соответствует.

Для объявления или, другими словами, создания переменной используется ключевое слово var:

```
var message;
```

После объявления, можно записать в переменную данные:

```
var message;

message = 'Привет'; // сохраним в переменной строку
```

Эти данные будут сохранены в соответствующей области памяти и в дальнейшем доступны при обращении по имени:

Для краткости можно совместить объявление переменной и запись данных:

```
var message = 'Привет';

document.write(message); // выведет содержимое переменной
```

Копирование значений

Переменные в JavaScript могут хранить не только строки, но и другие данные, например, числа.

Объявим две переменные, положим в одну - строку, а в другую - число.

Как вы можете видеть, переменной без разницы, что хранить:

```
var num = 100500;

var message = 'Привет';
```

Значение можно копировать из одной переменной в другую.

```
var num = 100500;

var message = 'Привет';

message = num;

//Значение из num перезаписывает текущее в message.
```

Важность директивы var

В JavaScript вы можете создать переменную и без var, достаточно просто присвоить ей значение:

x = "value"; // переменная создана, если ее не было

Технически, это не вызовет ошибки, но делать так все-таки не стоит.

Всегда определяйте переменные через var. Это хороший тон в программировании и помогает избежать ошибок.

```
<html>
<head>
  <meta http-equiv="X-UA-Compatible"
content="IE=8">
</head>
<body>
  <div id="test"></div>
  <script>
    test = 5;
    alert(test);
  </script>
</body>
</html>
```

Значение не выведется, будет ошибка. Почти все современные браузеры создают переменные для элементов, у которых есть id, а такую переменную изменять нельзя.

Ошибки не будет, если объявить test, используя var:

var test = 5;

Константы

Константа — это переменная, которая никогда не меняется. Как правило, их называют большими буквами, через подчёркивание. Например:

```
var COLOR_BLUE = "#00F";

var COLOR_RED = "#0F0";

var COLOR_GREEN = "#F00";

var COLOR_ORANGE = "#FF7F00";

document.write(COLOR_RED); // #0F0
```

Технически, константа является обычной переменной, то есть её можно изменить. Но существует некая негласная договоренность этого не делать.

Зачем нужны константы? Почему бы просто не использовать "#F00" или "#0F0"?

1) Во-первых, константа — это понятное имя, в отличие от строки "#FF7F00".
2) Во-вторых, опечатка в строке может быть не замечена, а в имени константы её упустить невозможно — будет ошибка при выполнении.

Константы используют вместо строк и цифр, чтобы сделать программу понятнее и избежать ошибок.

Имена переменных

На имя переменной наложены два ограничения.

1) Имя может состоять из: латинских букв, цифр, символов $ и _
2) Первый символ не должен быть цифрой.
3) Регистр букв имеет значение

Например:

```
var myName;

var test123;
```

Что здесь особенно интересно - доллар '$' и знак подчеркивания '_' являются такими же обычными символами, как буквы:

```
var $ = 5;  // объявили переменную с именем '$'

var _ = 15; // переменная с именем '_'

document.write($);
```

А такие переменные были бы неправильными:

```
var 1a; // начало не может быть цифрой

var my-name; // дефис '-' не является разрешенным символом
```

Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать при именовании переменных, так как они используются самим языком, например: **var**, **class**, **return**, **implements** и др.

Некоторые слова, например, class, не используются в современном JavaScript, но они заняты на будущее. Некоторые браузеры позволяют их использовать, но это может привести к ошибкам.

Следующий пример будет работать во многих старых браузерах, которые допускают использование слова 'class' и не сработает в современных. Они вернут синтаксическую ошибку:

```
var class = 5;

document.write(class); //error
```

Правильный выбор имени

Гораздо проще найти нужные данные, если они правильно помечены, т.е. переменная названа правильно.

Желательно использовать:

1) Корректные английские слова.
2) Использовать короткие имена только для переменных «местного значения»
3) Переменные из нескольких слов пишутся *вместеВотТак*. (camelCase)

   Например:

   ```
   var borderLeftWidth;
   ```

   Существует альтернативный стандарт, когда несколько слов пишутся через знак подчеркивания '_':

   ```
   var border_left_width;
   ```

4) Имя переменной должно максимально четко соответствовать хранимым в ней данным.

Задача 1

1) Объявите две переменные: admin и name.
2) Запишите в name строку "Василий".
3) Скопируйте значение из name в admin.
4) Выведите admin

Примитивные типы данных

1) **Число number:**

```
var n = 123;

N = 12.345;
```

Тип число используется как для целых, так и для дробных чисел. Также есть специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений). Они также принадлежат типу «число».

2) **Строка string:**

```
var str = "Мама мыла раму";

str = 'Одинарные кавычки тоже подойдут';
```

В JavaScript одинарные и двойные кавычки равноправны. Можно использовать или те или другие.

3) **Булевый (логический) тип boolean**

```
var checked = true; // поле формы помечено галочкой

checked = false;    // поле формы не содержит галочки
```

4) **null**

```
var x = null;
```

Специальное значение, имеет смысл «ничего». Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null

5) **undefined**

```
var u;

alert(u); // выведет "undefined"
```

Специальное значение, которое, как и null, образует свой собственный тип. Оно имеет смысл «значение не присвоено».

Присваивание

Оператор присваивания выглядит как знак равенства =

Он вычисляет выражение, которое находится справа, и присваивает результат переменной. Это выражение может быть достаточно сложным и включать в себя любые другие переменные:

```
var a = 1;

var b = 2;

a = b + a + 3;

document.write(a); // 6
```

Возможно присваивание по цепочке:

```
var a, b, c;

a = b = c = 2 + 2;
```

Такое присваивание работает справа-налево, то есть сначала вычислятся самое правое выражение 2+2, присвоится в c, затем выполнится b = c и, наконец, a = b.

Оператор "=" возвращает значение

Все операторы возвращают значение. Вызов x = ... записывает выражение справа в x, а затем возвращает его. Так что присваивание можно использовать как часть более сложного выражения:

```
var a = 1;
var b = 2;
var c = 3 - (a = b); // c = 1
document.write(a = b); // 2
```

В примере выше результатом (a = b) является значение, которое записывается в a (т.е. 2). Оно используется для вычисления c.

Арифметические операторы

плюс +, минус -, умножить *, поделить /

```
document.write(2 + 2); // 4
 var i = 2;
 i = (2 + i) * 3 / i;
 document.write(i); // 6
```

оператор % остаток от деления.

```
 document.write(5 % 2); // 1, остаток от деления 5 на 2
```

Термины: «унарный», «бинарный», «операнд»

Операнд — то, к чему применяется оператор. Например: 5 * 2 — оператор умножения с левым и правым операндами. Другое название: «аргумент оператора».

Унарным называется оператор, который применяется к одному выражению. Например, оператор унарный минус "-" меняет знак числа на противоположный:

```
var x = 1;

document.write( -x ); // -1, унарный минус
```

Бинарным называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

```
var x = 1, y = 3;

document.write( y - x ); // 2, бинарный минус
```

Большинство операторов — бинарные, например, умножение "*" требует ровно два аргумента.

Операторы плюс "+" и минус "-" в JavaScript существуют и в бинарной и в унарной формах.

Работа унарного "+" и бинарного "+" в JavaScript существенно различается.

Это действительно разные операторы. Бинарный плюс складывает операнды, а унарный — приводит операнд к числовому типу.

Сложение строк

Если бинарный оператор + применить к строкам, то он их объединяет в одну:

```
var a = "моя" + "строка";

document.write(a); // моястрока
```

Если хотя бы один аргумент является строкой, то второй будет также преобразован к строке!

Причем не важно, справа или слева находится операнд-строка, в любом случае нестроковый аргумент будет преобразован. Например:

```
document.write( '1' + 2 ); // "12"

document.write( 2 + '1' ); // "21"
```

Приведение к строке — особенность бинарного оператора плюс "+". Любой другой арифметический оператор, наоборот, всегда преобразует операнды к числам, например:

```
document.write( '1' - 2 ); // -1

document.write( 6 / '2'); // 3
```

Приоритет

В том случае, если в выражении есть несколько операторов - порядок их выполнения определяется приоритетом.

Каждый оператор получает числовой приоритет, и тот, у кого число меньше — выполнится раньше. Если приоритет одинаковый, то порядок выполнения — слева направо.

…	…	…
5	умножение	*
5	деление	/
6	сложение	+
6	вычитание	-
16	присвоение	=
…	…	…

В выражении x = 2 * 2 + 5 сначала выполнится умножение * (самый высокий приоритет: 5), затем сложение + (приоритет: 6), и после них — присвоение =(приоритет: 16).

Инкремент/декремент: ++, --

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу.

Для этого существуют даже специальные операторы:

Инкремент ++ увеличивает на 1:

```
var i = 2;
i++;  // более короткая запись для i = i + 1.
document.write(i); // 3
```

Декремент -- уменьшает на 1:

```
var i = 2;
i--;   // более короткая запись для i = i - 1.
document.write(i); // 1
```

Вызывать эти операторы можно не только после, но и перед переменной: i++ (называется «**постфиксная форма**») или ++i («**префиксная форма**»).

1) Если результат оператора не используется, а нужно только увеличить/уменьшить переменную — без разницы, какую форму использовать:

2) Если хочется тут же использовать результат, то нужна префиксная форма:

```
var i = 0;  document.write( ++i ); //
1
```

3) Если нужно увеличить, но нужно значение переменной до увеличения — постфиксная форма:

```
var i = 0; document.write( i++ ); // 0
```

Побитовые операторы

Побитовые операторы рассматривают аргументы как 32-разрядные целые числа и работают на уровне их внутреннего двоичного представления.

1)	AND	(и)	(&)
2)	OR	(или)	(|)
3)	XOR	(побитовое исключающее или)	(^)
4)	NOT	(не)	(~)
5)	LEFT SHIFT	(левый сдвиг)	(<<)
6)	RIGHT SHIFT	(правый сдвиг)	(>>)
7)	ZERO-FILL RIGHT SHIFT	(правый сдвиг с заполнением нулями)	(>>>)

Вызов операторов с присваиванием

Часто нужно применить оператор к переменной и сохранить результат в ней же, например:

```
n = n + 5;

d = d * 2;
```

Этот синтаксис можно сократить при помощи совмещенных операторов: +=, -=, *=, /=, >>=, <<=, >>>=, &=, |=, ^=, вот так:

```
n = 2;

n += 5; // теперь n=7 (аналогично n = n + 5)

n *= 2; // теперь n=14 (аналогично n = n * 2)
```

Все операторы присваивания имеют очень низкий приоритет, т.к. должны выполняться после остальных вычислений.

Оператор запятая

Запятая тоже является оператором. Ее можно вызвать явным образом, например:

```
a = (5, 6);

document.write(a);
```

Запятая позволяет перечислять выражения, разделяя их запятой ','. Каждое из них — вычисляется и отбрасывается, за исключением последнего, которое возвращается.

Запятая - единственный оператор, приоритет которого ниже присваивания. В выражении a = (5,6) для явного приоритета использованы скобки, иначе оператор '=' выполнялся бы до запятой ',', т.е. как (a=5), 6.

Зачем нужен такой оператор, который отбрасывает значения всех перечисленных выражений, кроме последнего?

Обычно он используется в составе более сложных конструкций, чтобы сделать несколько действий в одной строке.

```
for (a = 1, b = 3, c = a*b; a < 10; a++) {  … }
```

Операторы сравнения и логические значения

1) **Больше/меньше**: a > b, a < b.
2) **Больше/меньше или равно**: a >= b, a <= b.
3) **Равно** a == b. Для сравнения используется два символа равенства '='. Один символ a = b означал бы присваивание.
4) «**Не равно**». В JavaScript — знак равенства с восклицательным знаком перед ним !=.

Логические значения

Как и другие операторы, сравнение возвращает значение. Это значение имеет специальный логический тип.

Существует всего два логических значения:

1) **true** — имеет смысл «да», «верно», «истина».
2) **false** — означает «нет», «неверно», «ложь».

```
document.write( 2 > 1 ); // true, верно

document.write( 2 == 1 ); // false, неверно

document.write( 2 != 1 ); // true
```

Логические значения можно использовать и напрямую, присваивать переменным, работать с ними как с любыми другими:

```
var a = true; // присвоили явно
```

Сравнение строк

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа — свой номер (код). JavaScript использует кодировку Unicode. При этом сравниваются численные коды символов.

Строки сравниваются побуквенно:

```
document.write( 'Б' > 'А' ); // true
```

Буквы сравниваются в алфавитном порядке. Какая буква в алфавите позже — та и больше.

Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой. При этом любая буква больше отсутствия буквы.

```
document.write( 'Вася' > 'Ваня' ); // true, т.к. 'с' > 'н'

document.write( 'Привет' > 'Прив' ); // true, так как 'е' больше чем «ничего»
```

Сравнение разных типов

При сравнении значения преобразуются к числу, кроме случая когда оба значения — строки.

```
document.write( '2' > 1 ); // true

document.write( '01' == 1 ); //true

document.write( false == 0 ); // true, false становится 0, а true 1.
```

Строгое равенство

Обычное равенство не может отличить 0 от false:

```
document.write(0 == false); // true, т.к. false преобразуется к 0
```

Что же делать, если все же нужно отличить 0 от false?

Для проверки равенства без преобразования типов используются операторы строгого равенства === (тройное равно) и !==.

Они сравнивают без приведения типов. Если тип разный, то такие значения всегда неравны (строго):

```
document.write(0 === false); // false, т.к. типы различны
```

Специальные значения NULL и UNDEFINED

Проблемы со специальными значениями возможны, когда к переменной применяется операция сравнения > < <= >=, а у неё может быть как численное значение, так и null/undefined.

Программисты, пришедшие из других языков, зачастую ошибочно считают, что null/undefined эквивалентны нулю, но это не так.

1) Значения null и undefined равны == друг другу и не равны чему бы то ни было ещё.
2) При преобразовании в число null становится 0, а undefined становится NaN (числовое «значение-ошибка»).

Задача 2

1) Даны две переменные - a и b. Например, они равны a=2; b=3. Необходимо поменять местами эти числа (то есть a=3, b=2), не используя при этом третью переменную.
2) Ход вычислений должен быть выведен на страницу в виде нумерованного списка.

Оператор if

Оператор if («если») получает условие, вычисляет его, и если результат — true, то выполняет команду.

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year != 2011) alert('А вот и неправильно!');
```

Если нужно выполнить более одной команды — они оформляются блоком кода в фигурных скобках:

```
if (year != 2011) {

  alert('А вот..');

  alert('..и неправильно!');

}
```

Рекомендуется использовать фигурные скобки всегда, даже когда команда одна. Это улучшает читаемость кода.

Преобразование к логическому типу

Оператор if вычисляет и преобразует результат к логическому типу.

В логическом контексте число 0, пустая строка "", null и undefined, а также NaN являются false, остальные значения — true.

Например, такое условие никогда не выполнится:

```
if (0) { // 0 преобразуется к false
  ...
}
```

А такое — выполнится всегда:

```
if (1) { // 1 преобразуется к true
  ...
}
```

Вычисление условия в проверке if (year != 2011) может быть вынесено в отдельную переменную:

```
var cond = (year != 2011);  // true/false

if (cond) {

 ...

}
```

Неверное условие, else

Необязательный блок else («иначе») выполняется, если условие неверно:

```
var year = prompt('Введите год ECMA-262 5.1', '');
if (year == 2011) {
  alert('Верно!');
} else {
  alert('Неверно!'); // любое значение, кроме 2011
}
```

Несколько условий, else if

Бывает нужно проверить несколько вариантов условия. Для этого используется блок else if .... Например:

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year < 2011) {

 alert('Позднее');

} else if (year > 2011) {

 alert('Раньше');

} else {

 alert('Верно!');

}
```

В примере выше JavaScript сначала проверит первое условие, если оно ложно — перейдет ко второму — и так далее, до последнего else.

Оператор вопросительный знак '?'

Иногда нужно в зависимости от условия присвоить переменную.

Оператор вопросительный знак '?' позволяет делать это короче и проще чем if.

Он состоит из трех частей:

условие ? значение1 : значение2

Проверяется условие, затем если оно верно — возвращается значение1 , если неверно — значение2, например:

```
var access;

var age = prompt('Ваш возраст?', '');

access = (age > 14) ? true : false;

alert(access);
```

```
if (age > 14) {
   access = true;
} else {
   access = false;
}
```

Оператор '?' выполняется позже большинства других, в частности — позже сравнений, поэтому скобки можно не ставить:

```
access = age > 14 ? true : false;
```

Но когда скобки есть — код лучше читается. Так что рекомендуется их писать.

Несколько операторов '?'

Несколько операторов if..else можно заменить последовательностью операторов '?'. Например:

```
var a = prompt('a?', 1);

var res = (a == 1) ? 'значение1' :

 (a == 2) ? 'значение2' :

 (a > 2) ? 'значение3' :

 'значение4';

alert(res);
```

```
var res;
if (a == 1) {
  res = 'значение1';
} else if (a == 2) {
  res = 'значение2';
} else if (a > 2) {
  res = 'значение3';
} else {
  res = 'значение4';
}
```

Поначалу может быть сложно понять, что происходит. Однако, внимательно приглядевшись, мы замечаем, что это обычный if..else!

Вопросительный знак проверяет сначала a == 1, если верно — возвращает значение1, если нет — идет проверять a == 2. Если это верно — возвращает значение2, иначе проверка a > 2 и значение3.. Наконец, если ничего не верно, то значение4.

|| (ИЛИ)

Логическое ИЛИ в классическом программировании работает следующим образом: «если хотя бы один из аргументов true, то возвращает true, иначе — false».

Получается следующая таблица результатов:

```
alert( true  || true ); // true

alert( false || true ); // true

alert( true  || false); // true

alert( false || false); // false
```

Обычно оператор ИЛИ используется в if, чтобы проверить, выполняется ли хотя бы одно из условий, например:

```
var hour = 9;

if (hour < 10 || hour > 18) {

 alert('Время работы с 10 до 18 ');

}
```

```
var hour = 12, isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert('Время работы с 10 до 18,
выходной воскресенье');
}
```

Короткий цикл вычислений

JavaScript вычисляет несколько ИЛИ слева направо. При этом, чтобы экономить ресурсы, используется так называемый «короткий цикл вычисления».

Допустим, вычисляются несколько ИЛИ подряд: a || b || c || .... Если первый аргумент — true, то результат заведомо будет true (хотя бы одно из значений — true), и остальные значения игнорируются.

Это особенно заметно, когда выражение, переданное в качестве второго аргумента, имеет сторонний эффект — например, присваивает переменную.

При запуске примера ниже присвоение x не произойдёт:

```
var x;

true || (x = 1);  // просто вычислим ИЛИ, без if

alert(x); // undefined, x не присвоен
```

Значение ИЛИ

Итак, как мы видим, оператор ИЛИ вычисляет ровно столько значений, столько необходимо — до первого true.

Оператор ИЛИ возвращает то значение, на котором остановились вычисления.

```
alert( 1 || 0 ); // 1

alert( true || 'неважно что'); // true

alert( null || 1 ); // 1

alert( undefined || 0 ); // 0
```

&& (И)

В классическом программировании И возвращает true, если оба аргумента истинны, а иначе — false

```
alert( true  && true ); // true

alert( false && true ); // false

alert( true  && false); // false

alert( false && false); // false
```

```
var hour = 12, minute = 30;
if (hour == 12 && minute == 30) {
  alert('Время 12:30');
}
```

К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ. Если левый аргумент — false, оператор И возвращает его и заканчивает вычисления. Иначе — вычисляет и возвращает правый аргумент.

```
// Первый аргумент - true,
// Поэтому возвращается второй аргумент
alert(1 && 0);  // 0
alert(1 && 5);  // 5
```

```
// Первый аргумент - false,
// Он и возвращается, а второй
// аргумент игнорируется
alert(null && 5);  // null
alert(0 && "не важно"); // 0
```

Приоритет оператора И && больше, чем ИЛИ ||, т.е. он выполняется раньше.

! (НЕ)

```
var result = !value;
```

1) Сначала приводит аргумент к логическому типу true/false.
2) Затем возвращает противоположное значение.

Например:

```
alert( !true ) // false
alert( !0 )  // true
```

Двойное НЕ используется для преобразования значений к логическому типу:

```
alert( !!"строка" )  // true
alert( !!null ) // false
```

Цикл while

Цикл while имеет вид:

```
while (условие) {

 // код, тело цикла

}
```

Пока условие верно — выполняется код из тела цикла.

Например, цикл ниже выводит i пока i < 3:

```
var i = 0;
while (i < 3) {
  alert(i);
  i++;
}
```

Бесконечный цикл:

```
while (true) {
  // ...
}
```

Повторение цикла называется «итерация». Цикл в примере выше совершает три итерации.

Если бы i++ в коде выше не было, то цикл выполнялся бы (в теории) вечно. На практике, браузер выведет сообщение о «зависшем» скрипте и посетитель его остановит.

Цикл do..while

Проверку условия можно поставить под телом цикла, используя специальный синтаксис do..while:

```
do {

  // тело цикла

} while (условие);
```

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

Например:

```
var i = 0;
do {
  alert(i);
  i++;
} while (i < 3);
```

Цикл for

Чаще всего применяется цикл for. Выглядит он так:

```
for (начало; условие; шаг) {

 // ... тело цикла ...

}
```

Например, цикл ниже выводит значения от 0 до 3 (не включая 3):

```
var i;
for (i=0; i<3; i++) {
  alert(i);
}
```

1. Начало i=0 выполняется при заходе в цикл.
2. Условие i<3 проверяется перед каждой итерацией.
3. Шаг i++ выполняется после каждой итерации, но перед проверкой условия.

В цикле также можно определить переменную:

```
for (var i=0; i<3; i++) {
  ...
}
```

Любая часть for может быть пропущена.

```
for (;;) {
  // будет выполняться вечно
}
```

Выход: break

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива break.

Например, бесконечный цикл в примере прекратит выполнение при i==5:

```
var i=0;
 while(1) {
  i++;
   if (i==5) break;
   alert(i);
}
 alert('Последняя i = '+ i ); // 5 (*)
```

Выполнение продолжится со строки (*), следующей за циклом.

Следующая итерация: continue

Директива continue прекращает выполнение текущей итерации цикла. Например, цикл ниже не выводит четные значения:

```
for (var i = 0; i < 10; i++) {

  if (i % 2 == 0) continue;

  alert(i);
}
```

Для четных i срабатывает continue, выполнение блока прекращается и управление передается на for.

Метки

Метка имеет вид "имя:", имя должно быть уникальным. Она ставится перед циклом, вот так:

outer: for (var i = 0; i < 3; i++) { ... }

Можно также выносить ее на отдельную строку. Вызов break outer прерывает управление цикла с такой меткой, вот так:

```
outer:
for (var i = 0; i < 3; i++) {
  for (var j = 0; j < 3; j++) {
    var input = prompt('Значение в координатах '+i+','+j, '');
    if (input == null) break outer; // (*)
  }
}
alert('Готово!');
```

Директива continue также может быть использована с меткой. Управление перепрыгнет на следующую итерацию цикла с меткой.

В некоторых языках программирования есть оператор goto, который может передавать управление на любой участок программы.
Операторы break/continue более ограниченны. Они работают только внутри циклов, и метка должна быть не где угодно, а выше по уровню вложенности.
В JavaScript нет goto.

Конструкция switch

Синтаксис

```
switch(x) {
 case 'value1':  // if (x === 'value1')
   ...
   [break]
 case 'value2':  // if (x === 'value2')
   ...
   [break]
 default:
   ...
   [break]
}
```

```
var a = 2+2;

switch (a) {
  case 3:
    alert('Больше на единицу');
    break;
  case 4:
    alert('Верно!');
    break;
  default:
    alert('Неверно');
```

1) Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее. Тип переменной имеет значение

2) Если соответствие установлено — switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).

   При этом case называют вариантами switch.

3) Если ни один case не совпал — выполняется (если есть) вариант default.

Группировка case

Несколько значений case можно группировать.

В примере ниже case 3 и case 5 выполняют один и тот же код:

```
var a = 2+2;

switch (a) {
  case 4:
    alert('Верно!');
    break;

  case 3:                    // (*)
  case 5:                    // (**)
    alert('Отличается на единицу!');
    break;

  default:
    alert('Недопустимое значение');
}
```

При case 3 выполнение идёт со строки (*) и идёт вниз до ближайшего break, таким образом проходя и то, что предназначено для case 5.

Задача 3

Напишите скрипт, выводящий в окно браузера таблицу умножения. Задайте разный цвет фона для четных и нечетных значений.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Функции

Зачастую, нам надо повторять одно и то же действие во многих частях программы.

Например, вывести сообщение необходимо при приветствии посетителя, при выходе посетителя с сайта, еще где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели — это alert(message).

Но можно создавать и свои.

prompt()

Функция prompt принимает два аргумента:

```
result = prompt(title, default);
```

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием ESC на клавиатуре.

Вызов prompt возвращает то, что ввел посетитель - строку или специальное значение null, если ввод отменен.

Как и в случае с alert, окно prompt модальное.

```
var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')
```

confirm()

Синтаксис:

```
result = confirm(question);
```

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL.

Результатом будет true при нажатии OK и false - при CANCEL(Esc).

Например:

```
var isAdmin = confirm("Вы - администратор?")
alert(isAdmin);
```

Объявление

Пример объявления функции:

```
function showMessage() {
  alert('Сообщение');
}
```

Вначале идет ключевое слово function, после него имя функции, затем список параметров в скобках (в примере выше он пустой) и тело функции — код, который вызывается при её вызове.

Объявленная функция доступна по имени, например:

```
function showMessage() {
  alert('Сообщение');
}
showMessage();
showMessage();
```

Этот код выведет сообщение два раза. Уже здесь видна главная цель создания функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода — достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

Функция может содержать локальные переменные, объявленные через var. Такие переменные видны только внутри функции:

```
function showMessage() {
  var message = 'Сообщение'; // локальная переменная
   alert(message);
}
 showMessage(); // 'Сообщение'
 alert(message); // <-- будет ошибка, т.к. переменная видна только внутри
```

Блоки if/else, switch, for и т.п. не влияют на область видимости. При объявлении переменной в таких блоках, она всё равно будет видна во всей функции.

Например:

```
function count() {
  for (var i=0; i<3; i++) {
    /* .. */
  }

  alert(i); // i=3, последнее значение цикла
}
```

Внешние переменные

Функция может обратиться ко внешней переменной. Доступ возможен не только на чтение, но и на запись. При этом, так как переменная внешняя, то изменения будут видны и снаружи функции:

```
var userName = 'Имя1';

function showMessage() {
  userName = 'Имя2'; // (1) присвоение во внешнюю переменную
  var message =  userName;
  alert(message);
}

showMessage();
alert(userName); // Имя2, значение внешней переменной изменено функцией
```

Конечно, если бы в строке (1) была бы объявлена локальная переменная var userName = ..., то все обращения использовали бы её, и внешняя переменная осталась бы неизменной.

Глобальные переменные

Переменные, объявленные на уровне всего скрипта, называют «глобальными переменными».

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта.

В старом стандарте JavaScript существовала возможность неявного объявления переменных присвоением значения.

Например:

```
function showMessage() {
  message = 'Сообщение'; // без var!
}
showMessage();
alert(message); // Сообщение
```

В коде выше message нигде не объявлена, а сразу присваивается.

В современном стандарте JavaScript такое присвоение запрещено, а в старом, который работает в браузерах по умолчанию, переменная будет создана автоматически, причём в примере выше она создаётся не в функции, а на уровне всего скрипта.

Параметры

При вызове функции ей можно передать данные, которые та использует по своему усмотрению.

Например, этот код выводит два сообщения:

```
function showMessage(title, text) {
  title = '*' + title + '*';  // (1), оформили title
  alert(title + '\n\n' + text);
}

var title = 'Заголовок', msg = 'Сообщение'; // (2)

showMessage(title, msg); // значения будут скопированы в параметры
alert(title); // перезапись в строке (1) не повлияет на внешнюю переменную
```

Параметры копируются в локальные переменные функции.

В примере изменение title в строке (1) не отразится на значении внешней переменной title (2), т.к. изменена была копия значения:

Аргументы по умолчанию

Если какой-то параметр не передан при вызове — он считается равным undefined.

Например, у функции показа сообщения showMessage(text, title) может быть обязателен аргумент text, но не обязателен title.

```
function showMessage(title, text) {
  if (text === undefined) {
    text = 'Текст не передан!';
  }

  alert(title + ": " + text);
}

showMessage("Заголовок", "Сообщение"); // Заголовок: Сообщение
showMessage("Заголовок"); // Заголовок: Текст не передан!
```

Возврат значения

Функция может возвратить результат, который будет передан в вызвавший её код.

Для возврата значения используется директива return. Она может находиться в любом месте функции. Как только до нее доходит управление — функция завершается и значение передается обратно.

Вызовов return может быть и несколько, например:

```
function checkNum(num) {
  if (num > 10) {
    return true;
  } else {
    return false;
  }
}

var num = prompt('Введите число');

if (checkNum(num)) {
  alert('Больше чем 10');
} else {
  alert('Меньше чем 10');
}
```

Директива return может также использоваться без значения, чтобы прекратить выполнение и выйти из функции, например:

```
function showNum(num) {
  if (!checkNum(num)) {
    return;
    alert(123); // (никогда не выполнится)
  }

  // ...
}
```

Выбор имени

Имя функции следует тем же правилам, что и имя переменной. Основное отличие — оно должно быть глаголом, т.к. функция — это действие.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

Функции, которые начинаются с "show" — что-то показывают:

showMessage(..) // префикс show, "показать" сообщение

Функции, начинающиеся с "get" — получают, и т.п.:

getNum(..) // get, "получает" число

calcD(..) // calc, "вычисляет" дискриминант

createForm(..) // create, "создает" форму

checkPermission(..) // check, "проверяет" разрешение, возвращает true/false

Это очень удобно, поскольку взглянув на функцию — мы уже примерно представляем, что она делает и что она возвращает, даже если функцию написал другой программист.

Рекурсия

Функция может вызывать не только другие функции, но и сама себя. Это называется рекурсией.

Чтобы возвести x в натуральную степень n — можно умножить его на себя n раз в цикле.

А можно поступить проще. Ведь x^n = x * x^(n-1), т.е. можно вынести один x из-под степени. Таким образом, значение функции pow(x,n) легко получается из pow(x, n-1).

```
function pow(x, n) {
  var result = x;
  for(var i=1; i<n; i++) {
    result *= x;
  }
  return result;
}
```

```
function pow(x, n) {
  // пока n!=1,
 // сводить вычисление pow(..,n) к pow(..,n-1)
  return (n != 1) ? x*pow(x, n-1) : x;
}

alert( pow(2, 3) ); // 8
```

Говорят, что «функция pow рекурсивно вызывает сама себя».

Значение, на котором рекурсия заканчивается называют базисом рекурсии. В примере выше базисом является 1.

Общее количество вложенных подвызовов называют глубиной рекурсии. В случае со степенью, всего будет n-1 подвызовов. Максимальная глубина рекурсии ограничена и составляет около 10000, с точностью до порядка в зависимости от браузера.

Методы и свойства

Все значения в JavaScript, за исключением null и undefined, содержат набор вспомогательных функций и значений, доступных «через точку».

Такие функции называют «методами», а значения — «свойствами». Посмотрим на примеры.

У строки есть свойство length, содержащее длину:

```
alert( "Привет, мир!".length ); // 12
```

Еще у строк есть метод toUpperCase(), который возвращает строку в верхнем регистре:

```
var hello = "Привет, мир!";

alert( hello.toUpperCase() ); // "ПРИВЕТ, МИР!"
```

Если через точку вызывается функция (toUpperCase()) — это называют «вызов метода», если просто читаем значение (length) — «получение свойства».

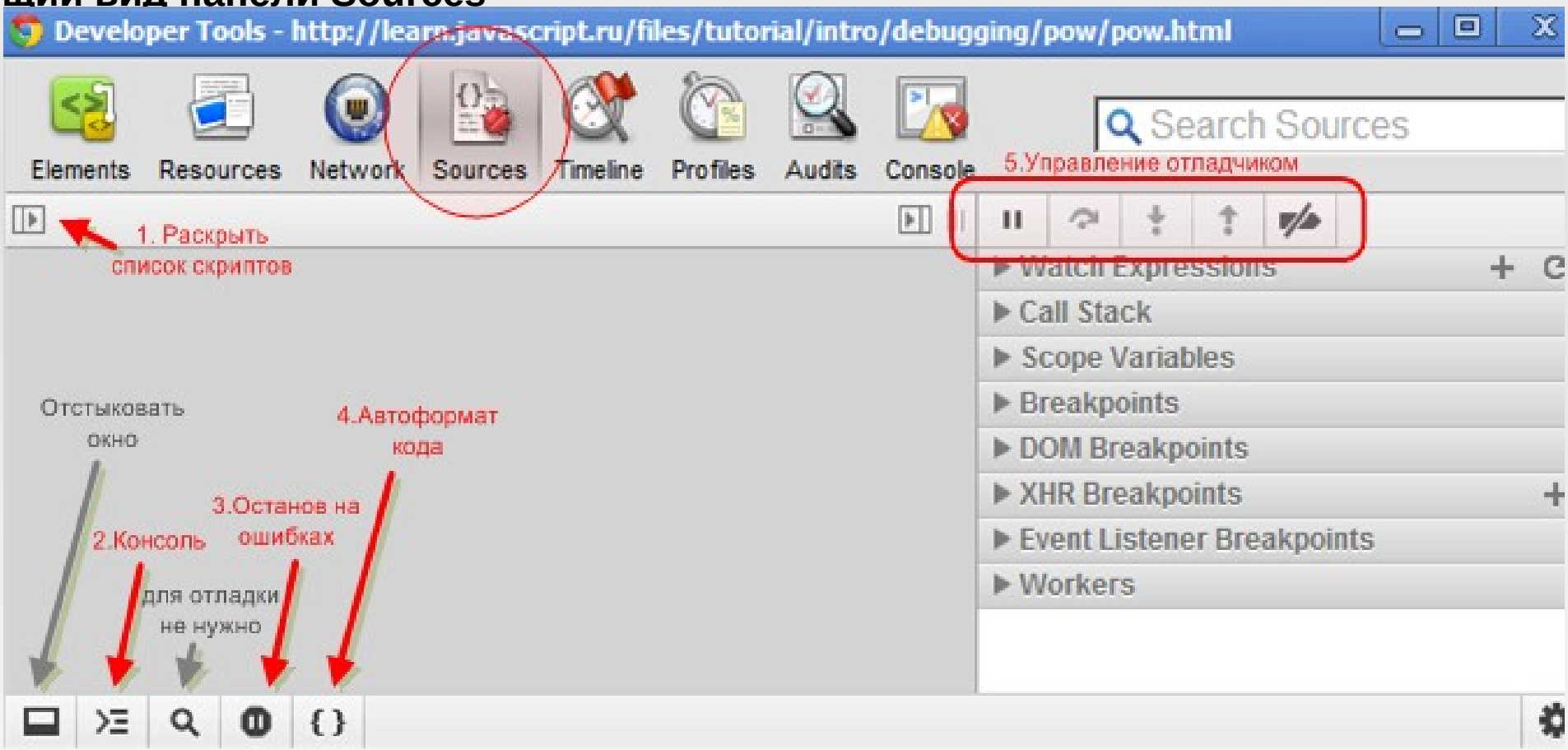
У чисел есть метод num.toFixed(n). Он округляет число num до n знаков после запятой, при необходимости добивает нулями до данной длины и возвращает в виде строки (удобно для форматированного вывода):

```
var n = 12.345;

alert( n.toFixed(2) ); // "12.34"
alert( n.toFixed(0) ); // "12"
alert( n.toFixed(5) ); // "12.34500"
```

Отладка

Отладка в браузере Chrome

Общий вид панели Sources

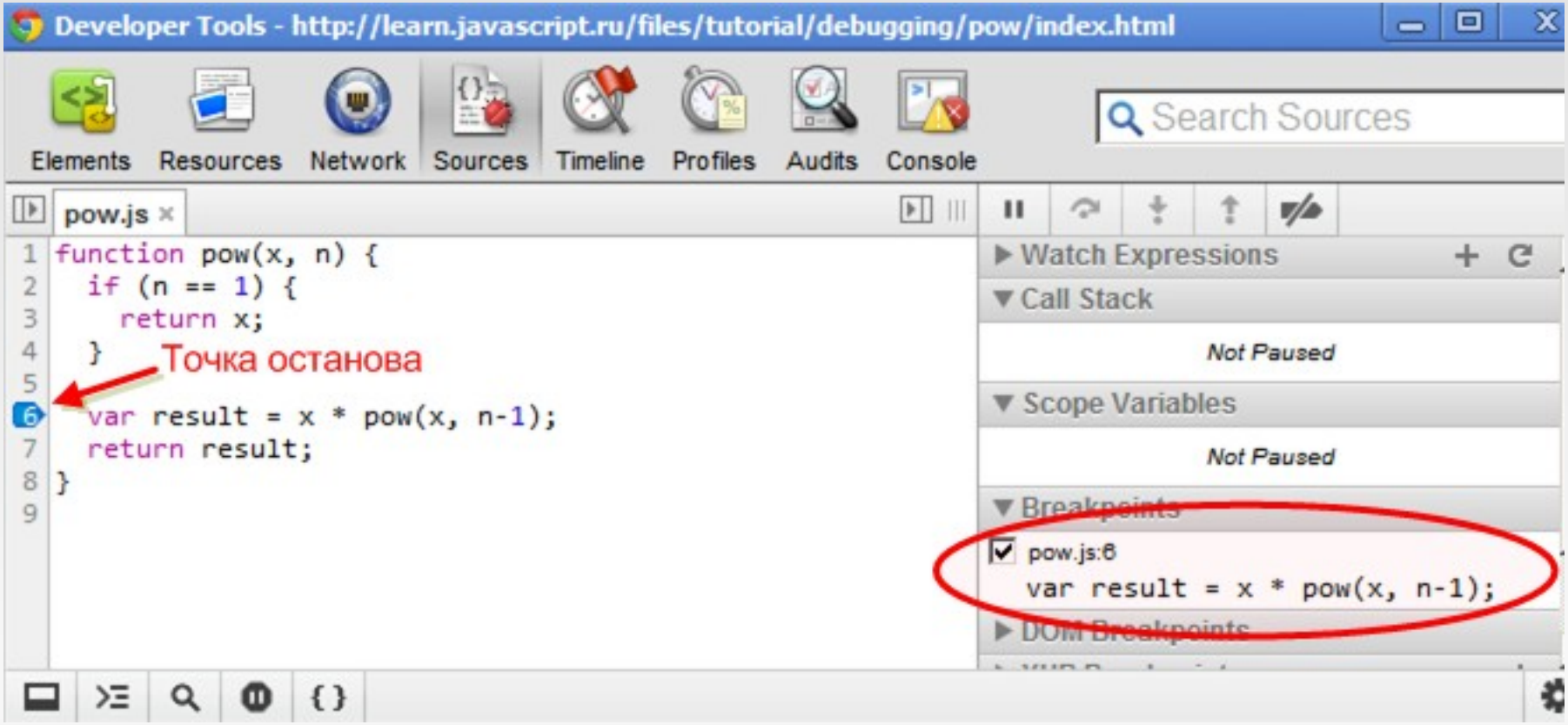


1. Раскрывает список скриптов и стилей, подключённых к странице.
2. Включает-выключает консоль (ESC).
3. Двойное нажатие на эту кнопку заставляет отладчик останавливаться на ошибках JavaScript (кнопка поменяет цвет на фиолетовый).
4. Код можно отлаживать в отформатированном виде, если включить эту кнопку.
5. Панель управления потоком исполнения в режиме отладки.

Точки останова

Слово Брейкпойнт (breakpoint) — это то место в коде, где отладчик будет останавливать выполнение JavaScript, как только оно до него дойдёт.

В остановленном коде можно посмотреть любые переменные, выполнить команды и т.п.



Останов можно инициировать и напрямую из кода скрипта, командой **debugger**

Текущие значения

Developer Tools - http://learn.javascript.ru/files/tutorial/debugging/pow/index.html

Elements Resources Network Sources Timeline Profiles Audits Console

Search Sources

pow.js

```
1 function pow(x, n) {
2   if (n == 1) {
3     return x;
4   }
5
6   var result = x * pow(x, n-1);
7   return result;
8 }
9
```

Pause

Watch Expressions

Call Stack

pow pow.js:6

(anonymous function) index.html:13

Scope Variables

Local

n: 3

result: undefined

this: Window

x: 5

Global Window

Breakpoints

pow.js:6

var result = x * pow(x, n-1);

DOM Breakpoints

Watch Expressions — показывает текущие значения любых выражений.
Call Stack — вложенные вызовы (иначе называют «стек вызовов»).
Scope Variables — переменные.

Управление выполнением

Процесс отладки заключается в том, что мы останавливаем скрипт, смотрим, что с переменными, переходим дальше и ищем, где поведение отклоняется от правильного.

Команды управления выполнением:

— продолжить выполнение (F8). Если скрипт не встретит новых точек останова, то на этом работа в отладчике закончится.

— сделать шаг, не заходя внутрь функции (F10). Выполняет одну строку скрипта. Если в ней есть вызов функции — то отладчик обходит его стороной, т.е. не переходит на код внутри.

— сделать шаг (F11). Переходит на следующую строку кода. Если есть вложенный вызов, то внутрь функции.

— выполнять до выхода из текущей функции (Shift + F11). Как только текущая функция завершилась, отладчик тут же останавливает скрипт. Удобно в случае, если мы нечаянно вошли во вложенный вызов, который нам совсем не интересен — чтобы быстро из него выйти.

— отключить/включить все точки останова. Эта кнопка никак не двигает нас по коду, она позволяет временно отключить все точки останова в файле.

Консоль

При отладке, кроме просмотра переменных, бывает полезно запускать команды JavaScript. Для этого нужна консоль.

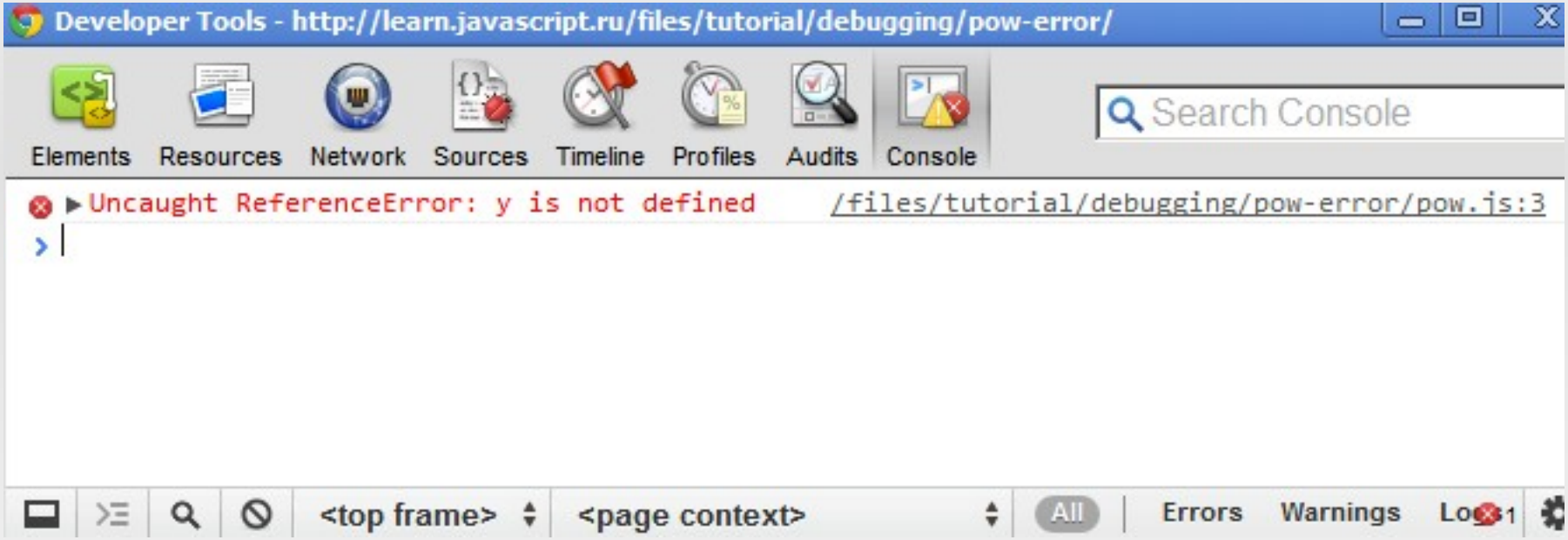
В неё можно перейти, нажав кнопку «Console» вверху-справа, а можно и открыть в дополнение к отладчику, нажав на кнопку или клавишей ESC.

Команда **console.log(...)** - пишет переданные ей аргументы в консоль, например:

```
// результат будет виден в консоли

for(var i=0; i<5; i++) {

  console.log("значение", i);

}
```

Ошибки

Ошибки JavaScript выводятся в консоли.



Задача 4

Факториал числа — это число, умноженное на «себя минус один», затем на «себя минус два» и так далее, до единицы. Обозначается n! = n*(n-1)*(n-2)*...*1. Например:

1! = 1

2! = 2*1 = 2

3! = 3*2*1 = 6

4! = 4*3*2*1 = 24

5! = 5*4*3*2*1 = 120

Задача — написать функцию factorial(n), которая возвращает факториал числа n!, используя рекурсивный вызов. Значение n задается пользователем в модальном окне. Скрипт должен выводить в таблицу все запрошенные значения факториалов пока не будет нажата кнопка отмены в модальном окне.

Подсказка: обратите внимание, что n! можно записать как n * (n-1)!, например 3! = 3*2! = 3*2*1! = 6

Структуры данных

Строки

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренним форматом строк, вне зависимости от кодировки страницы, является Юникод (Unicode)

Строки создаются при помощи двойных или одинарных кавычек:

```
var text = "моя строка";

var anotherText = 'еще строка';

var str = "012345";
```

В JavaScript нет разницы между двойными и одинарными кавычками

Специальные символы

Строки могут содержать специальные символы.

Символ	Описание
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab
\uNNNN	Символ в кодировке Юникод с шестнадцатиричным кодом NNNN. Например, \u00A9 — юникодное представление символа копирайт ©

Экранирование специальных символов

Если строка в одинарных кавычках, то внутренние одинарные кавычки внутри должны быть экранированы, то есть снабжены обратным слешем \', вот так:

```
var str = 'I\'m a JavaScript programmer';
```

В двойных кавычках — экранируются внутренние двойные:

```
var str = "I'm a JavaScript \"programmer\" ";

alert(str);
```

Экранирование служит исключительно для правильного восприятия строки JavaScript. В памяти строка будет содержать сам символ без '\'.

Сам символ обратного слэша '\' является служебным, поэтому всегда экранируется, т.е пишется как \\:

```
var str = ' символ \\ ';

alert(str); // символ \
```

Заэкранировать можно любой символ. Если он не специальный, то ничего не произойдёт:

```
alert( "\a" ); // a

// идентично alert( "a" );
```

Методы и свойства

1) **Длина length**

Одно из самых частых действий со строкой — это получение ее длины:

```
var str = "My\n"; // 3 символа. Третий - перевод строки
alert(str.length);  // 3
```

2) **Доступ к символам**

Чтобы получить символ, используйте вызов charAt(позиция).

Первый символ имеет позицию 0:

```
var str = "jQuery";
alert( str.charAt(0) );  // "j"
```

В JavaScript нет отдельного типа «символ», так что charAt возвращает строку, состоящую из выбранного символа.

3) **Изменения строк**

Строки в JavaScript нельзя изменять. Можно прочитать символ, но нельзя заменить его. Как только строка создана — она такая навсегда. Чтобы это обойти, создаётся новая строка и присваивается в переменную вместо старой:

```
var str = "строка"; str =  str.charAt(3) + str.charAt(4) + str.charAt(5); alert(str); // ока
```

Смена регистра

Методы toLowerCase() и toUpperCase() меняют регистр строки на нижний/верхний:

```
alert( "Интерфейс".toUpperCase() ); // ИНТЕРФЕЙС
```

Пример ниже получает первый символ и приводит его к нижнему регистру:

```
alert( "Интерфейс".charAt(0).toLowerCase() ); // 'и'
```

Поиск подстроки

Для поиска подстроки есть метод **indexOf**(подстрока[, начальная_позиция]).

Он возвращает позицию, на которой находится подстрока или -1, если ничего не найдено. Например:

```
var str = "Widget with id";

alert( str.indexOf("Widget") ); // 0, т.к. "Widget" найден прямо в начале str

alert( str.indexOf("id") ); // 1, т.к. "id" найден, начиная с позиции 1

alert( str.indexOf("notinstr") ); // -1, подстрока не найден
```

Необязательный второй аргумент позволяет искать, начиная с указанной позиции. Например, первый раз "id" появляется на позиции 1. Чтобы найти его следующее появление — запустим поиск с позиции 2:

```
var str = "Widget with id";

alert( str.indexOf("id", 2) ) // 12, поиск начат с позиции 2
```

Также существует аналогичный метод lastIndexOf, который ищет не с начала, а с конца строки.

Поиск всех вхождений

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Как только получаем очередную позицию — начинаем следующий поиск со следующей.

```
var str = "Строка в которой ищем вхождение"; // ищем в этой строке
var target = "в";        // цель поиска

var pos = 0;
while(true) {
  var foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert(foundPos); // нашли на этой позиции
  pos = foundPos + 1; // продолжить поиск со следующей
}
```

Такой цикл начинает поиск с позиции 0, затем найдя подстроку на позиции foundPos, следующий поиск продолжит с позиции pos = foundPos+1, и так далее, пока что-то находит.

Взятие подстроки: substr, substring, slice.

1) **substring(start [, end])**

Метод substring(start, end) возвращает подстроку с позиции start до, но не включая end.

```
var str = "stringify";
alert(str.substring(0,1)); // "s", символы с позиции 0 по 1 не включая 1.
```

Если аргумент end отсутствует, то поиск идет до конца строки:

2) **substr(start [, length])**

Первый аргумент имеет такой же смысл, как и в substring, а второй содержит не конечную позицию, а количество символов.

```
var str = "stringify";
str = str.substr(2,4); // ring, со 2й позиции 4 символа
alert(str)
```

Если второго аргумента нет — подразумевается «до конца строки».

3) **slice(start [, end])**

Возвращает часть строки от позиции start до, но не включая, позиции end. Смысл параметров — такой же как в substring.

Отрицательные аргументы

Различие между substring и slice — в том, как они работают с отрицательными и выходящими за границу строки аргументами:

1) **substring(start, end)**

Отрицательные аргументы интерпретируются как равные нулю. Слишком большие значения усекаются до длины строки:

```
alert( "testme".substring(-2) );  // "testme", -2 становится 0
```

Кроме того, если start > end, то аргументы меняются местами, т.е. возвращается участок строки между start и end:

```
alert( "testme".substring(4, -1) );  // "test"
// -1 становится 0 -> получили substring(4, 0)
// 4 > 0, так что аргументы меняются местами -> substring(0, 4) = "test"
```

2) **slice**

Отрицательные значения отсчитываются от конца строки:

```
alert( "testme".slice(-2) );  // "me", от 2 позиции с конца

alert( "testme".slice(1, -1) ); // "estm", от 1 позиции до первой с конца.
```

Кодировка Юникод

Все строки имеют внутреннюю кодировку Юникод.

Неважно, на каком языке написана страница, находится ли она в windows-1251 или utf-8. Внутри JavaScript-интерпретатора все строки приводятся к единому «юникодному» виду. Каждому символу соответствует свой код.

Символы сравниваются не по алфавиту, а по коду. У кого код больше — тот и больше. В юникоде есть много разных символов. Кириллическим буквам соответствует только небольшая часть из них

Строчные буквы идут после заглавных, поэтому они всегда больше. В частности, 'а'(код 1072) > 'Я'(код 1071). То же самое происходит и в английском алфавите, там 'a' > 'Z'.

Ряд букв, например ё, находятся вне основного алфавита. В частности, маленькая буква ё имеет код, больший чем я, поэтому 'ё'(код 1105) > 'я'(код 1103). Кстати, большая буква Ё располагается в Unicode до А, поэтому 'Ё'(код 1025) < 'А'(код 1040).

Сравнение строк

Строки сравниваются лексикографически, в порядке «телефонного справочника».

Сравнение строк s1 и s2 обрабатывается по следующему алгоритму:

Сравниваются первые символы: a = s1.charAt(0) и b = s2.charAt(0). Если они одинаковы, то следующий шаг, иначе, в зависимости от результата их сравнения, возвратить true или false

Сравниваются вторые символы, затем третьи и так далее… Если в одной строке закончились символы, то она меньше. Если в обоих закончились — они равны.

Спецификация языка определяет этот алгоритм более детально, но смысл в точности соответствует порядку, по которому имена заносятся в телефонный справочник.

"Z" > "A" // true

"Вася" > "Ваня" // true, т.к. с > н

"aa" > "a" // true, т.к. начало совпадает, но в 1й строке больше символов

Регулярные выражения

Регулярные выражения - это формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (wildcard characters). По сути это строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

В регулярных выражениях различают следующие виды символов:

Обычные символы

A..z — английские буквы от A до z,
строчные и заглавные;
0..9 — цифры;
{ } — фигурные скобки, кроме случаев,
когда они составляют группу вида {n,m}
(где n и m — числа) и её вариации;
= — равно;
< — меньше;
> — больше;
- — минус;
, — запятая;
и др.

Специальные символы

( ) — круглые скобки;
[ ] — квадратные скобки;
\ — обраный слеш;
. — точка;
^ — степень;
$ — знак доллара;
| — вертикальная черта;
? — вопросительный знак;
+ — плюс.

foob[aeiou]r - находит 'foobar', 'foober' и т.д. но не 'foobbr', 'foobcr' и т. д.
foob.*r - находит 'foobar', 'foobalkjdflkj9r' и 'foobr'

Спецсимволы в регулярном выражении

1) \ Для обычных символов - делает их специальными. Например, выражение /s/ ищет просто символ 's'. А если поставить \ перед s, то /\s/ уже обозначает пробельный символ.И наоборот, если символ специальный, например *, то \ сделает его просто обычным символом "звездочка". Например, /a*/ ищет 0 или больше подряд идущих символов 'a'. Чтобы найти a со звездочкой 'a*' - поставим \ перед спец. символом: /a*/.

2) ^ Обозначает начало входных данных. Если установлен флаг многострочного поиска ("m"), то также сработает при начале новой строки.Например, /^A/ не найдет 'A' в "an A", но найдет первое 'A' в "An A."

3) $ Обозначает конец входных данных. Если установлен флаг многострочного поиска, то также сработает в конце строки.Например, /t$/ не найдет 't' в "eater", но найдет - в "eat".

4) * Обозначает повторение 0 или более раз. Например, /bo*/ найдет 'boooo' в "A ghost booooed" и 'b' в "A bird warbled", но ничего не найдет в "A goat grunted".

5) + Обозначает повторение 1 или более раз. Эквивалентно {1,}. Например, /a+/ найдет 'a' в "candy" и все 'a' в "caaaaaaandy".

6) \s Найдет любой пробельный символ, включая пробел, табуляцию, переводы строки и другие юникодные пробельные символы. Например, /\s\w*/ найдет ' bar' в "foo bar."

Спецсимволы в регулярном выражении

1)? Обозначает, что элемент может как присутствовать, так и отсутствовать. Например, /e?le?/ найдет 'el' в "angel" и 'le' в "angle."Если используется сразу после одного из квантификаторов *, +, ?, или {}, то задает "нежадный" поиск (повторение минимально возможное количество раз, до ближайшего следующего элемента паттерна), в противоположность "жадному" режиму по умолчанию, при котором количество повторений максимально, даже если следующий элемент паттерна тоже подходит.Кроме того, ? используется в предпросмотре, который описан в таблице под (?=), (?!), и (?:).

2). (Десятичная точка) обозначает любой символ, кроме перевода строки: \n \r \u2028 or \u2029. (можно использовать [\s\S] для поиска любого символа, включая переводы строк). Например, /.n/ найдет 'an' и 'on' в "nay, an apple is on the tree", но не 'nay'.

3)() Находит x и запоминает. Это называется "запоминающие скобки". Например, /(foo)/ найдет и запомнит 'foo' в "foo bar." Найденная подстрока хранится в массиве-результате поиска или в предопределенных свойствах объекта RegExp: $1, ..., $9.Кроме того, скобки объединяют то, что в них находится, в единый элемент паттерна. Например, (abc)* - повторение abc 0 и более раз.

4)\w Найдет любой словесный (латинский алфавит) символ, включая буквы, цифры и знак подчеркивания. Эквивалентно [A-Za-z0-9_]. Например, /\w/ найдет 'a' в "apple," '5' в "$5.28," и '3' в "3D."

RegExp

```
// полная форма записи
var expr = new RegExp(pattern [, flags]);
// сокращенная форма записи (литеральный формат)
var expr = /pattern/flags;
```

pattern

Шаблон поиска (текст регулярного выражения).

flags

Способы поиска по шаблону:

g — глобальный поиск (обрабатываются все совпадения с шаблоном поиска);
i — не различать строчные и заглавные буквы;
m — многострочный поиск.

Порядок указания флагов не имеет значения.

Когда регулярное выражение создается при помощи конструктора new RegExp(…), необходимо помнить, что обратные слеши (\) должны экранироваться, например:

```
var expr = new RegExp('\\w', 'ig');
```

При использовании литерального формата, этого делать не нужно:

```
var expr = /\w/gi;
```

exec, test

exec Метод RegExp, который выполняет поиск совпадения в строке. Он возвращает массив данных.

```
var myRe = /ab*/g;
var str = "abbcdefabh";
var myArray;
while ((myArray = myRe.exec(str)) !== null)
{
  var msg = "Found " + myArray[0] + ".  ";
  msg += "Next match starts at " + myRe.lastIndex;
  console.log(msg);
}
//Found abb. Next match starts at 3
//Found ab. Next match starts at 9
```

test Метод RegExp, который тестирует совпадение в строке. Возвращет либо истинну либо ложь.

```
function testinput(re, str){
   if (re.test(str)) {
      midstring = " contains ";
   } else {
      midstring = " does not contain ";
   }
   document.write (str + midstring + re.source);
}
```

search, match

1) **match** Метод String, который выполняет поиск совпадения в строке. Он возвращет массив данных либо null если совпадения отсутствуют.

```
str.match(regexp) //regexp регулярное выражение в виде объекта RegExp или строка
```

2) **search** Метод String, который тестирует на совпадение в строке. Он возвращет индекс совпадения, или -1 если совпадений не будет найдено.

```
idx = str.search(regexp) //regexp - объект типа RegExp или строка
```

```
function testinput(re, str){
  if (str.search(re) != -1)
    return true
  else
    return false
}
alert(testinput(/something/, "somestr"))
 // false
```

replace

1) **replace** Метод String, который выполняет поиск совпадения в строке, и заменяет совпавшую подстроку другой подстрокой переданной как аргумент в этот метод.

```
str.replace(regexp, newSubStr|function)
```

```
var re = /apples/gi;
var str = "Apples are round, and apples are juicy.";
var newstr = str.replace(re, "oranges");
//newstr  = oranges are round, and oranges are juicy
```

2) **split** Метод String, который использует регулярное выражение или фиксированую строку чтобы разбить строку на массив подстрок.

```
string.split([separator][, limit])
```

```
var names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand ";
var re = /\s*;\s*/;
var nameList = names.split(re);
//names = Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ;Chris Hand
//nameList = Harry Trump,Fred Barney,Helen Rigby,Bill Abel,Chris Hand
```

Задача 5

1) Нужно создать функцию которая будет принимать любую текстовую строку и возвращать строку полученную из исходной, где каждое слово в перевернутом виде. Тоесть передаем 'Привет мир!' получаем 'тевирП !рим'.
2) Создать фунуцию которая подсчитает количество слов в строке.

Числа

Все числа в JavaScript, как целые так и дробные, имеют тип Number и хранятся в 64-битном формате IEEE-754, также известным как «double precision».

Способы записи

В JavaScript можно записывать числа не только в десятичной, но и в шестнадцатеричной (начинается с 0x), а также восьмиричной (начинается с 0) системах счисления:

```
alert( 0xFF ); // 255 в шестнадцатиричной системе

alert( 010 ); // 8 в восьмиричной системе
```

Также доступна запись в «научном формате» (ещё говорят «запись с плавающей точкой»), который выглядит как <число>e<кол-во нулей>.

Например, 1e3 — это 1 с 3 нулями, то есть 1000.

```
// еще пример научной формы: 3 с 5 нулями

alert( 3e5 );  // 300000
```

Если количество нулей отрицательно, то число сдвигается вправо за десятичную точку:

```
alert( 3e-5 ); // 0.00003  <-- 5 нулей, включая начальный ноль
```

Деление на ноль, Infinity

В математическом анализе деление на ноль не является ошибкой. В качестве результата мы получаем бесконечность, Infinity. Если число отрицательное, то минус бесконечность -Infinity.

Так же и в JavaScript:

```
alert(1/0);  // Infinity

alert(-1/0); // -Infinity
```

Infinity - особенное численное значение, которое ведет себя в точности как математическая бесконечность ∞.

1) Infinity больше любого числа.

2) Добавление к бесконечности не меняет её.

```
alert(Infinity > 99999999999999999999999999); // true

alert(Infinity + 5 == Infinity);  // true
```

Бесконечность можно присвоить и в явном виде: var x = Infinity.

NaN

Если математическая операция не может быть совершена, то возвращается специальное значение NaN (Not-A-Number).

Например, деление 0/0 в математическом смысле неопределено, поэтому возвращает NaN:

```
alert( 0 / 0 );  // NaN
```

Значение NaN используется для обозначения математической ошибки и обладает следующими свойствами:

1) Значение NaN — единственное, в своем роде, которое не равно ничему, включая себя. Следующий код ничего не выведет:

```
if (NaN == NaN) alert("=="); // Ни один вызов

if (NaN === NaN) alert("==="); // не сработает
```

2) Значение NaN можно проверить специальной функцией isNaN(n), которая возвращает true если аргумент — NaN и false для любого другого значения.

```
var n = 0/0;

alert( isNaN(n) ); // true
```

3) Любая операция с NaN возвращает NaN.

```
alert( NaN + 1 ); // NaN
```

isFinite(n)

Итак, в JavaScript есть обычные числа и три специальных числовых значения: NaN, Infinity и -Infinity.

Функция isFinite(n) возвращает true только тогда, когда n — обычное число, а не одно из этих значений:

```
alert( isFinite(1) ); // true

alert( isFinite(Infinity) ); // false

alert( isFinite(NaN) ); // false
```

Если аргумент isFinite — не число, то он автоматически преобразуется к числу.

Преобразование к числу

Строгое преобразование можно осуществить унарным плюсом '+'

```
var s = "12.34";

alert( +s );  // 12.34
```

Строгое — означает, что если строка не является в точности числом, то результат будет NaN:

```
alert( +"12test" );  // NaN
```

Единственное исключение — пробельные символы в начале и в конце строки, которые игнорируются:

```
alert( +"  -12");  // -12

alert( +" \n34  \n"); // 34, перевод строки \n является пробельным символом

alert( +"" ); // 0, пустая строка становится нулем

alert( +"1 2" ); // NaN, пробел посередине числа - ошибка
```

Аналогичным образом происходит преобразование и в других математических операторах и функциях:

```
alert( '12.34' / "-2" );  // -6.17
```

isNaN — проверка на число для строк

Функция isNaN является математической, она преобразует аргумент в число, а затем проверяет, NaN это или нет.

Поэтому можно использовать ее для проверки:

```
var x = "-11.5";
if (isNaN(x)) {
  alert("Строка преобразовалась в NaN. Не число");
} else {
  alert("Число");
}
```

Единственный тонкий момент — в том, что пустая строка и строка из пробельных символов преобразуются к 0:

```
alert(isNaN(" \n\n  ")) // false, т.к. строка из пробелов преобразуется к 0
```

И, конечно же, проверка isNaN посчитает числами значения false, true, null, т.к. они хотя и не числа, но преобразуются к ним:

+false = 0 +null = 0

+true = 1 +undefined = NaN;

Мягкое преобразование: parseInt и parseFloat

В мире HTML/CSS многие значения не являются в точности числами. Например, метрики CSS: 10pt или -12px.

Оператор '+' для таких значений возвратит NaN:

```
alert( +"12px" ) // NaN
```

Для удобного чтения таких значений существует функция parseInt:

```
alert( parseInt('12px') ); // 12
```

parseInt и ее аналог parseFloat преобразуют строку символ за символом, пока это возможно.

При возникновении ошибки возвращается число, которое получилось. parseInt читает из строки целое число, а parseFloat — дробное.

```
alert( parseInt('12px') ) // 12, ошибка на символе 'p'

alert( parseFloat('12.3.4') ) // 12.3, ошибка на второй точке
```

Конечно, существуют ситуации, когда parseInt/parseFloat возвращают NaN. Это происходит при ошибке на первом же символе:

```
alert( parseInt('a123') ); // NaN
```

Проверка на число для всех типов

Если вам нужна действительно точная проверка на число, которая не считает числом строку из пробелов, логические и специальные значения — используйте следующую функцию isNumeric:

```
function isNumeric(n) {
  return !isNaN(parseFloat(n)) && isFinite(n);
}
```

Разберёмся, как она работает. Начнём справа.

1) Функция isFinite(n) преобразует аргумент к числу и возвращает true, если это не Infinity/-Infinity/NaN.

Таким образом, правая часть отсеет заведомо не-числа, но оставит такие значения как true/false/null и пустую строку '', т.к. они корректно преобразуются в числа.

2) Для их проверки нужна левая часть. Вызов parseFloat(true/false/null/'') вернёт NaN для этих значений.

Так устроена функция parseFloat: она преобразует аргумент к строке, т.е. true/false/null становятся "true"/"false"/"null", а затем считывает из неё число, при этом пустая строка даёт NaN.

В результате отсеивается всё, кроме строк-чисел и обычных чисел.

Округление

Одна из самых частых операций с числом — округление. В JavaScript существуют целых 3 функции для этого.

1) Math.floor Округляет вниз
2) Math.ceil Округляет вверх
3) Math.round Округляет до ближайшего целого

```
alert( Math.floor(3.1) );  // 3

alert( Math.ceil(3.1) );   // 4

alert( Math.round(3.1) );  // 3
```

Округление до заданной точности

Обычный трюк — это умножить и поделить на 10 с нужным количеством нулей. Например, округлим 3.456 до 2го знака после запятой:

```
var n = 3.456;

alert( Math.round( n * 100 ) / 100 );  // 3.456 -> 345.6 -> 346 -> 3.46
```

Таким образом можно округлять число и вверх и вниз.

1) **num.toFixed(precision) -** округляет число num до точности precision и возвращает результат в виде строки:

```
var n = 12.34;

alert( n.toFixed(1) ); // "12.3"
```

Округление идёт до ближайшего значения, аналогично Math.round:

```
var n = 12.36;

alert( n.toFixed(1) ); // "12.4"
```

Итоговая строка, при необходимости, дополняется нулями до нужной точности:

```
var n = 12.34;

alert( n.toFixed(5) ); // "12.34000", добавлены нули до 5 знаков после запятой
```

Неточные вычисления

```
alert(0.1 + 0.2 == 0.3);  //false

 alert(0.1 + 0.2); // 0.30000000000000004
```

Дело в том, что в стандарте IEEE 754 на число выделяется ровно 8 байт(=64 бита), не больше и не меньше.

Число 0.1 (=1/10) короткое в десятичном формате, а в двоичной системе счисления это бесконечная дробь. Также бесконечной дробью является 0.2 (=2/10).

Двоичное значение бесконечных дробей хранится только до определенного знака, поэтому возникает неточность. Это даже можно увидеть:

```
 alert( 0.1.toFixed(20) );  // 0.10000000000000000555
```

есть два способа сложить 0.1 и 0.2:

1) Сделать их целыми, сложить, а потом поделить:
2) Сложить, а затем округлить до разумного знака после запятой. Округления до 10-го знака обычно бывает достаточно, чтобы отсечь ошибку вычислений:

Тригонометрия

1) Math.acos(x) Возвращает арккосинус x (в радианах)
2) Math.asin(x) Возвращает арксинус x (в радианах)
3) Math.atan Возвращает арктангенс x (в радианах)
4) Math.atan2(y, x) Возвращает угол до точки (y, x). Описание функции: Atan2.
5) Math.sin(x) Вычисляет синус x (в радианах)
6) Math.cos(x) Вычисляет косинус x (в радианах)
7) Math.tan(x) Возвращает тангенс x (в радианах)

Функции общего назначения

1) Math.sqrt(x) Возвращает квадратный корень из x.
2) Math.log(x) Возвращает натуральный (по основанию e) логарифм x.
3) Math.pow(x, exp) Возводит число в степень, возвращает x^exp, например Math.pow(2,3) = 8. Работает в том числе с дробными и отрицательными степенями, например: Math.pow(4, -1/2) = 0.5.
4) Math.abs(x) Возвращает абсолютное значение числа
5) Math.exp(x) Возвращает e^x, где e — основание натуральных логарифмов.
6) Math.max(a, b, c...) Возвращает наибольший из списка аргументов
7) Math.min(a, b, c...) Возвращает наименьший из списка аргументов
8) Math.random(x) Возвращает псевдо-случайное число в интервале [0,1) - то есть между 0(включительно) и 1(не включая). Генератор случайных чисел инициализуется текущим временем.

Задача 6

1) Последовательность чисел Фибоначчи имеет формулу Fn = Fn-1 + Fn-2. То есть, следующее число получается как сумма двух предыдущих. Первые два числа равны 1, затем 2(1+1), затем 3(1+2), 5(2+3) и так далее: 1, 1, 2, 3, 5, 8, 13, 21.... Напишите функцию fib(n), которая возвращает n-е число Фибоначчи.

```
function fib(n) { /* ваш код */ }

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77)); // 5527939700884757
```

2) Существует формула Бине, согласно которой Fn равно ближайшему целому для (ϕ^n)/√5, где ϕ=(1+√5)/2 — золотое сечение. Напишите функцию fib_b(n), которая будет вычислять Fn, используя эту формулу.

3) Проверьте равенство полученных значений

Объекты как ассоциативные массивы

Объекты в JavaScript сочетают в себе два важных функционала.

Во-первых, они обладают функционалом для объектно-ориентированного программирования.

Но их сущность — ассоциативный массив: структура, пригодная для хранения любых данных. В этой главе мы рассмотрим использование объектов именно как массивов.

Ассоциативный массив — структура (или тип) данных, в которой можно хранить любые данные в формате ключ-значение.

Другое название для ассоциативного массива — «словарь» или «хэш». Одному ключу соответствует единственное значение, которое можно быстро прочитать, записать или удалить.

Создание объектов

Пустой объект (пустой ассоциативный массив) может быть создан одним из двух синтаксисов:

```
1. o = new Object();
2. o = {};   // пустые фигурные скобки
```

Обычно все пользуются синтаксисом (2), т.к. он короче.

Объект может содержать в себе любые значения, которые называются свойствами объекта. Доступ к свойствам осуществляется по имени свойства (иногда говорят «по ключу»).

Например, создадим объект person для хранения информации о человеке:

```
var person = {}; // пока пустой
```

Операции с объектом

Основные операции с объектами — это:

1) Присвоение свойства по ключу.
2) Чтение свойства по ключу.
3) Удаление свойства по ключу.

Для обращения к свойствам используется запись «через точку», вида объект.свойство:

```
var person = {};
// 1. присвоение
person.name = 'Вася'; // запишем свойство с именем 'name' и значением 'Вася'
person.age = 25;  // запишем свойство с именем 'age' и значением 25
// 2. чтение
alert(person.name + ': ' + person.age); // вывести значения
// 3. удаление
delete person.name; // удалить значение с именем name
```

Проверка существования свойства с опред. ключом.

Например, есть объект person, и нужно проверить, существует ли в нем свойство age.

Для проверки существования есть оператор in:

Его синтаксис: "prop" in obj, причем имя свойства — в виде строки, например:

```
var person = { };

if ("age" in person) {

 alert("Свойство age существует!");
```

Другой способ — сравнение значения с undefined. Дело в том, что в JavaScript можно обратиться к любому свойству объекта, даже если его нет. Ошибки не будет. Но если свойство не существует, то вернется специальное значение undefined. Таким образом мы можем легко проверить существование свойства — получив его и сравнив с undefined:

```
var person = { name: "Василий" };

alert(person.lalala === undefined); // true, свойства нет

alert(person.name === undefined); // false, свойство есть.
```

Сравнение с undefined не работает, если значение свойства равно undefined.

Доступ через квадратные скобки

Существует альтернативный синтаксис работы со свойствами, использующий квадратные скобки объект['свойство']:

```
var person = {};
person['name'] = 'Вася'; // то же что и person.name
alert(person['name']);
delete person['name'];
```

В квадратные скобки можно передать переменную:

obj[key] использует значение переменной key в качестве имени свойства:

```
var person = { age: 25 };
var key = 'age';
alert(person[key]);  // возвратит person['age']
```

Записи person['age'] и person.age идентичны. С другой стороны, если имя свойства хранится в переменной (var key = "age"), то единственный способ к нему обратиться — квадратные скобки person[key].

Обращение через точку используется, если мы на этапе написания программы уже знаем название свойства. А если оно будет определено по ходу выполнения, например, введено посетителем и записано в переменную, то единственный выбор — квадратные скобки.

Объявление со свойствами

Объект можно заполнить значениями при создании, указав их в фигурных скобках: { ключ1: значение1, ключ2: значение2, ... }. Такой синтаксис называется литеральным (оригинал - literal).

Следующие два фрагмента кода создают одинаковый объект:

```
var menuSetup = {
    width:  300,
    height: 200,
    title: "Menu"
};
```

```
var menuSetup = {};
menuSetup.width = 300;
menuSetup.height = 200;
menuSetup.title = 'Menu';
```

Названия свойств можно перечислять в кавычках или без, если они удовлетворяют ограничениям для имён переменных.

Например:

```
var menuSetup = {
    width:  300,
    'height': 200,
    "мама мыла раму": true
};
```

Вложенные объекты в объявлении

Значением свойства может быть объект:

```
var obj = {
  name:  "Первый",
  child: {
     name:  "Второй"
  }
}

alert( obj.name ) // Первый
alert( obj.child.name ) // Второй
```

Здесь значением свойства child является объект { name: "Второй" }.

Перебор свойств и значений

Для перебора всех свойств из объекта используется цикл по свойствам for..in. Это специальная синтаксическая конструкция, которая работает не так, как обычный цикл for (i=0;i<n;i++).

Синтаксис:

```
for (key in obj) {
  /* ... делать что-то с obj[key] ... */
}
```

При этом в key будут последовательно записаны имена свойств. Например:

```
var menu = {
    width:  300,
    height: 200,
    title: "Menu"
};

for (var key in menu) {
    // этот код будет вызван для каждого свойства объекта
    // ..и выведет имя свойства и его значение

    alert("Ключ: " + key + " значение:" + menu[key]);
}
```

Конечно, вместо key может быть любое другое имя переменной.

Количество свойств в объекте

Одного метода, который вернул бы количество свойств нет. Кросс-браузерный способ — это сделать цикл по свойствам и посчитать:

```
function getKeysCount(obj) {
  var counter = 0;
  for (var key in obj) {
    counter++;
  }
  return counter;
}
```

А вот так выглядит функция проверки на «пустоту»:

```
function isEmpty(obj) {
  for (var key in obj) {
    return false; // если цикл хоть раз сработал, то объект не пустой => false
  }
  // дошли до этой строки - значит цикл не нашёл ни одного свойства => true
  return true;
}
```

Порядок перебора свойств

Упорядочены ли свойства в объектах? В теории, т.е. по спецификации — нет. На практике, всё сложнее.

Браузеры придерживаются важного соглашения о порядке перебора свойств. Оно, хоть и не прописано в стандарте, но достаточно надежно, т.к. много существующего кода зависит от него.

1) **Если свойства в объекте строковые, то такой объект упорядочен. Свойства будут перебираться в порядке их присвоения**:

2) **Если ключи не строковые, некоторые браузеры (IE9+, Chrome, Opera) выведут ключи не в том порядке, в котором они заданы, а в отсортированном порядке**

```
var codes = {
  "7": "Россия",
  "38": "Украина",
  "1": "США"
};

for (var key in codes) {
  alert(key + ": "+codes[key]); // 1, 7, 38 в IE9+, Chrome, Opera
}
```

Передача по ссылке

1) **Обычные значения: строки, числа, булевы значения, null/undefined копируются «по значению».**

Это означает, что если в переменной message хранится значение "Привет", и её скопировали phrase = message, то значение копируются, и у нас будут две переменные, каждая из которых хранит значение: "Привет".

2) **Объекты присваиваются и передаются «по ссылке». В переменной хранится не сам объект, а ссылка на его место в памяти.**

3) **При копировании переменной с объектом — копируется эта ссылка, а объект по-прежнему остается в единственном экземпляре.**

4) **Так как объект всего один, то в какой бы переменной его не меняли — это отражается на других:**

```
var user = { name: 'Вася' };
var admin = user;
admin.name = 'Петя'; // поменяли данные через admin

alert(user.name); // 'Петя', изменения видны в user
```

Задача 7

1) Создайте функцию multiplyNumeric, которая получает объект и умножает все численные свойства на заданное значение. Например:

```
// до вызова
var menu = {
    width: 200,
    height: 300,
    title: "My menu"
};
multiplyNumeric(menu,int);
// после вызова
menu = {
    width: 400,
    height: 600,
    title: "My menu"
};
```

2) Оформите результат в виде таблицы(имя: значение).

Массивы с числовыми индексами

Массив с числовыми индексами - это коллекция данных, которая хранит сколько угодно значений, причем у каждого значения - свой уникальный номер. В массиве может храниться любое число элементов любого типа. В том числе, строки, числа, объекты

Синтаксис для создания нового массива — квадратные скобки со списком элементов внутри.

```
var arr = [];

var arr = ["Первый", "Второй", "Третий"];
```

Элементы нумеруются, начиная с нуля. Чтобы получить нужный элемент из массива — указывается его номер в квадратных скобках:

```
var arr = ["Первый", "Второй", "Третий"];
alert(arr[0]); // Первый
alert(arr[1]); // Второй
alert(arr[2]); // Третий
```

Элемент можно всегда заменить. Общее число элементов, хранимых в массиве, содержится в его свойстве length:

```
arr[2] = '_Третий'; // теперь ["Первый", "Второй", "_Третий"]
alert(arr.length); // 3
```

Методы pop/push, shift/unshift

Одно из применений массива — это очередь. В классическом программировании так называют упорядоченную коллекцию элементов, такую что элементы добавляются в конец, а обрабатываются — с начала.

В реальной жизни эта структура данных встречается очень часто. Например, **очередь** сообщений, которые надо отослать.

Очень близка к очереди еще одна структура данных: **стек**. Это такая коллекция элементов, в которой новые элементы добавляются в конец и берутся с конца.

Например, стеком является колода карт, в которую новые карты кладутся сверху, и берутся — тоже сверху.

Для того, чтобы реализовывать эти структуры данных, и просто для более удобной работы с началом и концом массива существуют специальные методы.

Конец массива

1) **pop**

Удаляет последний элемент из массива и возвращает его:

```
var arr = ["Первый", "Второй", "Третий"];
alert( arr.pop() ); // удалили "Третий"
alert(arr); // "Первый", "Второй"
```

2) **push**

Добавляет элемент в конец массива:

```
var arr = [ "Первый", "Второй"];
arr.push("Третий");
alert(arr); // "Первый", "Второй", "Третий"
```

Является полным аналогом arr[arr.length] = ...

Начало массива

1) **shift -** Удаляет из массива первый элемент и возвращает его:

```
var arr = ["Первый", "Второй", "Третий"];
alert( arr.shift() ); // удалили "Первый"
alert(arr); // "Второй", "Третий"
```

2) **unshift** - Добавляет элемент в начало массива:

```
var arr = ["Второй", "Третий"];
arr.unshift("Первый");
alert(arr); // "Первый", "Второй", "Третий"
```

Методы push и unshift могут добавлять сразу по несколько элементов:

```
var arr = ["Третий"];
fruits.push("Четвертый", "Пятый");
fruits.unshift("Первый", "Второй");
// результат: ["Первый", "Второй", "Третий", "Четвертый", "Пятый"]
```

Внутреннее устройство массива

Массив — это объект, где в качестве ключей выбраны цифры, с дополнительными методами и свойством length.

В функцию массив передаётся по ссылке:

```
function rem(arr) {
  arr.pop();
}
var arr = ["Первый", "Второй", "Третий", "Четвертый", "Пятый"]
alert(arr.length); // 5
rem(arr);
rem(arr);
alert(arr.length); // 3, в функцию массив не скопирован, а передана ссылка
```

Ещё одно следствие — можно присваивать в массив любые свойства.

```
var arr = []; // создать массив
arr[99999] = 5; // присвоить свойство с любым номером
arr.name = 'Массив'; // назначить свойство со строковым именем
```

Как правило, нет причин использовать массив как обычный объект, хотя технически это и возможно.

Влияние на быстродействие

Методы push/pop выполняются быстро, а shift/unshift — медленно.

Чтобы понять, почему работать с концом массива быстрее, чем с его началом, разберём происходящее подробнее.

Операция shift выполняет два действия:

1) Удалить элемент в начале.
2) Переинднксировать содержимое всех остальных ячеек
3) Обновить внутреннее свойство length.

При этом, так как все элементы находятся в своих ячейках, просто очистить ячейку с номером 0 недостаточно. Нужно еще и переместить оставшиеся ячейки на 1 к началу. Чем больше элементов в массиве, тем дольше их перемещать.Аналогично работает unshift: чтобы добавить элемент в начало массива, нужно сначала перенести все существующие.

У методов push/pop таких проблем нет. Для того, чтобы удалить элемент, метод pop очищает ячейку и укорачивает length.

Перебор элементов

Для перебора элементов обычно используется цикл:

```
var arr = ["Первый", "Второй", "Третий"];

for (var i=0; i<arr.length; i++) {

  alert( arr[i] );

}
```

Так как массив является объектом, то возможен и вариант for..in:

```
var arr = ["Первый", "Второй", "Третий"];

for (var key in arr) {

  alert( arr[key] ); // "Первый", "Второй", "Третий"

}
```

Недостатки этого способа:

1) Цикл for..in выведет все свойства объекта, а не только цифровые.
2) Цикл for (var i=0; i<arr.length; i++) в современных браузерах выполняется в 10-100 раз быстрее.Т.к. браузер особым образом оптимизирует такие циклы.

Особенности работы length

Встроенные методы для работы с массивом автоматически обновляют его длину length.

Длина length — не количество элементов массива, а последний индекс + 1.

Это легко увидеть на следующем примере:

```
var arr = [];

arr[1000] = true;

alert(arr.length); // 1001
```

Вообще, если у вас элементы массива нумеруются случайно или с большими пропусками, то стоит подумать о том, чтобы использовать обычный объект.

Массивы предназначены именно для работы с непрерывной упорядоченной коллекцией элементов.

Используем length для укорачивания массива

При уменьшении length массив укорачивается. Причем этот процесс необратимый, т.е. даже если потом вернуть length обратно — значения не восстановятся:

```
var arr = [1, 2, 3, 4, 5];

arr.length = 2; // укоротить до 2 элементов

alert(arr); // [1, 2]


arr.length = 5; // вернуть length обратно, как было

alert(arr[3]); // undefined: значения не вернулись
```

Самый простой способ очистить массив — это arr.length=0.

Создание вызовом new Array

Существует еще один синтаксис для создания массива:

```
var arr = new Array("Яблоко", "Груша", "и т.п.");
```

Обычно new Array(элементы, ...) создаёт массив из данных элементов, но если у него один аргумент, и это число —то он создает массив без элементов, но с заданной длиной.

```
var arr = new Array(2,3); // создает массив [2, 3]

arr = new Array(2);

alert(arr[0]); // undefined, массив без элементов, длины 2
```

Получившийся массив ведёт себя так, как будто его элементы равны undefined.

Многомерные массивы

Массивы в JavaScript могут содержать в качестве элементов другие массивы. Это можно использовать для создания многомерных массивов, например матриц:

```
var matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
alert(matrix[1][1]); // 5
```

Object.keys(obj)

В JavaScript есть встроенный метод Object.keys(obj), который возвращает ключи объекта в виде массива. Он поддерживается везде, кроме IE<9:

```
var user = {
  name: "Guest",
  pass: '123'
}
var keys = Object.keys(user);
alert(keys); // name, pass
```

В более старых браузерах аналогом будет цикл:

```
var keys = [];

for(var key in user) keys.push(key);
```

Методы split, join

1) **split позволяет превратить строку в массив, разбив ее по разделителю s.** В примере ниже таким разделителем является строка из запятой и пробела.

```
var names ="Первый, Второй, Третий, Четвертый";

var arr = names.split(', ');

for (var i=0; i<arr.length; i++) {
  alert(arr[i] + ' элемент ');
}
```

2) **join(str)** берет массив и склеивает его в строку, используя str как разделитель.

```
var arr = ["Первый", "Второй", "Третий", "Четвертый"];

var str = arr.join(';');

alert(str); // Первый;Второй;Третий;Четвертый
```

Метод splice

Метод splice - умеет все: удалять элементы, вставлять элементы, заменять элементы - по очереди и одновременно.

arr.splice(index[, deleteCount, elem1, ..., elemN]) *Удалить deleteCount элементов, начиная с номера index, а затем вставить elem1, ..., elemN на их место.*

```
var arr = ["Первый", "Второй", "Третий"];
arr.splice(1, 1);  // удалить 1 элемент, начиная с позиции 1
alert(arr); //  осталось ["Первый", "Третий"]
```

Здесь продемонстрировано, как использовать splice для удаления одного элемента. Следующие за удаленным элементы сдвигаются, чтобы заполнить его место.

```
var arr = [Первый", "Второй", "Третий"];
arr.splice(0, 1);  // удалить 1 элемент, начиная с позиции 0
alert( arr[0] ); // "Второй" стал первым элементом
```

Следующий пример показывает, как заменять элементы:

```
var arr = ["Первый", "Второй", "Третий", "Четвертый"];
// удалить 3 первых элемента и добавить другие вместо них
arr.splice(0, 3, "_Первый", "_Второй")
alert( arr ) // теперь [ "_Первый", "_Второй", "Четвертый"]
```

Метод splice

Метод splice возвращает массив из удаленных элементов:

```
var arr = ["Первый", "Второй", "Третий", "Четвертый"];
// удалить 2 первых элемента
var removed = arr.splice(0, 2);
alert( removed ); // "Первый", "Второй" <-- array of removed elements
```

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

```
var arr = ["Первый", "Второй", "Третий"];
// с позиции 2
// удалить 0
// вставить "Второй_1", "Второй_2"
arr.splice(2, 0,"Второй_1", "Второй_2");
alert(arr); // "Первый", "Второй","Второй_1", "Второй_2","Третий"
```

Допускается использование отрицательного номера позиции, которая в этом случае отсчитывается с конца:

```
var arr = [1, 2, 5]
 // начиная с позиции индексом -1 (предпоследний элемент)
// удалить 0 элементов,// затем вставить числа 3 и 4
arr.splice(-1, 0, 3, 4);
alert(arr); // результат: 1,2,3,4,5
```

Метод slice

Метод slice(begin, end) копирует участок массива от begin до end, не включая end. Исходный массив при этом не меняется.

```
var arr = ["Первый", "Второй", "Третий", "Четвертый"];

var arr2 = arr.slice(1,3); // элементы 1, 2 (не включая 3)

alert(arr2); //"Второй", "Третий"
```

Аргументы ведут себя также, как и в строковом slice:

1) Если не указать end — копирование будет до конца массива:

```
var arr = ["Первый", "Второй", "Третий", "Четвертый"];

alert( arr.slice(1) ); // взять все элементы, начиная с номера 1
```

2) Можно использовать отрицательные индексы, они отсчитываются с конца:

```
var arr2 = arr.slice(-2); // копировать от 2го элемента с конца и дальше
```

3) Если вообще не указать аргументов — скопируется весь массив:

```
var fullCopy = arr.slice();
```

Сортировка, метод sort(fn)

Метод sort() сортирует массив на месте. Например:

```
var arr = [ 1, 2, 15 ];

arr.sort();

alert( arr );  // 1, 15, 2
```

Порядок стал 1, 15, 2. Это произошло потому, что sort сортирует, преобразуя элементы к строке. Поэтому и порядок у них строковый, ведь "2" > "15".

Внутренняя реализация метода arr.sort(fn) умеет сортировать любые массивы, если указать функцию fn от двух элементов, которая умеет сравнивать их.

Например, укажем эту функцию явно, отсортируем элементы массива как числа:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a < b) return -1;
}

var arr = [ 1, 2, 15 ];
arr.sort(compareNumeric);
alert(arr);  // 1, 2, 15
```

Алгоритм сортировки, встроенный в JavaScript, будет передавать ей для сравнения элементы массива. Функция должна возвращать:

- Положительное значение, если a > b,
- Отрицательное значение, если a < b,
- Если равны — не важно, что возвращать, их взаимный порядок не имеет значения.

reverse

Метод arr.reverse() меняет порядок элементов в массиве на обратный.

```
var arr = [1,2,3];

arr.reverse();

alert(arr); // 3,2,1
```

concat

Метод arr.concat(value1, value2, … valueN) создаёт новый массив, в который копируются элементы из arr, а также value1, value2, ... valueN.

```
var arr = [1,2];

var newArr = arr.concat(3,4);

alert(newArr); // 1,2,3,4
```

Если value — массив, то concat добавляет его элементы.

```
var arr = [1,2];

var newArr = arr.concat( [3,4], 5);// то же самое, что arr.concat(3,4,5)

alert(newArr); // 1,2,3,4,5
```

indexOf/lastIndexOf

Метод **arr.indexOf(searchElement[, fromIndex])** возвращает номер элемента searchElement в массиве arr или -1, если его нет.

Поиск начинается с номера fromIndex, если он указан. Если нет — с начала массива.

Для поиска используется строгое сравнение ===.

```
var arr = [ 1, 0, false ];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1
```

Как вы могли заметить, по синтаксису он полностью аналогичен методу indexOf для строк.

Метод **arr.lastIndexOf(searchElement[, fromIndex])** ищет справа-налево: с конца массива или с номера fromIndex, если он указан.

Задача 8

Анаграммы — слова, состоящие из одинакового количества одинаковых букв, но в разном порядке.

Например:

1) перемена - перенаем
2) парадокс - просадка - распадок
3) пасечник - песчаник - песчинка

Напишите функцию aclean(arr), которая возвращает массив слов, очищенный от анаграмм.

Например:

```
var arr = ["пасечник", "парадокс", "просадка", "песчаник", "ПЕРЕМЕНА", "перенаем", "песчинка"];

alert( aclean(arr) );

// "ПЕРЕМЕНА,парадокс,пасечник" или "распадок,песчаник,перенаем"
```

Из каждой группы анаграмм должно остаться только одно слово, не важно какое.

Дата и Время

Для работы с датой и временем в JavaScript используются объекты Date. Для создания нового объекта типа Date используется один из синтаксисов:

1) **new Date()** Создает объект Date с текущей датой и временем:

```
var now = new Date();
```

2) **new Date(milliseconds)** Создает объект Date, значение которого равно количеству миллисекунд (1/1000 секунды), прошедших с 1 января 1970 года GMT+0.

```
// 24 часа после 01.01.1970 GMT+0
var Jan02_1970 = new Date(3600*24*1000);
```

3) **new Date(datestring)** Если единственный аргумент - строка, используется вызов Date.parse для ее разбора.

4) new Date(year, month, date, hours, minutes, seconds, ms) Дату можно создать, используя компоненты в местной временной зоне. Для этого формата обязательны только первые два аргумента. Отсутствующие параметры, начиная с hours считаются равными нулю, а date — единице. Год year должен быть из 4 цифр, а отсчет месяцев month начинается с нуля 0

```
new Date(2011, 0, 1) // 1 января 2011, 00:00:00 в местной временной зоне
new Date(2011, 0) // то же самое, date по умолчанию равно 1
new Date(2011, 0, 1, 0, 0, 0, 0); // то же самое
```

Получение компонент даты

Для доступа к компонентам даты-времени объекта Date используются следующие методы:

1) **getFullYear(),getUTCFullYear()** Получить год(из 4 цифр)
2) **GetMonth(), getUTCMonth()** Получить месяц, от 0 до 11.
3) **getDate()** Получить число месяца, от 1 до 31.
4) **getHours(), getMinutes(), getSeconds(), getMilliseconds()** Получить соответствующие компоненты.
5) **getDay(), getUTCDay()** Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом от 0(воскресенье) до 6(суббота).

Методы без **UTC**, указанные выше, возвращают результат для местной временной зоны.

```
var date = new Date();
alert( date.getHours() ); // час в вашей зоне для даты date
```

Кроме описанных выше, существует два специальных метода без UTC-варианта:

6) **getTime()** Возвращает число миллисекунд, прошедших с 01.01.1970 00:00:00 UTC. Это то же число, которое используется в конструкторе new Date(milliseconds).
7) **getTimezoneOffset()** Возвращает разницу между местным и UTC-временем, в минутах.

Установка компонент даты

Следующие методы позволяют устанавливать компоненты даты и времени:

1) **setFullYear(year [, month, date])**
2) **setMonth(month [, date])**
3) **setDate(date)**
4) **setHours(hour [, min, sec, ms])**
5) **setMinutes(min [, sec, ms])**
6) **setSeconds(sec [, ms])**
7) **setMilliseconds(ms)**
8) **setTime(milliseconds) (устанавливает всю дату)**

Все они, кроме setTime(), обладают также UTC-вариантом, например: setUTCHours().

Автоисправление даты

Автоисправление - очень удобное свойство объектов Date. Оно заключается в том, что можно устанавливать заведомо некорректные компоненты (например 32 января), а объект сам себя поправит.

```
var d = new Date(2012, 0, 32); // 32 января 2012
alert(d); // 1 февраля 2012
```

Неправильные компоненты даты автоматически распределяются по остальным.

Например, нужно увеличить на 2 дня дату «28 февраля 2011». Может быть так, что это будет 2 марта, а может быть и 1 марта, если год високосный. Но нам обо всем этом думать не нужно. Просто прибавляем два дня. Остальное сделает Date:

```
var d = new Date(2011, 1, 28);
d.setDate( d.getDate() + 2 );
alert(d); // 2 марта, 2011
```

Также это используют для получения даты, отдаленной от имеющейся на нужный промежуток времени. Например, получим дату на 70 секунд большую текущей:

```
var d = new Date();
d.setSeconds( d.getSeconds()+70);
alert(d); // выведет корректную дату
```

Преобразование к числу, разность дат

Когда объект Date используется в числовом контексте, он преобразуется в количество миллисекунд:

```
alert( +new Date ) // +date то же самое, что: +date.valueOf()
```

Важный побочный эффект: даты можно вычитать, результат вычитания объектов Date — их временная разница, в миллисекундах.

Это используют для измерения времени и профилирования:

```
var start = new Date; // засекли время
for (var i=0; i<100000; i++) {
  var doSomething = i*i*i;
}
var end = new Date; // конец измерения
alert("Цикл занял " + (end-start) + " ms");
```

Форматирование

Встроенные в Date методы форматирования используются редко, и преимущественно, для отладки.

1) **toString(), toDateString(), toTimeString()** Возвращают стандартное строчное представление, не указанное в стандарте, а зависящее от браузера. Единственное требование - читаемость человеком. Метод toString возвращает дату целиком, toDateString() и toTimeString() - только дату и время соответственно.

```
var d = new Date();
alert( d.toString() ); // вывод, похожий на 'Wed Jan 26 2011 16:40:50 GMT+0300'
```

2) **toLocaleString(), toLocaleDateString(), toLocaleTimeString()** То же самое, но строка должна быть с учетом локальных настроек и языка посетителя.

```
var d = new Date();
alert( d.toLocaleString() ); // дата на языке посетителя
```

3) **toUTCString()** То же самое, что toString(), но дата в зоне UTC.

4) **toISOString()** Возвращает дату в формате ISO

```
var d = new Date();
alert( d.toISOString() ); // вывод, похожий на '2011-01-26T13:51:50.417Z'
```

Встроенные методы форматирования Date не допускают указание собственного формата.

Поэтому, как правило, любой вывод, кроме отладочного, форматируется своей, а не встроенной функцией.

Разбор строки, Date.parse

Все современные браузеры, включая IE9+, понимают даты в упрощённом формате ISO 8601 Extended. Этот формат выглядит так: YYYY-MM-DDTHH:mm:ss.sssZ. Для разделения даты и времени в нем используется символ 'T'. Часть ‘Z’ обозначает (необязательную) временную зону — она может отсутствовать, тогда зона UTC, либо может быть символ z — тоже UTC, или зона в формате +-hh:mm.

Также возможны упрощенные варианты, к примеру:

YYYY, YYYY-MM, YYYY-MM-DD

Метод Date.parse(str) разбирает строку str в таком формате и возвращает соответствующее ей количество миллисекунд. Если это невозможно, Date.parse возвращает NaN.

```
var msNoZone = Date.parse('2012-01-26T13:51:50.417'); // без зоны, значит UTC

alert(msNoZone); // 1327571510417 (число миллисекунд)

var msZ = Date.parse('2012-01-26T13:51:50.417z'); // зона z означает UTC
alert(msZ == msNoZone); // true, значение совпадает

var ms = Date.parse('2012-01-26T13:51:50.417-07:00'); //С таймзоной -07:00 GMT
alert(ms); // 1327611110417 (число миллисекунд)
```

Задача

Задача: вывести на странице три даты вида: (1) <текущий день> <текущий месяц>, (2) <текущий день + 3 дня> <текущий месяц>, (3) <следующий месяц> <текущий год>, причем названия месяцев должны выводиться на русском языке в первых двух случаях в Родительном падеже со строчной буквы, а в последнем пункте в Именительном падеже с заглавной буквы.

Т.е. вывод должен быть следующего вида:

1) Сегодня: 22 июля
2) Через три дня: 25 июля
3) Через месяц: Август 2011

Функция как объект

В JavaScript функция является значением, таким же как строка или число.

Объявление создает функцию и записывает ссылку на неё в переменную.

Как и любое значение, функцию можно вывести, вот так:

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

Здесь выводится не результат работы функции sayHi() , а сама функция, т.е. ее код.

Функция — не просто значение, это объект.

В него можно даже записать свойство:

```
function sayHi() { }
sayHi.test = 5;
alert(sayHi.test); // 5
```

Копирование функций

Функцию можно скопировать в другую переменную.

```
function sayHi(num) {
  alert('Функция, ' + num);
}
var func = sayHi;
alert(func); // выведет код функции
```

При этом, так как функция — это объект, то и копируется она «по ссылке».

То есть, **сама функция лежит где-то в памяти, а переменная содержит «адрес», где она находится. При присваивании func = sayHi копируется этот адрес, обе переменные начинают указывать на одно и то же «место в памяти», где находится функция.** После копирования ее можно вызывать:

```
function sayHi(num) {
  alert('Функция ' + num);
}
var func = sayHi;
func('1'); // выведет 'Функция 1'
sayHi('2'); // и так по-прежнему работает: 'Функция 2'
```

Заметим ещё раз, что обращение вида alert(func) отличается от alert(func()). В первом случае выводится функция (её код), во втором — эта функция вызывается, и выведен будет уже её результат.

Именованные функции

Объявление функции, о котором мы говорили все время до этого, называется в спецификации языка **именованными**.

При объявлении функции создаётся переменная со значением-функцией

Другими словами, объявление function func(параметры) { код } говорит интерпретатору: «создай переменную func, и положи туда функцию с указанными параметрами и кодом».

При этом func — на самом деле не «имя функции». Оно совсем никак не привязано к функции! Это название переменной, в которую будет помещена функция, и из которой она может быть затем удалена, скопирована в другую переменную, и в результате её как func вызвать будет нельзя:

```
function func() { alert(1); }
var g = func; // скопировали
func = null;  // поменяли значение
g(); // работает, теперь функция в g, а в func ничего нет
func(); // вызываем null()? ошибка!
```

Невозможно иметь функцию и переменную с одинаковым именем

```
function f() { } // объявить переменную f и записать в нее функцию
var f = 5; // объявить переменную f (а она уже объявлена) и присвоить 5
alert(f); // результат: 5
```

Время создания функции

Функции, объявленные как именованные, создаются интерпретатором до выполнения кода.

Перед тем, как выполнять первую строку, интерпретатор сканирует код, ищет в нём именованные функции и обрабатывает их. Поэтому их можно вызвать до объявления, например:

```
sayHi("1");
function sayHi(name) {
  alert("Функция " + name);
}
```

Этот код будет работать, т.к. объявление function sayHi обрабатывается и функция создаётся до выполнения первой строчки кода.

Время создания функции

Попробуем, в зависимости от условия, объявить функцию по-разному:

```
var status = 0;
if (status > 0) {
  function getStatus() {  alert('Не ноль');  }
} else {
  function getStatus() {  alert('Ноль'); }
}
getStatus(); //Ноль
```

Именованные функции обрабатываются до выполнения первой строчки кода. Браузер сканирует код и создает из таких объявлений функции. При этом второе объявление перезапишет первое. Дальше, во время выполнения объявления игнорируются (они уже обработаны), как если бы код был таким:

```
function getStatus() {   alert('Не ноль');  }
function getStatus() {  alert('Ноль'); }
var status = 0;
if (status > 0) {
} else {
}
getStatus(); //Ноль
```

Как видно, конструкция if здесь ни на что не влияет. По-разному объявить функцию, в зависимости от условия, не получилось.

Анонимная функция

Существует альтернативный синтаксис для создания функций, который решает проблемы с «условным» объявлением. **Функцию можно создать и присвоить переменной как самое обычное значение.** Такое объявление называется **анонимная функция** и выглядит так:

```
var f = function(параметры) {  // тело функции  };
```

Такую функцию можно объявить в любом выражении, там где допустимо обычное значение. **В отличие от именованных функций, которые создаются заранее, до выполнения кода, анонимная функция создается, когда до нее доходит выполнение. Поэтому и пользоваться ими можно только после объявления.**

```
getStatus(); // <-- работать не будет, функции еще нет
var getStatus = function() {  alert(1)  };
```

Благодаря этому свойству анонимные функции можно (и даже нужно) использовать для условного объявления функции:

```
var status = prompt('Введите число');
var getStatus;
if (status > 0) {
  getStatus = function() {  alert('Не ноль');  }
} else {
  getStatus = function() {  alert('Ноль');  }
}
getStatus(); // запустит ту функцию, которая присвоена в if
```

Функция с вызовом «на месте»

Например, наш скрипт задаёт переменные a, b, и если на странице они уже используются — будет конфликт.

Чтобы его не было, нашему скрипту нужна своя отдельная область видимости, в которой он будет работать.

Для этого используют функции с вызовом «на месте». **Такая функция объявляется — и тут же вызывается**, вот так:

```
(function() {
  var a = 1 , b = 2; // переменные для нашего скрипта
  // код скрипта
})();
```

Теперь внутренние переменные скрипта стали локальными. Даже если в другом месте страницы объявлена своя переменная a — проблем не будет. Наш скрипт теперь имеет свою, изолированную область видимости.

У функции, которая объявлена таким образом, и тут же вызвана, нет имени. Она никуда не сохраняется, и поэтому не может быть вызвана второй раз. Но в данном случае это и не нужно, ведь задача такой функции обёртки — создать отдельную область видимости для скрипта, что она и сделала.

Эта практика широко используется в библиотеках JavaScript, чтобы временные переменные и функции, которые используются при инициализации, не конфликтовали с внешним кодом.

Зачем скобки вокруг функции?

В примерах выше вокруг function() {...} находятся скобки. Если записать без них — такой код вызовет ошибку:

```
function() {
 // syntax error
}();
```

Эта ошибка произойдет потому, что браузер, видя ключевое слово function в основном потоке кода, попытается объявить именованную функцию, а здесь даже имени нет.

Дело в том, что «на месте» разрешено вызывать только анонимные функции.

1) Если браузер видит function в основном потоке кода - он считает, что это именованная функция.

2) Если же function идёт в составе более сложного выражения, то он считает, что это анонимная функция.

Для этого и нужны скобки - показать, что у нас анонимная функция, которую по правилам JavaScript можно вызвать «на месте».

Скобки не нужны, если это и так анонимная функция, например в таком вызове:

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

Именованные функциональные выражения

Именованное функциональное выражение - расширение анонимной функции, позволяющее указать имя, действительно привязанное к функции.

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

Проще говоря, это анонимная функция с дополнительным именем (в примере выше getName). Имя функционального выражения (getName) имеет особый смысл. **Оно доступно только изнутри самой функции**. Такое имя жёстко привязано к функции.

```
var f = function getName(name) {
  alert(getName); // изнутри функции - видно (выведет код функции)
}

alert(getName); // снаружи - не видно (ошибка: undefined variable 'getName')
```

Используется **для единственной цели — позволить функции вызвать саму себя**.

```
var f = function factorial(n) {
  return n ? n*factorial(n-1) : 1;
}
var g = f;  // скопировали ссылку
            // на функцию-факториал в g
f = function() { return 0; }; // перезаписали f
alert( g(5) ); // 120, работает!
```

```
var f = function(n) {
  return n ? n*f(n-1) : 1;
}
var g = f;
f = null;
alert( g(5) ); // ошибка, т.к. g попытается
//вызвать f(n-1), а f перезаписана
```

Лексическое окружение

В JavaScript все переменные и фунции являются свойствами специального объекта, который называется Лексическое окружение.

Лексическое окружение верхнего уровня называют глобальным объектом. В браузере этот объект доступен как window, в других окружениях, например Node.JS, он может быть скрыт от программиста.

Присваивая или читая переменную вне функций, мы фактически работаем с этим объектом.

```
var a = 5;   // создаёт и присваивает свойство window.a

alert(window.a); // 5
```

Свойства и функции, которые находятся в window, называют глобальными.

Порядок создания глобальных переменных и функций

Выполнение скрипта происходит в две фазы. На первой фазе происходит инициализация, подготовка к запуску. На второй — собственно, выполнение.

Во время инициализации скрипт сканируется на предмет объявления именованных функций, а затем — на предмет объявления переменных var. Каждое такое объявление добавляется в лексическое окружение.

Функции создаются сразу работающие, а переменные равны undefined.

Затем код начинает выполняться, и уже на фазе выполнения происходит присваивание (=) значений переменным.

Локальные переменные

При запуске функция создает свой объект лексического окружения(LexicalEnvironment), и записывает в него аргументы, локальные переменные и т.п.

Этот процесс полностью аналогичен описанному выше, но, в отличие от window, объект лексического окружения для функции — внутренний, он скрыт от прямого доступа.

```
function getName(name) {
  var phrase = "Функция " + name;
  alert(phrase);
}
getName('1'); //Функция 1
```

1. При вызове функции, до выполнения первой строчки её кода, интерпретатор создает пустой объект лексического окружения и заполняет его. В данном случае туда попадает аргумент name и единственная переменная phrase.
2. Функция выполняется. Происходит присвоение локальной переменной phrase, которое является по сути присвоением свойству LexicalEnvironment.phrase нового значения.
3. В конце выполнения, объект с переменными обычно выбрасывается и память очищается.

Доступ ко внешним переменным

Из функции мы можем обратиться не только к локальной переменной, но и к внешней. При этом интерпретатор сначала пытается найти ее в текущем объекте с переменными, а затем — ищет ее во внешнем LexicalEnvironment. В данном случае им является window, т.е. выводится window.a.

```
var a = 5;
function f() {
  alert(a);  // 5
}
```

1) Каждая функция при создании получает ссылку [[Scope]] на объект с переменными, в контексте которого была создана.

2) При запуске функции создается новый объект с переменными. В него копируется ссылка на внешний объект из [[Scope]].

3) При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом — по этой ссылке. Благодаря этому в функции доступны внешние переменные.

Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции. Как правило, это делают для вспомогательных операций.

Вложенные функции могут быть объявлены и как именованные и как анонимные. Вложенные функции обрабатываются в точности так же, как и глобальные. Единственная разница — они создаются в объекте переменных внешней функции, а не в window.

Вложенная функция имеет доступ к внешним переменным через свойство [[Scope]].

1) При создании она получает свойство [[Scope]], указывающее на объект переменных, в котором она была создана.

2) При запуске она будет искать переменные сначала у себя, потом во внешнем объекте переменных (по [[Scope]]), затем в более внешнем, и так далее.

```
function showName(object) {
  var message = makeMessage(object);
  alert( message );
  function getName() {  return  object.name;  }
  function makeMessage(object) { return  'Получен объект:  ' + getName(object);  }
}
showName({
  name: 'Объект 1'
}); // Получен объект:  Объект 1
```

Вложенную функцию можно возвратить.

```
function showName(object) {
  return function() {
    var message = makeMessage(object);
    alert( message );
  };
  function getName() {  return  object.name;  }
  function makeMessage(object) { return  'Получен объект:  ' + getName(object);  }
}
var objectName = showName({ name: 'Объект 1' });
objectName(); // эта функция может быть вызвана позже
```

В реальной жизни это нужно, чтобы вызвать получившуюся функцию позже, когда это будет нужно. Например, при нажатии на кнопку.

Возвращаемая функция при запуске будет иметь полный доступ к аргументам внешней функции, а также к другим вложенным функциям makeMessage и getName, так как при создании она получает ссылку [[Scope]], которая указывает на текущий LexicalEnvironment. Переменные, которых нет в ней, например, object, будут взяты из него. Функция makeMessage тоже будет взята из внешнего объекта переменных. Порядок поиска для функций — совершенно такой же, как и для других переменных.

Внешний LexicalEnvironment, в свою очередь, может ссылаться на ещё более внешний LexicalEnvironment. В данном случае им будет window, но возможны и более длинные цепочки, при более глубокой вложенности функций.

Задача 9

Напишите функцию sum, которая работает так: pow(a)(b) = a*b.

Например:

sum(1)(2) = 2

sum(5)(-1) = -5

Еще раз про LexicalEnvironment

1) Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создает новый такой объект.
2) На верхнем уровне роль LexicalEnvironment играет «глобальный объект», в браузере это window.
3) При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана (кроме new Function).
4) При запуске функции её LexicalEnvironment ссылается на внешний, сохраненный в [[Scope]]. Переменные сначала ищутся в своём объекте, потом — в объекте по ссылке и так далее, вплоть до window.

[[Scope]] для new Function

Есть одно исключение из общего правила присвоения [[Scope]].

При создании функции с использованием new Function, её свойство [[Scope]] ссылается не на текущий LexicalEnvironment, а на window.

Следующий пример демонстрирует как функция, созданная new Function, игнорирует внешнюю переменную a и выводит глобальную вместо нее.

```
var a = 1;
function getFunc() {
  var a = 2;
  var func = function() { alert(a); };
  return func;
}

getFunc()();
// 2, из LexicalEnvironment функции getFunc
```

```
var a = 1;
function getFunc() {
  var a = 2;
  var func = new Function('', 'alert(a)');
  return func;
}

getFunc()(); // 1, из window
```

Замыкание функции

Пока доступна хоть одна внутренняя функция —- доступнен и объект LexicalEnvironment, на который она ссылается, через [[Scope]].

Функция «замыкает» на себе вышестоящие области видимости и не дает им быть удаленными если к ним теряется доступ, а основное применение замыканий в программировании — хранение данных между вызовами функций и инкапсуляция данных.

«Замыканием» функции называются вся цепочка объектов переменных, которая при этом образуется.

Вот в этом случае лексическое окружение, включая переменную a, будет сохранено:

```
function f() {
  var value = 123;
  function g() { } // g видна только изнутри
}
f();

//После окончания работы f() она станет
//недоступной для вызовов и будет убрана
//из памяти вместе с другими
//локальными переменными.
```

```
function f() {
  var a = 123;
  function g() { }
  return g;
}
var g = f();

// функция g будет доступна и
//сохранит ссылку на объект переменных
```

Хранение данных в замыкании

```
// makeCounter создает функцию,
// которая считает свои вызовы:

function makeCounter() {
  var currentCount = 0;

  return function() {
    currentCount++;
    return currentCount;
  };
}

var counter = makeCounter();

// каждый вызов увеличивает счетчик
counter();
counter();
alert( counter() ); // 3
```

В примере ниже makeCounter создает функцию, которая считает свои вызовы.

Хранение текущего числа вызовов осуществляется в переменной currentCount внешней функции.

При этом, так как каждый вызов makeCounter создает новый объект переменных, то все создаваемые функции-счетчики взаимно независимы.

```
var c1 = makeCounter();

var c2 = makeCounter();

alert( c1() ); // 1
alert( c2() ); // 1, счетчики независимы
```

Хранение данных в замыкании

```
function makeCounter() {
  var currentCount = 0;

  return function(newCount) {
    if (newCount !== undefined) {
      currentCount = +newCount;
      return currentCount;
    }

    currentCount++;
    return currentCount;
  };
}

var counter = makeCounter();

alert( counter() ); // 1
alert( counter(3) ); // 3
alert( counter() ); // 4
```

Добавим счётчику аргумент, который, если передан, устанавливает значение.

Здесь для проверки первого аргумента использовано сравнение с undefined, так что вызовы counter(undefined) и counter() сработают идентично, как будто аргументов нет.

Методы объектов

До этого мы говорили об объекте лишь как о хранилище значений. Теперь пойдём дальше и поговорим о добавлении в объекты собственных функций (методов), а также о конструкторах — функциях, создающих объекты.

При объявлении в объект можно записать функцию. Она становится его методом. Можно создать метод и позже, явным присвоением:

```
var user = {
  name: 'Василий',
  // метод
  sayHi: function() {
    alert('Привет!');
  }
};
// Вызов метода
user.sayHi();
```

```
var user = {
  name: 'Василий'
};
user.sayHi = function() {
  alert('Привет!');
};
// Вызов метода:
user.sayHi();
```

Доступ к объекту через this

Для полноценной работы метод должен иметь доступ к данным объекта. В частности, вызов user.sayHi() может захотеть вывести имя пользователя.

Для доступа к объекту из метода используется ключевое слово this. Значением this является объект, в контексте которого вызван метод, например:

```
var user = {
  name: 'Василий',
  sayHi: function() {
    alert( this.name );
  }
};
user.sayHi();
```

```
var user = {
  name: 'Василий',
  sayHi: function() {
    showName(this); // передать текущий объект в showName
  }
};
function showName(obj) {
  alert( obj.name );
}
user.sayHi();
```

Здесь при выполнении функции user.sayHi() в this будет храниться ссылка на текущий объект user.

В данном случае вместо this можно было бы и использовать и переменную: alert(user.name), но объект user может быть куда-то передан, переменная user перезаписана и т.п. **Использование this гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.**

Хранение приватных данных (*)

```
function makeCounter() {
  var currentCount = 0;

  return {
    getNext: function() {
      return ++currentCount;
    },

    set: function(value) {
      currentCount = value;
    },

    reset: function() {
      this.set(0);
    }
  };
}
var counter = makeCounter();

alert( counter.getNext() ); // 1
alert( counter.getNext() ); // 2
counter.reset();
alert( counter.getNext() ); // 1
```

Можно пойти дальше и возвратить полноценный объект с функциями управления счетчиком:

1) getNext — получить следующее значение, то, что раньше делал вызов counter().
2) set(value) — поставить значение.
3) reset() — обнулить счётчик.

Теперь counter — объект с методами, которые при работе используют currentCount. Никак иначе, кроме как через эти методы, к currentCount получить доступ нельзя.

Приём проектирования «Модуль»

```
function makeCounter() {
  var currentCount = 0;

  // возвращаемся к функции
  function counter() {
    return ++currentCount;
  }
  // ...и добавляем ей методы!
  counter.set = function(value) {
    currentCount = value;
  };
  counter.reset = function() {
    this.set(0);
  };
  return counter;
}
var counter = makeCounter();

alert( counter() ); // 1
alert( counter() ); // 2
counter.reset();
alert( counter() ); // 1
```

Приём программирования «модуль» имеет громадное количество вариаций. Его цель — объявить функции так, чтобы ненужные подробности их реализаций были скрыты. В том числе, временные переменные, константы, вспомогательные мини-функции и т.п.

1) Создаётся функция-обёртка, которая выполняется «на месте».

2) Внутрь этой функции пишутся локальные переменные и функции, которые пользователю модуля не нужны, но нужны самому модулю. Они будут доступны только изнутри.

3) Те функции, которые нужны пользователю, возвращаются во внешнюю область видимости. Если функция одна — это можно сделать явным возвратом return

Все функции модуля будут через замыкание иметь доступ к другим переменным и функциям. Но снаружи программист, использующий модуль, может обращаться напрямую только к тем, которые экспортированы.

Статические переменные

Статическая переменная функции — это такая, которая сохраняет значение между вызовами.

Такие переменные есть во многих языках. В JavaScript они не реализованы синтаксически, но можно организовать аналог.

Функция — это объект. К ней можно добавить любые свойства и использовать их как статические

Отличие статической переменной — в том, что она неразрывно привязана к функции, поэтому:

1) Если передать функцию в другую часть скрипта — статическая переменная будет за ней.

2) Пока жива функция — жива и переменная. Не может оказаться так, что количество вызовов есть, а функции — нет.

```
function sayHi() {
  sayHi.count++;
  alert("Привет " + sayHi.count);
}
sayHi.count = 0; // начальное значение
sayHi(); // Привет 1
sayHi(); // Привет 2
```

Используя эту особенность функций, можно реализовать статические переменные, присваивая значение в свойства функции как объекта.

Статическая переменная — общедоступна. К ней имеет доступ любой, у кого есть объект функции.

Функция-конструктор, «new»

Обычный синтаксис {...} позволяет создать один объект. Но зачастую нужно создать много однотипных объектов.

Для этого используют функции, запуская их при помощи специального оператора new.

Конструктором становится любая функция, вызванная через new.

```
function Animal(name) {
  this.name = name;
  this.canWalk = true;
}

var animal = new Animal("ёжик");
```

```
//результат выполнения
animal = {
  name: "ёжик",
  canWalk: true
}
```

Любую функцию можно вызвать при помощи new. При этом она работает несколько иным образом, чем обычно:

1) Автоматически создается новый, пустой объект.
2) Специальное ключевое слово this получает ссылку на этот объект.
3) Функция выполняется. Как правило, она модифицирует this, добавляет методы, свойства.
4) Возвращается this.

Создание методов в конструкторе

Использование функций для создания объекта дает большую гибкость. Можно передавать функции свойства создаваемого объекта и параметры, определяющие как его создавать.

Например, функция User(name) создает объект с заданным значением свойства name и методом sayHi:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert("Моё имя: " + this.name);
  };
}

var ivan = new User("Иван");
/* Объект ivan имеет вид:
 {
   name: "Иван",
   sayHi: функция, обращение к имени идёт через this.name
 }
*/
ivan.sayHi(); // Моё имя: Иван
```

Приватные свойства (*)

Локальные переменные функции-конструктора, с одной стороны, доступны вложенным функциям, с другой — недоступны снаружи. В объектно-ориентированном программировании это называется «приватный (private) доступ».

Например, в коде ниже к name имеет доступ только метод say. Со стороны объекта, после его создания, больше никто не может получить name.

```
function User(name) {
  this.say = function(phrase) {
    alert(name + ' сказал: ' + phrase);
  };
  this.upperCaseName = function() {
    name = name.toUpperCase(); // <-- изменяет name из User
  };
}

var user = new User('Вася');
user.upperCaseName();

user.say("Да здравствует ООП!") // ВАСЯ сказал: Да здравствует ООП!
```

Если бы name было свойством this.name — можно было бы получить его как user.name, а тут — локальная переменная. Все переменные конструктора User становятся приватными, так как доступны только через замыкание, из внутренних функций.

Контекст this в деталях

Значение this в JavaScript не зависит от объекта, в котором создана функция. Оно определяется во время вызова.

Любая функция может иметь в себе this. Совершенно неважно, объявлена она в объекте или вне него.

Значение this называется контекстом вызова и будет определено в момент вызова функции.

Например: такая функция вполне допустима:

```
function sayHi() {

 alert( this.firstName );

}
```

Эта функция ещё не знает, каким будет this. Это выяснится при выполнении программы.

Есть несколько правил, по которым JavaScript устанавливает this.

Вызов в контексте объекта

```
var user = {
  name: "Пользователь"
};
function func() {
  alert( this.name );
}

user.getName = func;
user.getName();  // this = user
```

Самый распространенный случай — когда функция объявлена в объекте или присваивается ему.

При вызове функции как метода объекта, через точку или квадратные скобки — функция получает в this этот объект.

В данном случае user.getName() присвоит this = user.

```
var user = { name: "Пользователь" };
var admin = { name: "Админ" };

function func() {
  alert( this.name );
}
// присвоим одну функцию в свойства
// двух разных объектов user и admin
user.a = func;
admin.b = func;

user.a(); // Пользователь
admin['b']() // Админ
```

Если одну и ту же функцию запускать в контексте разных объектов, она будет получать разный this

Значение this не зависит от того, как функция была создана, оно определяется исключительно в момент вызова.

Вызов в режиме обычной функции

Если функция использует this - это подразумевает работу с объектом. Но и прямой вызов func() технически возможен.

Как правило, такая ситуация возникает при ошибке в разработке.

При этом this получает значение window, глобального объекта.

```
function func() {
  alert(this); // выведет [object Window] или [object global]
}

func();
```

Аргументы функции

В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.

```
function go(a,b) {
  alert("a="+a+", b="+b);
}

go(1)     // a=1, b=undefined
go(1,2)   // a=1, b=2
go(1,2,3) // a=1, b=2, третий аргумент не вызовет ошибку
```

В некоторых языках, программист может создать две функции с одинаковым именем, но разным набором аргументов, а при вызове интерпретатор сам выберет нужную. Это называется «полиморфизмом функций» или «перегрузкой функций». В JavaScript ничего подобного нет. Может быть только одна функция с именем go, которая вызывается с любыми аргументами. А уже внутри она может посмотреть, с чем вызвана и по-разному отработать. В противном случае второе определение просто переопределит первое.

Псевдо-массив arguments

Как получить значения аргументов, которых нет в списке параметров?

Доступ к ним осуществляется через «псевдо-массив» arguments.

Он содержит список аргументов по номерам: arguments[0], arguments[1]…, а также свойство length.

Например, выведем список всех аргументов:

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

sayHi("первый", "второй");  // 'Обьект первый', 'Обьект второй'
```

В старом стандарте JavaScript псевдо-массив arguments и переменные-параметры ссылаются на одни и те же значения. В современной редакции стандарта это поведение изменено. Аргументы отделены от локальных переменных:

```
function f(x) {
  "use strict"; // для браузеров с поддержкой строгого режима
  arguments[0] = 5;
  alert(x); // не 5, а 1! Переменная "отвязана" от arguments
}
f(1);
```

arguments — не Array

Частая ошибка новичков — попытка применить методы Array к arguments. Это невозможно:

```
  function sayHi() {
    var a = arguments.shift(); // ошибка! нет такого метода!
 }

sayHi(1);
```

Дело в том, что arguments — это не массив Array.

В действительности, это обычный объект, просто ключи числовые и есть length. На этом сходство заканчивается. Никаких особых методов у него нет, и методы массивов он тоже не поддерживает.

Именованные аргументы

В JavaScript именованные параметры реализуются при помощи объекта. Вместо списка аргумента передается объект с параметрами, вот так:

```
function showWarning(options) {
  var width = options.width || 200;  // по умолчанию
  var height = options.height || 100;

  var title = options.title || "Предупреждение";

  // ...
}
```

Вызвать такую функцию очень легко. Достаточно передать объект аргументов, указав в нем только нужные:

```
showWarning({
  contents: "Вы вызвали функцию",
  showYesNo: true
});
```

Прием позволяет повторно использовать объект аргументов:

Метод call

Функцию можно вызвать, явно указав значение this.

Для этого у неё есть два метода: call и apply.

Синтаксис метода call:

```
func.call(context, arg1, arg2,...)
```

При этом вызывается функция func, первый аргумент call становится её this, а остальные передаются «как есть».

Вызов func.call(context, a, b...) — то же, что обычный вызов func(a, b...), но с явно указанным контекстом context.

Например, функция getName в примере ниже вызывается через call в контексте объекта user:

```
var user = {
  firstName: "Василий",
  surname: "Петров"
};
function getName(a, b) {
  alert( this[a] + ' ' + this[b] );
}
getName.call(user, 'firstName', 'surname')  // "Василий Петров"
```

Метод apply

Метод call жёстко фиксирует количество аргументов, через запятую:

```
f.call(context, 1, 2, 3);
```

А что, если мы захотим вызвать функцию с четырьмя аргументами? А что, если количество аргументов заранее неизвестно, и определяется во время выполнения?

Для решения этой задачи существует метод apply.

Вызов функции при помощи func.apply работает аналогично func.call, но принимает массив аргументов вместо списка:

```
func.call(context, arg1, arg2...)
// то же что и:
func.apply(context, [arg1, arg2 ... ]);
```

Эти две строчки сработают одинаково:

```
getName.call(user, 'firstName', 'surname');
getName.apply(user, ['firstName', 'surname']);
```

Метод apply гораздо мощнее, чем call, так как можно сформировать массив аргументов динамически:

```
var args = [];
args.push('firstName');
args.push('surname');
func.apply(user, args); // вызовет func('firstName', 'surname') с this=user
```

«Одалживание метода»

В JavaScript методы объекта, даже встроенные — это функции. Поэтому можно скопировать функцию, даже встроенную, из одного объекта в другой.

Это называется «одолживание метода» (на англ. method borrowing).

Используем эту технику для упрощения манипулиций с arguments. Как мы знаем, arguments - это не массив, а обычный объект.. Но как бы хотелось вызывать на нём методы массива.

```
function sayHi() {
  arguments.join = [].join; // одолжили метод (1)
  var argStr = arguments.join(':');  // (2)
  alert(argStr);  // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

В строке (1) создали массив. У него есть метод [].join(..), но мы не вызываем его, а копируем, как и любое другое свойство в объект arguments. В строке (2) запустили его, как будто он всегда там был.

Безопасное одалживание

Представим на минуту, что вместо arguments у нас — произвольный объект, и мы хотим вызвать в его контексте метод [].join. Копировать этот метод, как мы делали выше, опасно: вдруг у объекта есть свой собственный join, а мы его перезапишем и что-то перестанет работать?

Для безопасного вызова используем apply/call:

```
function sayHi() {
  var join = [].join; // ссылка на функцию теперь в переменной

  // вызовем join с this=arguments,
  // этот вызов эквивалентен arguments.join(':') из примера выше
  var argStr = join.call(arguments, ':');

  alert(argStr);  // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

Мы вызвали метод без копирования. При этом собственный метод join (если такой существует или будет существовать) останется и может быть вызван в другом месте.

«Переадресация» вызова через apply

При помощи apply мы можем сделать универсальную «переадресацию» вызова из одной фунции в другую.

Например, функция f вызывает g в том же контексте, с теми же аргументами:

```
function f(a, b) {
  g.apply(this, arguments);
}
```

Плюс этого подхода — в том, что он полностью универсален:

1) Его не понадобится менять, если в f добавятся новые аргументы.
2) Если f является методом объекта, то текущий контекст также будет передан. Если не является — то this здесь вроде как не при чём, но и вреда от него не будет.

Шаблоны проектирования

Шаблоны проектирования это хорошо продуманные решения задач программирования.

Существует три вида шаблонов проектирования:

1) **Структурные шаблоны** определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу. (Декоратор, Адаптер, Фасад ...)
2) **Порождающие** обеспечивают создание новых объектов в нужной ситуации. (Одиночка, Абстрактная фабрика, Строитель, Фабричный метод ...)
3) **Поведенческие шаблоны** отвечают за взаимодействие между сущностями, позволяя сделать этот процесс более гибким и простым. (Наблюдатель, Шаблонный метод, Посетитель ...)

Литература:

4) **Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес: Приемы ООП. Паттерны проектирования**

Задача 10

Необходимо написать реализацию строкового буфера на JavaScript, со следующим функционалом:

1. **Создание буфера**: var txt = newBuffer();.
2. **Занесение строки в буфер** (Вызов newBuffer должен возвращать такую функцию, которая при вызове txt(value) добавляет значение в некоторое внутреннее хранилище)
3. **Получение текущего содержимого буфера** (при вызове без аргументов txt() — возвращает текущее содержимое буфера).
4. **Очистка буфера**. Метод clear(), который будет очищать текущее содержимое буфера.

```
function newBuffer() { /* ваш код */ }

var txt = newBuffer();
txt('Значение1'); // добавяем значение к буферу
txt(' Значение2'); // добавяем значение к буферу
txt(' Значение3'); // добавяем значение к буферу

alert( txt() ); //получаем текущее значение (Значение1 Значение2 Значение3)
txt.clear();  // очищаем буфер
alert( txt() ); // получаем текущее значение (пустая строка)
```

Объекты. Строковое преобразование

Строковое преобразование проще всего увидеть, если вывести объект при помощи alert:

```
var obj = {
  firstName: 'Иванов'
};
alert(obj); // [object Object]
```

Как видно, содержимое объекта не вывелось. Это потому, что стандартным строковым представлением пользовательского объекта является строка "[object Object]".

Такой вывод объекта не содержит интересной информации. Поэтому имеет смысл его поменять на что-то более полезное.

Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования.

```
var user = {
  firstName: 'Иванов',
  toString: function() {
    return 'Пользователь ' + this.firstName;
  }
};
alert( user );  // Пользователь Иванов
```

Все объекты, включая встроенные, имеют свои реализации метода toString

```
alert( new Date ); // toString для дат выводит дату в виде строки
alert( function() { } ); // toString для функции выводит её код
```

Численное преобразование

Для численного преобразования объекта используется метод valueOf, а если его нет — то toString:

```
var room = {
  number: 777,
  valueOf: function() { return this.number; },
  toString: function() { return this.number; }
};

alert( +room );  // 777, вызвался valueOf
delete room.valueOf;
alert( +room );  // 777, вызвался toString
```

Метод valueOf обязан возвращать примитивное значение(не обязательно числовое), иначе его результат будет проигнорирован.

У большинства встроенных объектов такого valueOf нет, поэтому численное и строковое преобразования для них работают одинаково.

Исключением является объект Date, который поддерживает оба типа преобразований:

```
alert( new Date() ); // toString: Дата в виде читаемой строки
alert( +new Date() ); // valueOf: кол-во миллисекунд, прошедших с 01.01.1970
```

Преобразование в примитив

Если объект участвует в операции, которая предполагает примитивные значения, то он автоматически преобразуется к примитиву.

Например, арифметическая операция или сравнение(> < >= <=) сначала преобразует объект в примитив, а потом уже идёт операция с примитивами.

При приведении объекта к примитиву используется численное преобразование.

```
var obj = {
  valueOf: function() { return 1; }
};

// операции сравнения, кроме ===, приводят к примитиву
alert(obj == true); // true, объект приведён к примитиву 1

// бинарный + приводит к примитиву, а затем складывает
alert(obj + "test"); // 1test
```

Оператор typeof

Оператор typeof возвращает тип аргумента. У него есть два синтаксиса:

1) Синтаксис оператора: typeof x.
2) Синтаксис функции: typeof(x).

Работают они одинаково, но первый синтаксис короче.

Результатом typeof является строка, содержащая тип:

```
typeof undefined // "undefined"
typeof 0    // "number"
typeof true // "boolean"
typeof "foo" // "string"
typeof {} // "object"
typeof null  // "object"
typeof function(){} // "function"
```

* Результат typeof null == "object" — это официально признанная ошибка в языке, которая сохраняется для совместимости. На самом деле null — это не объект, а примитив.

* Для функции f значением typeof f является "function". Конечно же, функция является объектом. С другой стороны, такое выделение функций на практике скорее плюс, т.к. позволяет легко определить функцию.

Перехват ошибок, "try..catch"

Типы ошибок

1) Синтаксические ошибки — когда нарушена структура кода, например:

```
for (a = 5) { // неправильный аргумент for
]; // неправильная закрывающая скобка
```

Обычно это ошибки программиста. Их также называют «ошибки времени компиляции», поскольку они происходят на этапе разбора кода.

Как-либо «восстановить» выполнение скрипта здесь нельзя. Браузер не может такой код разобрать и выполнить.

2) Семантические ошибки, также называемые «ошибки времени выполнения» — возникают, когда браузер смог прочитать скрипт и уже выполняет код, но вдруг натыкается на проблему.

Например, «не определена переменная»:

```
alert(nonexistant); // ошибка, переменная не определена!
```

Эти ошибки можно перехватить и обработать самостоятельно, не дать скрипту «упасть».

Конструкция try..catch

Конструкция try..catch состоит из двух основных блоков: try, и затем catch. Например:

```
try {
  var number = 9;
  alert( number.toString(2) );
  // ...
  alert('Выполнено без ошибок!');
} catch(e) {
  alert('Ошибка!');
}
```

Работает это так:

1. Выполняется код внутри блока try.
2. Если в нём возникнет ошибка, то выполнение try прерывается, и управление прыгает в начало блока catch.
3. Если ошибки нет — блок catch игнорируется.

Объект ошибки

У блока catch есть аргумент, в примере выше он обозначен err. Это — объект ошибки или, как ещё говорят, объект исключения (exception object).

Он содержит информацию о том, что произошло, и может быть разным, в зависимости от ошибки.

Как правило, err — объект встроенного типа Error и производных от него.

Есть несколько свойств, которые присутствуют в объекте ошибки:

1) **Name** - Тип ошибки. Например, при обращении к несуществующей переменной равен "ReferenceError".
2) **Message** - Текстовое сообщение о деталях ошибки.
3) **Stack** - Везде, кроме IE<9, есть также свойство stack, указывающее, где в точности произошла ошибка.

В зависимости от браузера, у него могут быть и дополнительные свойства

```
try {
  var number = 9;
  var base = 100;
  var convertedNumber = number.toString(base);
} catch(err) {
  alert("name: " + err.name + "\nmessage: " + err.message + "\nstack: " + err.stack);
}
```

Секция finally

Конструкция try..catch может содержать ещё один блок: finally. Секция finally не обязательна, но если она есть, то она выполняется всегда:

1) после блока try, если ошибок не было,
2) после catch, если они были.

Её используют, чтобы завершить начатые операции и очистить ресурсы, которые должны быть очищены в любом случае — как при ошибке, так и при нормальном потоке выполнения. Блок finally позволяет избежать дублирования кода в try/catch.

Например, функция в процессе работы создает новый объект, который нужно в конце уничтожить. Реализация будет выглядеть так:

```
function doSomething() {
  var tmpObject = getObject(); // создать что-то, что нужно будет очистить
  try {
    .. поработать с tmpObject ..
  } catch(e) {
    .. обработать ошибку ..
  } finally {
    .. удалить объект tmpObject ..
  }
}
```

Генерация ошибки: throw

Конструкцию try..catch можно использовать не только со встроенными типами ошибок JavaScript. Можно создавать свои типы ошибок.

Синтаксис: throw <объект ошибки>.

Инициирует ошибку, которую можно поймать в catch. Объектом ошибки может быть что угодно, например число:

```
var value = Input.value;

try {
  checkRequired(value);
  checkInteger(value);
  checkValidNum(value);

  /* ввод успешен */
} catch(e) {
  alert(e.message);
  /* обработать ошибку */
}
```

```
function BadValueError(message) {
  this.name = "BadValue";
  this.message = message;
}

function checkValidNum(age) {
  if (age < 1000) {
    throw new BadValueError("Некорректное значение");
  }
}

function checkRequired(value) {
  if (value == '') {
    throw new BadValueError("Отсутствует значение");
  }
}
```

Вложенные вызовы и try..catch

Брошенное исключение выпадает из всех циклов, функций и т.п.

Это — особое, уникальное свойство исключений. Оно означает, что в случае вложенных вызовов — не важно, где было исключение, оно будет поймано ближайшим внешним try..catch.

```
function checkAll(value) {
  checkRequired(value);
  checkInteger(value);
  checkValidNum(value);
}

try {
  checkAll(value);
  alert('Значение корректно');
} catch(e) {
  if (e.name == "BadValue") {
    //обрабатываем исключение
    alert('Ошибка '+e.message);
  } else {
    //пробрасываем исключние
    throw e;
  }
}
```

Данная конкретная функция checkAll умеет обрабатывать только ошибку проверки BadValue, что и делает. В том случае, если исключение какое-то другое, то она «пробрасывает» (throw e) его дальше, а там оно либо попадёт во внешний блок try..catch, который знает, что с ним делать, либо выпадает из скрипта как ошибка — и тогда разработчик увидит его в консоли.

Задача 11

1) Переписать код функции текстового буфера таким образом, чтобы в буфер можно было заносить только цифры.
2) Сделать обработку искючения для случая попытки занесения неверных данных.

DOM

Дерево DOM

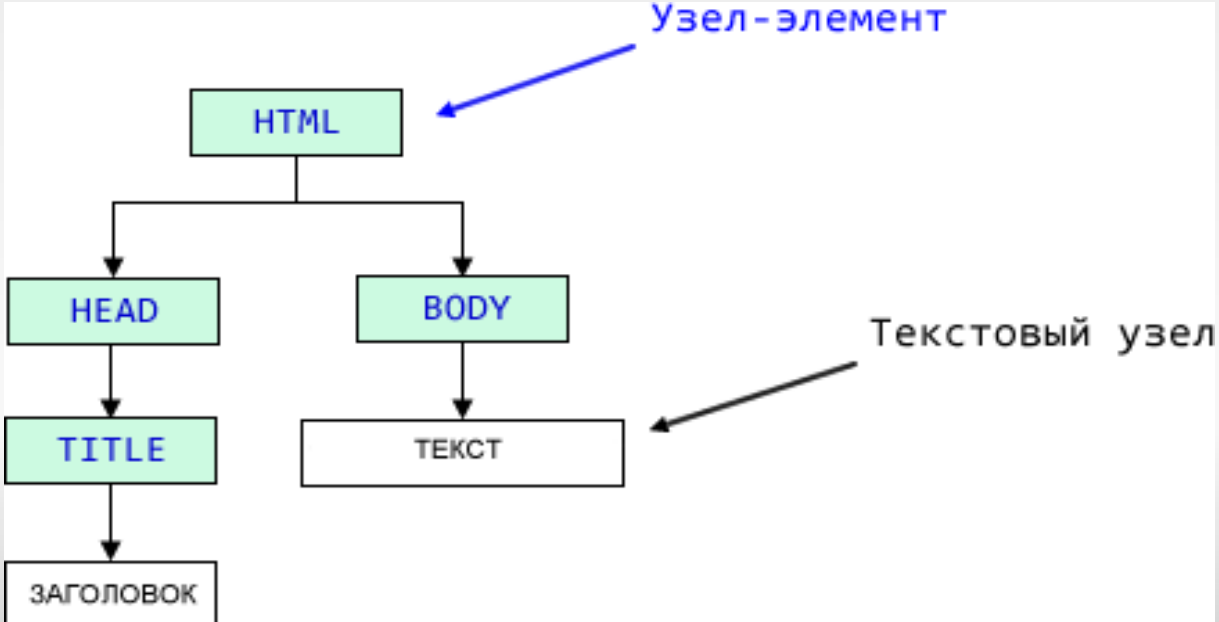
Основным инструментом работы и динамических изменений на странице является DOM (Document Object Model) — объектная модель, используемая для XML/HTML-документов. Согласно DOM-модели, документ является иерархией, деревом. Каждый HTML-тег образует узел дерева с типом «элемент». Вложенные в него теги становятся дочерними узлами. Для представления текста создаются узлы с типом «текст».

В этом дереве выделено два типа узлов.

1) Теги образуют узлы-элементы (element node) DOM-дерева. Естественным образом одни узлы вложены в другие.

2) Текст внутри элементов образует текстовые узлы. Текстовый узел содержит исключительно строку текста и не может иметь потомков.

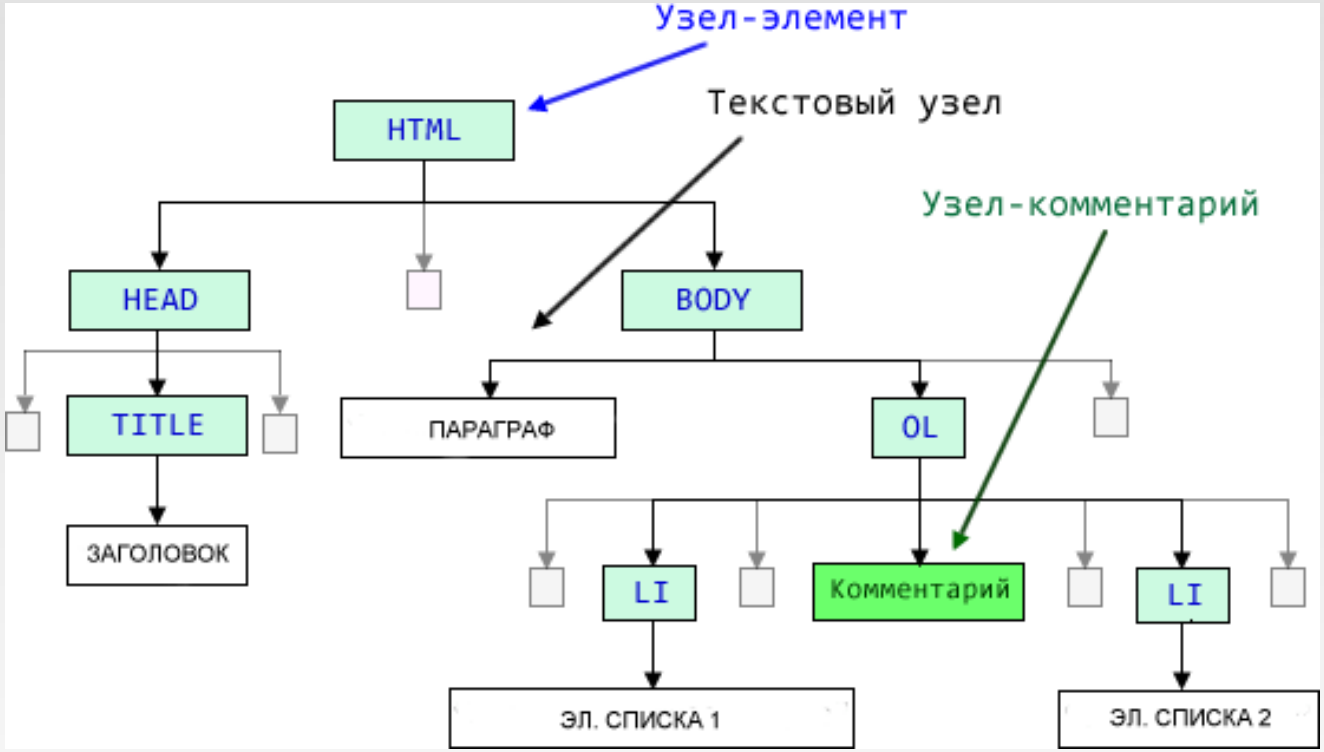
```
<html>
  <head>
    <title>ЗАГОЛОВОК</title>
  </head>
  <body>
     ТЕКСТ
   </body>
</html>
```



Пробельные узлы и комментарии

Всё, что есть в HTML, находится и в DOM. Поскольку DOM-модель в точности соответствует документу, пробельные символы так же важны, как и любой другой текст. Комментарии также содержатся в DOM.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>ЗАГОЛОВОК</title>
  </head>
  <body>
   ПАРАГРАФ
    <ol>
      <li>ЭЛ. СПИСКА 1</li>
      <!-- комментарий -->
      <li>ЭЛ. СПИСКА 2</li>
    </ol>
  </body>
</html>
```



Возможности, которые дает DOM

DOM нужен для того, чтобы манипулировать страницей — читать информацию из HTML, создавать и изменять элементы.

Например, можно поменять цвет BODY и вернуть обратно:

```
document.body.style.backgroundColor = 'red';
alert('Поменяли цвет BODY');

document.body.style.backgroundColor = '';
alert('Сбросили цвет BODY');
```

Фактически, DOM предоставляет возможность делать со страницей всё, что угодно.

Для того, чтобы изменить узел DOM, например, раскрыть в нем меню, нужно сначала его получить.

Доступ к DOM начинается с document. Оттуда можно добраться до любых других узлов.

Корень: documentElement и body

Получить «корень» дерева DOM можно двумя путями.

1) Первая точка входа — document.documentElement. Это свойство ссылается на DOM-объект для тега HTML.

2) Вторая точка входа — document.body, который соответствует тегу BODY.

Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.

```
<!DOCTYPE HTML>
<html>
  <head>
    <script>
      alert("Из HEAD: " + document.body); // null
    </script>
  </head>
  <body>
    <script>
      alert("Из BODY: " + document.body);
    </script>
  </body>
</html>
```

Дочерние элементы

Из узла-родителя можно получить все дочерние элементы. Для этого есть несколько способов.

1) childNodes - Псевдо-массив childNodes хранит все дочерние элементы, включая текстовые.

```
<!DOCTYPE HTML>
<html>
  <head><meta charset="utf-8"></head>
  <body>
    <div>Список</div>
    <ul>
      <li>Эл списка 1</li>
      <li>Эл списка 2</li>
    </ul>
    <!-- комментарий -->
    <script>
      var childNodes = document.body.childNodes;
      for(var i=0; i<childNodes.length; i++) {
        alert(childNodes[i]);
      }
    </script>
  </body>
</html>
```

Во всех браузерах, кроме старых IE, document.body.childNodes[0] это текстовый узел из пробелов, а DIV — второй потомок: document.body.childNodes[1].

В IE<9, не создаются пустые текстовые узлы, поэтому там дети начнутся с DIV.

Результат выполнения:
[object Text][object HTMLDivElement]
[object Text][object HTMLUListElement]
[object Text][object Comment]
[object Text][object HTMLScriptElement]

children

1) Свойство children перечисляет только дочерние узлы, соответствующие тегам.

```
<!DOCTYPE HTML>
<html>
  <head><meta charset="utf-8"></head>
  <body>
   <div>Список</div>
    <ul>
      <li>Эл списка 1</li>
      <li>Эл списка 2</li>
    </ul>
    <!-- комментарий -->

    <script>
      var children = document.body.children;
      for(var i=0; i<children.length; i++) {
        alert(children[i]); // DIV, UL, SCRIPT
      }
    </script>

  </body>
</html>
```

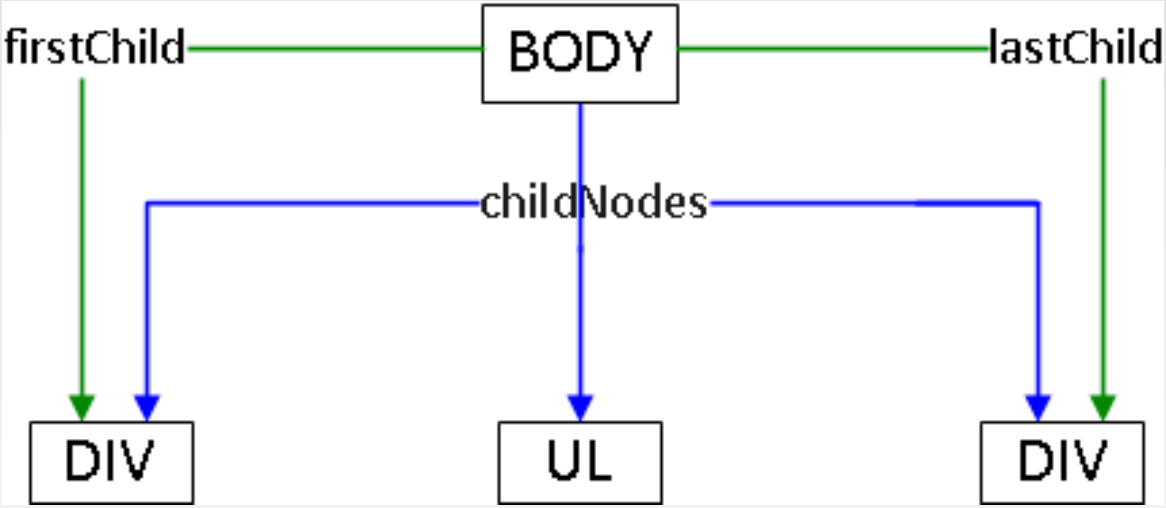
Результат выполнения:
[object HTMLDivElement]
[object HTMLUListElement]
[object HTMLScriptElement]

firstChild и lastChild

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему потомку.

Например, для документа:

```
<html><body><div>...</div><ul>...</ul><div>...</div></body></html>
```



firstChild и lastChild — это более быстрый и короткий способ обратиться к первому и последнему элементам childNodes. Верны равенства:

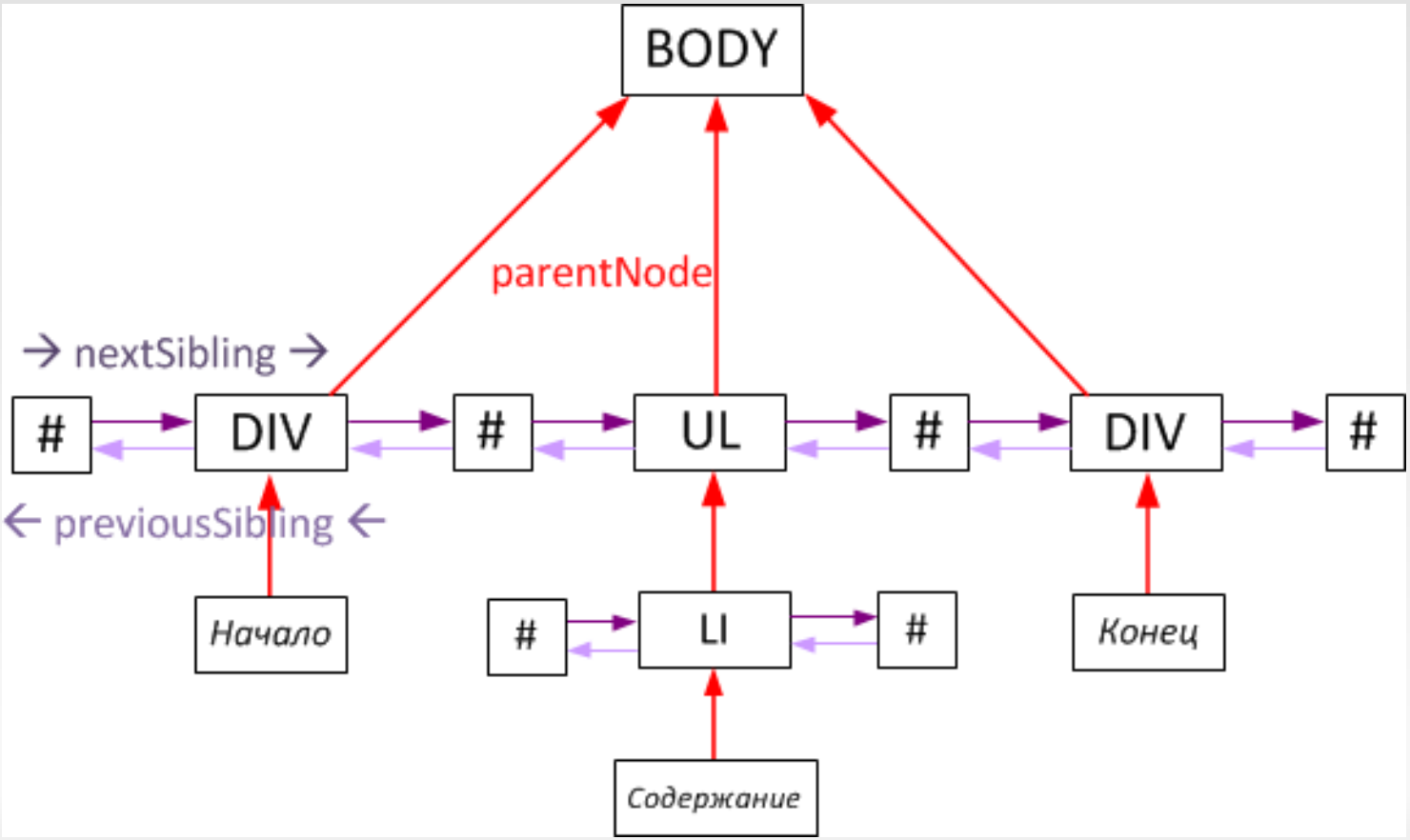
```
document.body.firstChild === document.body.childNodes[0]

document.body.lastChild === document.body.childNodes[body.childNodes.length-1]
```

parentNode, previousSibling и nextSibling

1) Свойство parentNode ссылается на родительский узел.
2) Свойства previousSibling и nextSibling дают доступ к левому и правому соседу.

Ниже изображены ссылки между BODY и его потомками документа:

```
<!DOCTYPE HTML>
<html>
<head>
 <meta charset="utf-8">
</head>
<body>
  <div>Начало</div>

  <ul>
    <li>Содержание</li>
  </ul>

  <div>Конец</div>
</body>
</html>
```



Все навигационные ссылки — только для чтения. При изменениях DOM, добавлении или удалении элементов они обновляются автоматически.

Таблицы

У таблиц есть дополнительные свойства для более удобной навигации по ним:

TABLE

1) **table.rows** — список строк TR таблицы.
2) table.caption/tHead/tFoot — ссылки на элементы таблицы CAPTION, THEAD, TFOOT.
3) table.tBodies — список элементов таблицы TBODY.

THEAD/TFOOT/TBODY

4) tbody.rows — список строк TR секции.

TR

5) **tr.cells** — список ячеек TD/TH
6) **tr.sectionRowIndex** — номер строки в текущей секции THEAD/TBODY
7) tr.rowIndex — номер строки в таблице

TD/TH

8) **td.cellIndex** — номер ячейки в строке

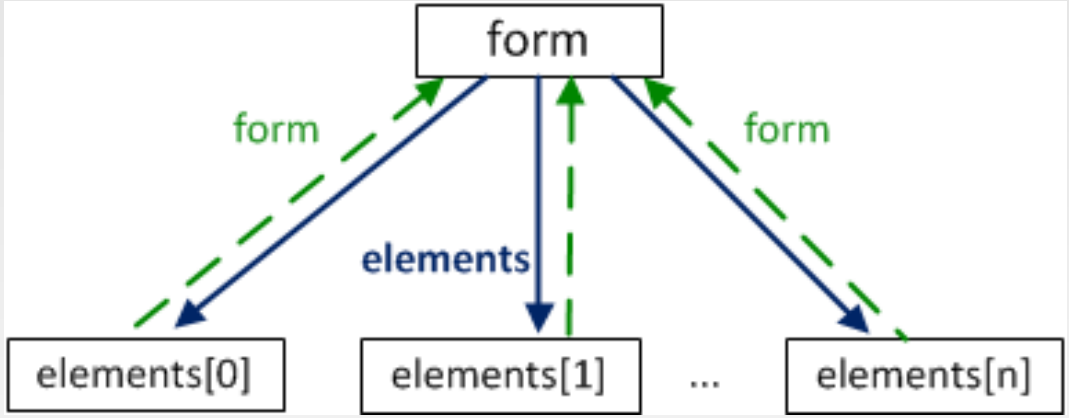
```
<table>
  <tr> <td>один</td> <td>два</td>    </tr>
  <tr> <td>три</td>  <td>четыре</td> </tr>
</table>

<script>
var table = document.body.children[0];

alert( table.rows[0].cells[0].innerHTML ) // "один"
</script>
```

Формы

Элементы FORM можно получить по имени или номеру, используя свойство document.forms[name/index].

document.forms.my -- форма с именем 'my'
document.forms[0] -- первая форма в документе

Любой элемент формы form можно получить аналогичным образом, используя свойство form.elements.




```
<body>
<form>
<input type="radio" name="age" value="10">
<input type="radio" name="age" value="20">
</form>

<script>
var form = document.forms[0];

var elems = form.elements.age;

alert(elems[0].value); // 10, первый input
</script>
</body>
```

Может быть несколько элементов с одинаковым именем. В таком случае form.elements[name] вернет коллекцию элементов.
Эти ссылки не зависят от окружающих тегов. Элемент может быть зарыт где-то глубоко в форме, но он всё равно доступен через form.elements.

От элементов к форме

По элементу можно получить его форму, используя свойство element.form.

```
<body>
<form>
  <input type="text" name="surname">
</form>

<script>
var form = document.forms[0];

var elem = form.elements.surname;

alert(elem.form == form); // true
</script>
</body>
```

Дополнительные ссылки для элементов.

Все современные браузеры, включая IE9+, поддерживают дополнительные ссылки:

1) childElementCount — число потомков-элементов (=children.length)
2) firstElementChild — первый потомок-элемент (=children[0])
3) lastElementChild — последний потомок-элемент (=children[children.length-1])
4) nextElementSibling — правый соседний-элемент
5) previousElementSibling — левый соседний-элемент

Любые другие узлы, кроме элементов, при этом просто игнорируются

```
<body>
  firstElementChild: <div>...</div>
  <!-- комментарий -->
  lastElementChild: <span>...</span>
  <script>
    alert(document.body.firstElementChild.nextElementSibling); // SPAN
  </script>
</body>
```

Современные браузеры также поддерживают дополнительные интерфейсы для обхода DOM с фильтром по узлам: NodeIterator, TreeFilter и TreeWalker.

Тип узла: nodeType

С тремя типами мы уже встречались. Это 1) элемент, 2) текстовый узел и 3) комментарий. На самом деле типов узлов гораздо больше. Строго говоря, их 12, и они описаны в спецификации

```
interface Node {
  // NodeType
  const unsigned short      ELEMENT_NODE       = 1;
  const unsigned short      ATTRIBUTE_NODE     = 2;
  const unsigned short      TEXT_NODE          = 3;
  const unsigned short      CDATA_SECTION_NODE = 4;
  const unsigned short      ENTITY_REFERENCE_NODE = 5;
  const unsigned short      ENTITY_NODE        = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short      COMMENT_NODE       = 8;
  const unsigned short      DOCUMENT_NODE      = 9;
  const unsigned short      DOCUMENT_TYPE_NODE = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short      NOTATION_NODE      = 12;
  ...
}
```

Самые важные — это ELEMENT_NODE под номером 1 и TEXT_NODE под номером 3.

Тип узла содержится в его свойстве nodeType.

```
childNodes[i].nodeType
```

Тег: nodeName и tagName

Существует целых два свойства: nodeName и tagName, которые содержат название(тег) элемента узла.

Название HTML-тега всегда находится в верхнем регистре.

```
alert( document.body.nodeName );   // BODY
alert( document.body.tagName );   // BODY
```

1) Свойство nodeName определено для многих типов DOM-узлов.

2) Свойство tagName — есть только у элементов (в IE<9 также у комментариев, это ошибка в браузере).

Иначе говоря, при помощи tagName мы можем работать только с элементами, а nodeName может что-то сказать и о других типах узлов.

```
<body><!-- комментарий -->
<script>
  // для комментария
  alert(document.body.firstChild.nodeName); // #comment
  alert(document.body.firstChild.tagName); // undefined (в IE<9 воскл. знак "!")
  // для документа
  alert(document.nodeName); // #document, т.к. корень DOM -- не элемент
  alert(document.tagName); // undefined
</script>
</body>
```

innerHTML: содержимое элемента

Свойство innerHTML позволяет получить HTML-содержимое узла в виде строки. В innerHTML можно и читать и писать.

Пример выведет на экран все содержимое document.body, а затем заменит его на другое:

```
<body>
  <p>Параграф</p>
  <div>Div</div>
  <script>
    alert( document.body.innerHTML ); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
  </script>
</body>
```

При чтении innerHTML HTML-код всегда будет валиден. При записи — можно записать что угодно, браузер поправит некорректный HTML-код:

Тонкости innerHTML

1) Для таблиц в IE9- — innerHTML только для чтения

   В Internet Explorer версии 9 и ранее, innerHTML доступно только для чтения для элементов COL, COLGROUP, FRAMESET, HEAD, HTML, STYLE, TABLE, TBODY, TFOOT, THEAD, TITLE, TR.

2) Добавление innerHTML+= осуществляет перезапись

```
chatDiv.innerHTML += "<div>Привет<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "Как дела?";
```

   - Удаляется старое содержание
   - На его место становится новое значение innerHTML.

   Так как новое значение записывается с нуля, то все изображения и другие ресурсы будут перезагружены. В примере выше вторая строчка перезагрузит smile.gif, который был до неё. Если в chatDiv много текста, то эта перезагрузка будет очень заметна.

3) Если в innerHTML есть тег script — он не будет выполнен.

```
<div id="my"></div>
<script>
  var elem = document.getElementById('my');
  elem.innerHTML = 'ТЕСТ<script>alert(1);</scr'+'ipt>';
</script>
```

outerHTML: HTML узла целиком

Свойство outerHTML содержит HTML узла целиком.

Оно поддерживается всеми браузерами, а в Firefox начиная с версии 11.

```
<div id="hi">Привет <b>Мир</b></div>
<script>
  var div = document.getElementById('hi');
  alert(div.outerHTML); // <div>Привет <b>Мир</b></div>
</script>
```

Можно и записать outerHTML, однако при этом новый текст будет вставлен вместо узла, а сама переменная не изменится.

```
<div id="hi">Привет <b>Мир</b></div>
<script>
  var div = document.getElementById('hi');
  div.outerHTML = '<p>Новый HTML!</p>';
  alert(div.tagName); // DIV
  alert(div.innerHTML); // Привет <b>Мир</b>
</script>
```

При замене div.outerHTML в переменной div остаётся старый узел, который теперь оторван от документа, но полностью «жизнеспособен».

nodeValue/data: содержимое текстового узла

Свойство innerHTML есть только у узлов-элементов. Содержимое других узлов, например, текстовых или комментариев, доступно через два свойства: nodeValue и data. Его тоже можно читать и обновлять. Следующий пример демонстрирует это:

```
<body>
  Привет
  <!-- Комментарий -->
  <script>
    for (var i=0; i<document.body.childNodes.length; i++) {
      alert(document.body.childNodes[i].nodeValue);
      alert(document.body.childNodes[i].data);
    }
    document.body.firstChild.data = "Здравствуйте!";
  </script>
</body>
```

В этом примере выводятся последовательно:

1. Содержимое первого узла (текстового): Привет (2 раза).

2. Содержимое второго узла (комментария): Комментарий (2 раза).

3. Содержимого третьего узла (текста между комментарием и скриптом): (там пробелы, 2)

4. Свойство nodeValue=null для узла SCRIPT, так как это узел-элемент. А вот data=undefined. Это единственное различие в поведении этих свойств.

5. Наконец, последний вызов заменит содержимое document.body.firstChild, и это тут же отразится в документе.

Кстати, после SCRIPT есть еще один текстовый узел, но на момент работы скрипта браузер дошел в разборе документа только до SCRIPT, поэтому он не будет выведен.

Другие свойства

У DOM-узлов есть свойства, зависящие от типа, например:

1) value — значение для INPUT, SELECT или TEXTAREA
2) id — идентификатор
3) href — адрес ссылки

```
<input type="text" id="my-input" value="значение">
<script>
  var input = document.body.children[0];
  alert(input.type); // "text"
  alert(input.id); // "my-input"
  alert(input.value); // значение
</script>
```

Полный список свойств можно получить из спецификации. В частности, для input[type="text"] она находится http://www.w3.org/TR/html-markup/input.text.html. Как правило, основные атрибуты элемента дают свойство с тем же названием.

Пользовательские свойства

Узел DOM - это объект, поэтому, как и любой объект в JavaScript, он может содержать пользовательские свойства и методы. Можно добавить и новую функцию.

```
document.body.myData = {
  name: 'Петр',
  familyName: 'Петрович'
};

alert(document.body.myData.name); // Петр
```

```
document.body.sayHi = function() {
  alert(this.nodeName);
}

document.body.sayHi();
// BODY, выполнилась с правильным this
```

Такие свойства и методы видны только в JavaScript и никак не влияют на отображение соответствующего тега. Эти свойства можно даже перечислить с помощью for..in (свойства будут все, и встроенные тоже)

```
document.body.custom = 5;

var list = [];
for (var key in document.body) {
  list.push(key);
}

alert( list.join('\n') );
```

Пользовательские DOM-свойства:

- Могут иметь любое значение.
- Названия свойств чувствительны к регистру.
- Работают за счет того, что DOM-узлы являются объектами JavaScript

Задача 12

1) Для произвольной страницы посчитать количество содержащихся в ней элементов различных типов
2) Вывести полученый результат в виде таблицы

ПАРАГРАФ

1. ЭЛ. СПИСКА 1
2. ЭЛ. СПИСКА 2

#text	10
OL	1
LI	2
#comment	1
DIV	1
SCRIPT	1

Атрибуты

Узлы DOM, с другой стороны, являются HTML-элементами, у которых есть атрибуты.

Доступ к атрибутам осуществляется при помощи стандартных методов:

1) elem.hasAttribute(name) - проверяет наличие атрибута
2) elem.getAttribute(name) - получает значение атрибута
3) elem.setAttribute(name, value) - устанавливает атрибут
4) elem.removeAttribute(name) - удаляет атрибут

В отличие от свойств, атрибуты:

5) Могут быть только строками.
6) Их имя нечувствительно к регистру(т.к. это HTML)
7) Видны в innerHTML (за исключением старых IE)
8) Все атрибуты элемента можно получить с помощью свойства attributes, которое содержит псевдо-массив объектов типа Attr.

Особенности аттрибутов

```
<body>
  <div about="Elephant" class="smiling">
  </div>

  <script>
    var div = document.body.children[0];
    alert( div.getAttribute('ABOUT') ); // (1)

    div.setAttribute('Test', 123);   // (2)
    alert( document.body.innerHTML );   // (3)

    var attrs = div.attributes;  // (4)
    for (var i=0; i<attrs.length; i++) {
      alert(attrs[i].name + " = " + attrs[i].value);
    }
  </script>
</body>
```

При запуске кода

1) getAttribute('ABOUT') использует имя атрибута в верхнем регистре, но это не имеет значения, т.к имена нечувствительны к регистру.

2) Вы можете записать в атрибут строку или другое значение, которые будет превращено в строку. Объект, например, будет автоматически сконвертирован.(у IE<9 с этим проблемы, поэтому придерживайтесь примитивов).

3) В innerHTML появился новый атрибут "test".

4) Коллекция attributes содержит все атрибуты в виде объектов со свойствами name и value.

Синхронизация свойств и атрибутов

Каждый тип узлов DOM имеет свои стандартные свойства. Например, свойства тега 'A' описаны в спецификации DOM: HTMLAnchorElement. Например, у него есть свойство "href". Кроме того, он имеет "id" и другие свойства, общие для всех элементов, которые описаны в спецификации в HTMLElement.

Стандартные свойства DOM синхронизируются с атрибутами.

Например, браузер синхронизирует атрибут "id" со свойством id. Изменим одно - поменяется и другое.

```
<script>
  document.body.setAttribute('id','la-la-la');
  alert(document.body.id); // la-la-la
</script>
```

Синхронизация не гарантирует одинакового значения. Для примера, посмотрим что произойдет с атрибутом "href" при изменении свойства:

```
<a href="#"></a>
<script>
  var a  = document.body.children[0];
  a.href = '/';
  alert( 'атрибут:' + a.getAttribute('href') ); // '/'
  alert( 'свойство:' + a.href );  // полный URL
</script>
```

Это происходит потому, что атрибут может быть любым, а свойство href, в соответствии со спецификацией W3C, должно быть полной ссылкой.

Синхронизация свойств и атрибутов

А еще есть встроенные свойства, которые синхронизируются в одну сторону.

Например, input.value синхронизируется только из атрибута в свойство:

```
<body>
  <input type="text" value="markup">
  <script>
    var input = document.body.children[0];

    input.setAttribute('value', 'new');

    alert( input.value );
    // 'new', input.value изменено
  </script>
</body>
```

```
<body>
  <input type="text" value="markup">
  <script>
    var input = document.body.children[0];

    input.value = 'new';

    alert(input.getAttribute('value'));
   // 'markup', не изменилось!
  </script>
</body>
```

В результате атрибут value хранит оригинальное (исходное) значение даже после того, как пользователь заполнил поле и свойство изменилось. Текущее же значение хранится в свойстве value.

Изначальное значение может использоваться для проверки, было ли изменено значение input и, при необходимости, для того, чтобы отменить изменения.

Синхронизация свойств и атрибутов

Исключение: атрибут "class" соответствует свойству className.

Так как слово "class" является зарезервированным словом в Javascript, то атрибуту "class" соответствует свойство className.

```
<body>
  <script>
    document.body.setAttribute('class', 'big red bloom');
    alert( document.body.className );  // big red bloom
  </script>
</body>
```

Приведенный пример работает во всех современных браузерах, т.к. в них это соответствие выполняется точно.

Но он не работает в IE<9, точнее — не работает установка класса.

Поэтому рекомендуется, по возможности, не использовать атрибут "class", а использовать только свойство className.

getElementById

У каждого DOM-элемента может быть атрибут id. Его значение должно быть уникальным для документа.

Вызов document.getElementById(id) возвращет элемент с данным id.

Если не найден, то null.

```
<body>
  <div id="info">Информация</div>
  <script>
    var div = document.getElementById('info');
    alert( div.innerHTML ); // Информация
  </script>
</body>
```

getElementsByTagName

Метод elem.getElementsByTagName(tag) ищет все элементы с заданным тегом tag внутри элемента elem и возвращает их в виде списка.

Регистр тега не имеет значения.

Можно искать и в элементе и в документе:

```
// получить все div-элементы

var elements = document.getElementsByTagName('div');
```

Можно получить все элементы, передав звездочку '*' вместо тега:

```
// получить все элементы документа

var allElems = document.getElementsByTagName('*');
```

Если хочется получить только один элемент — можно указать индекс сразу же:

```
var element = document.getElementsByTagName('input')[0]
```

getElementsByName

Вызов document.getElementsByName(name) позволяет получить все элементы с данным атрибутом name.

Например, все элементы с именем age:

var elems = document.getElementsByName('age');

До появления стандарта HTML5 этот метод возвращал только те элементы, в которых предусмотрена поддержка атрибута name, в частности: iframe, a, input и другими.

```
<input name="test">
<div name="test"></div>
<script>
  var elems = document.getElementsByName('test');
  alert(elems.length); // 2 в современных браузерах, 1 в IE<10
</script>
```

getElementsByClassName

Вызов elem.getElementsByClassName(className) возвращает коллекцию элементов с классом className. Находит элемент и в том случае, если у него несколько классов, а искомый - один из них.

Поддерживается всеми современными браузерами, кроме IE<9.

```
<div class="article">Статья</div>
<div class="long article">Длинная статья</div>

<script>
  var articles = document.getElementsByClassName('article');
  alert( articles.length ); // 2, найдёт оба элемента
</script>
```

Как и getElementsByTagName, этот метод может быть вызван и в контексте DOM-элемента и в контексте документа.

querySelectorAll, querySelector

Вызов elem.querySelectorAll(cssQuery) возвращает все элементы внутри elem, удовлетворяющие CSS-селектору cssQuery.

Следующий запрос получает все элементы LI, которые являются последними потомками своих UL.

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  var elements = document.querySelectorAll('UL > LI:last-child');
  for (var i=0; i<elements.length; i++) {
    alert(elements[i].innerHTML ); // "тест", "пройден"
  }
</script>
```

querySelector - то же самое, что elem.querySelectorAll(cssQuery), но возвращает только первый элемент. Фактически, эквивалентен elem.querySelectorAll(cssQuery)[0], но быстрее, так как ищутся не все элементы, а только первый.

Относительное расположение узлов

Для проверки взаимного расположения узлов есть стандартный метод compareDocumentPosition

```
var result = nodeA.compareDocumentPosition(nodeB);
```

Возвращаемое значение — битовая маска (см. Побитовые операторы), биты в которой означают следующее:

Биты	Число	Значение
000000	0	nodeA и nodeB — один и тот же узел
000001	1	Узлы в разных документах (или один из них не в документе)
000010	2	nodeB предшествует nodeA (в порядке обхода документа)
000100	4	nodeA предшествует nodeB
001000	8	nodeB содержит nodeA
010000	16	nodeA содержит nodeB
100000	32	Зарезервировано для браузера

Понятие «предшествует» — имеется в виду не «предыдущий сосед при общем родителе», а «раньше встречается в порядке прямого обхода дерева документа.

Могут быть и сочетания битов.

Создание элементов: createElement

1) document.createElement(tag)

   Создает новый элемент с указанным тегом:

   var div = document.createElement('div');

2) document.createTextNode(text)

   Создает новый текстовый узел с данным текстом:

   var textElem = document.createTextNode('Тут был я');

Новому элементу тут же можно поставить свойства

```
var newDiv = document.createElement('div');
newDiv.className = 'myclass';
newDiv.id = 'myid';

newDiv.innerHTML = 'Привет, мир!';
```

Добавление элемента: appendChild

Чтобы DOM-узел был показан на странице, его необходимо вставить в документ.

Для этого у любого элемента есть метод appendChild - Добавляет elem в список дочерних элементов parentElem. Новый узел добавляется в конец списка.

```
parentElem.appendChild(elem)
```

```
<div>
  ...
</div>
<script>
  var parentElem = document.body.children[0];

  var newDiv = document.createElement('div');
  newDiv.innerHTML = 'Привет, мир!';

  parentElem.appendChild(newDiv);
</script>
```

appendChild - возвращают вставленный узел.

Добавление элемента: insertBefore

1) insertBefore(elem, nextSibling) - Вставляет elem в список дочерних parentElem, перед элементом nextSibling.

Сделать новый div первым дочерним можно так:

```
<div>
  ...
</div>
<script>

  var parentElem = document.body.children[0];

  var newDiv = document.createElement('div');
  newDiv.innerHTML = 'Привет, мир!';

  parentElem.insertBefore(newDiv, parentElem.firstChild);
</script>
```

Вместо nextSibling может быть null, тогда insertBefore работает как appendChild.

```
  parentElem.insertBefore(elem, null);
  // то же что:
  parentElem.appendChild(elem)
```

Все методы вставки возвращают вставленный узел, например parent.appendChild(elem) возвращает elem.

Удаление узлов: removeChild

Для удаления узла есть два метода:

1) parentElem.removeChild(elem)

   Удаляет elem из списка потомков parentElem.

2) parentElem.replaceChild(elem, currentElem)

   Среди потомков parentElem заменяет currentElem на elem.

Оба этих метода возвращают удаленный узел. Они просто вынимают его из списка, никто не мешает вставить его обратно в DOM в будущем.

Перемещение элементов

Если вы хотите переместить элемент на новое место — не нужно его удалять со старого.

Методы appendChild/insertBefore автоматически удаляют вставляемый элемент со старого места.

Например, поменяем элементы местами

```
<div>First div</div>
<div>Last div</div>
<script>
  var first = document.body.children[0];
  var last = document.body.children[1];

  // нет необходимости в предварительном removeChild(last)
  document.body.insertBefore(last, first); // поменять местами
</script>
```

cloneNode()

Метод cloneNode() создает копию указанного узла.

```
newElem = elem.cloneNode(true)
```

Метод cloneNode() имеет параметр (true – истина или false – ложь). Этот параметр определяет, должен ли клонируемый узел включать все атрибуты и дочерние узлы исходного узла.

```
<ul id="list"><li><a href="http://www.samspublishing.com/">Item 1</a></li></ul>

<script>
  var list = document.getElementById("list");
  var oldItem = list.firstChild;
  var newItem = oldItem.cloneNode(true);
  list.appendChild(newItem);
</script>
```

Создание сложных структур

Для создания сложных структур DOM, как правило, используют либо готовый «шаблонный узел» и метод cloneNode, либо свойство innerHTML.

Следующая функция создает сообщение с указанным телом и заголовком.

```
function createMessage(title, body) {
  // (1)
  var container = document.createElement('div');

  // (2)
  container.innerHTML = '<div class="message"> \
    <h1>' + title + '</h1> \
    <div class="content">' + body + '</div> \
    <input class="ok" type="button" value="OK"> \
  </div>';

  // (3)
  return container.firstChild;
}
var messageElem = createMessage('Привет, Мир!', 'Я - элемент DOM!')
document.body.appendChild(messageElem);
```

Чтобы создать элемент по текстовому шаблону, она сначала создает временный элемент container (1), а потом записывает (2) сообщение как innerHTML временного элемента. Теперь готовый элемент можно получить как container.firstChild и вернуть в (3).

Задача 13

Напишите интерфейс для создания многоуровнево списка.

Для каждого пункта:

1) Запрашивайте номер(число) родительского элемента и содержимое пункта у пользователя с помощью prompt().
2) Создавайте пункт добавляйте его в конец списка потомков родительского элемента и установите атрибут id равный номеру следуюшему за последним.
3) Если номер родительского элемента не найден то пункт добавляется в список первого уровня

Все элементы должны создаваться динамически.

Если посетитель вводит теги — в списке они отображаются как обычный текст.

Браузерные события

Для реакции на действия посетителя и внутреннего взаимодействия скриптов существуют события.

Событие - это сигнал от браузера о том, что что-то произошло.

Существует много видов событий.

1. DOM-события, которые инициализируются элементами DOM. Например:
   1) Событие **click** происходит, когда кликнули на элемент
   2) Событие **mouseover** — когда на элемент наводится мышь.
   3) Событие **focus** — когда посетитель фокусируется на элементе.
   4) Событие **keydown** — когда посетитель нажимает клавишу.
2. События для окна браузера. Например, **resize** — когда изменяется размер окна.
3. Есть загрузки файла/документа: **load, readystatechange, DOMContentLoaded**…

События соединяют JavaScript-код с документом и посетителем, позволяя создавать динамические интерфейсы.

Назначение обработчиков событий(HTML)

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Например, чтобы прикрепить click-событие к input кнопке, можно присвоить обработчик onclick. При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <script>
    function showMsg() {
      alert('Сообщение');
    }
  </script>
</head>
<body>
  <input type="button" onclick="showMsg()" />
</body>
</html>
```

```
<input id="my" type="button"
value="Ok"/>
<script>
var elem =
document.getElementById('my');

elem.onclick = function() {
    alert('Сообщение');
}
</script>
```

```
function showMsg() {
  alert('Сообщение');
}

document.getElementById('button')
.onclick = showMsg();
```

Недостатки назначения через onсобытие

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

Например, если необходимо в одной части кода при клике на кнопку делать ее подсвеченной, а в другой — выдавать сообщение. Нужно в разных местах два обработчика повесить.

При этом новый обработчик будет затирать предыдущий. Например, следующий код на самом деле назначает один обработчик — последний:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // заменит предыдущий обработчик
```

Конечно, это можно обойти разными способами, в том числе написанием обертки вокруг обработчиков. Но существует и другой метод назначения обработчиков, который свободен от указанного недостатка.

Назначение обработчиков по стандарту

Официальный способ назначения обработчиков из стандарта W3C работает во всех современных браузерах, включая IE9+.

Назначение обработчика:

```
element.addEventListener( event, handler, phase);
```

Удаление:

```
element.removeEventListener( event, handler, phase);
```

```
function handler() { ... }

elem.addEventListener( "click" , handler, false) // назначение обработчика

elem.removeEventListener( "click", handler, false) // удаление обработчика
```

1) Можно поставить столько обработчиков, сколько вам нужно.
2) Нельзя получить все назначенные обработчики из элемента.
3) Браузер не гарантирует сохранение порядка выполнения обработчиков.

   Они могут быть назначены в одном порядке, а выполниться — в другом.

Получение объекта события

Чтобы хорошо обработать событие, недостаточно знать о том, что это «клик» или «нажатие клавиши». Могут понадобиться детали: координаты курсора, введённый символ и другие, в зависимости от события.

Эти детали браузер записывает в «объект события», к которому может обратиться обработчик.

Некоторые свойства объекта события для события onclick:

1) **event.type** — тип события, равен click.
2) **event.target** — элемент, по которому кликнули.
3) **event.clientX / event.clientY** - координаты курсора в момент клика, относительно окна.
4) и другие свойства

Получение объекта события

Все современные браузеры передают объект события первым аргументом в обработчик.

Для примера:

```
element.onclick = function(event) {
  alert(event.clientX); // вывести координату клика
}
```

IE8- вместо передачи параметра обработчику создаёт глобальный объект window.event.

Универсальное решение для получения объекта события:

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

При назначении обработчика в HTML, можно использовать переменную event, это будет кросс-браузерно:

```
<input type="button" onclick="alert(event.type)" value="Тип события">
```

Всплытие и перехват

Элементы DOM могут быть вложены друг в друга. И каким-то образом, обработчик, привязанный к родителю срабатывает, даже есть вы кликнули по потомку.

Например, обработчик для DIV сработает, если вы кликните по вложенному тегу EM или CODE:

```
<div onclick="alert('Обработчик для Div сработал!')">
  <em>Кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>
</div>
```

Это происходит потому, что событие всплывает.

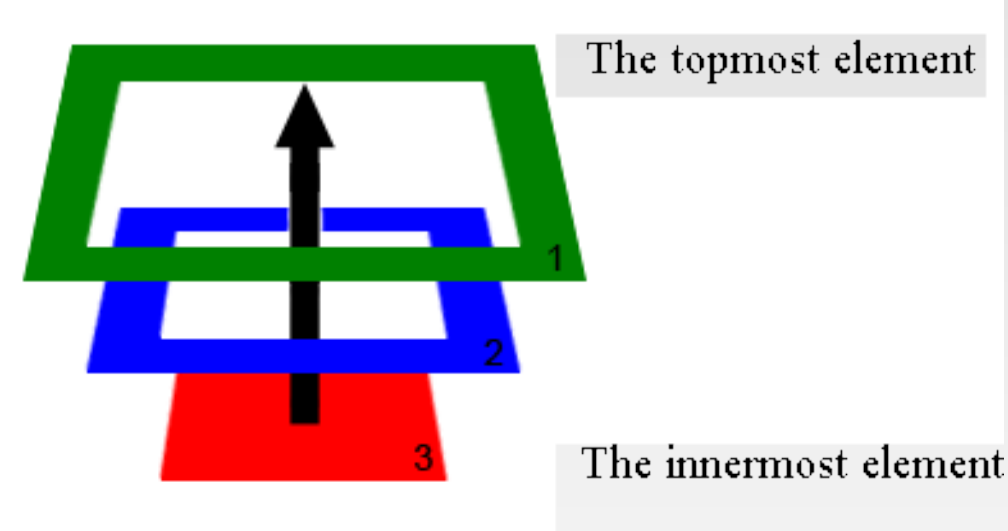
Всплытие

Основной принцип всплытия:

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

Например, есть 3 вложенных блока:

```
<div class="d1">
   1  <!-- внешний (topmost) -->
   <div class="d2">2
      <div class="d3">
         3 <!-- внутренний (innermost) -->
      </div>
   </div>
</div>
```



Всплытие гарантирует, что клик по внутреннем div 3 вызовет событие onclick сначала на внутреннем элементе 3, затем на элементе 2 и в конце концов на элементе 1.

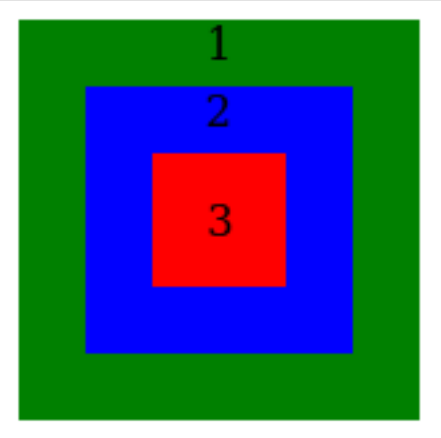
Этот процесс называется всплытием, потому что события «всплывают» от внутреннего элемента вверх через родителей.

Текущий элемент, this

Элемент, на котором сработал обработчик, доступен через this.

Например, повесим клик по каждому DIV функцию highlight, которая подсвечивает текущий элемент:

```
div class="d1" onclick="highlight(this)">1
    <div class="d2" onclick="highlight(this)">2
        <div class="d3" onclick="highlight(this)">3
        </div>
    </div>
</div>

<script>
function highlight(elem) {
    elem.style.backgroundColor = 'yellow';
    alert(elem.className);
    elem.style.backgroundColor = '';
}
</script>
```



Во всех браузерах, кроме IE8-, также существует свойство объекта события event.currentTarget, аналогичное this.

Целевой элемент, event.target

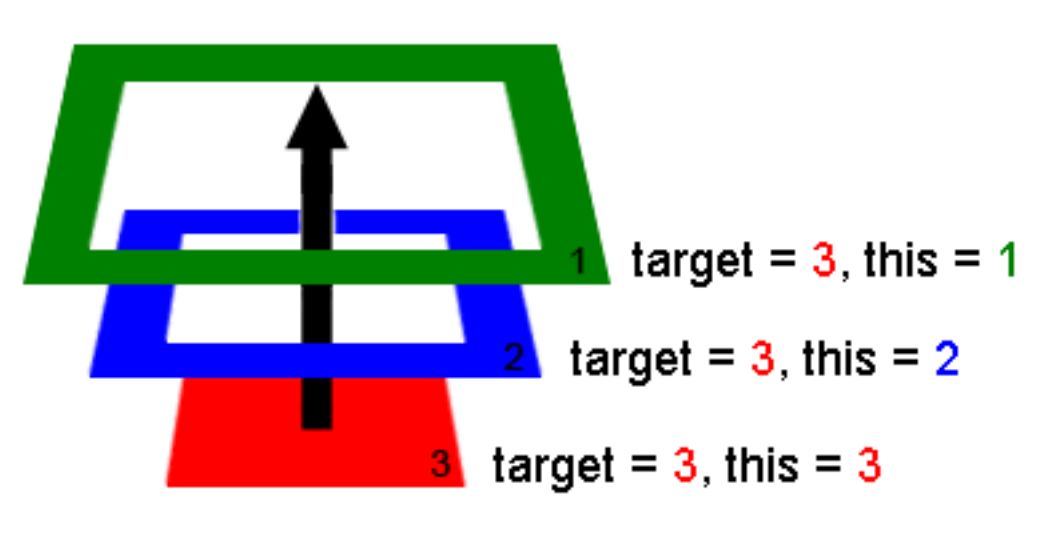
Самый глубокий элемент, который вызывает событие называется «целевым», или «исходным» элементом.

В IE<9 он доступен как event.srcElement, остальные браузеры используют event.target. Кроссбраузерное решение выглядит так:

```
var target = event.target || event.srcElement;
```

1) event.target/srcElement - означает исходный элемент, на котором произошло событие.

2) this - текущий элемент, до которого дошло всплытие и который запускает обработчик.



Прекращения всплытия

Всплытие идет прямо наверх. Обычно оно будет всплывать до <HTML>, а затем до document, вызывая все обработчики на своем пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Код для остановки всплытия различается между IE<9 и остальными браузерами:

1) Стандартный код — это вызов метода:

   event.stopPropagation()

2) Для IE<9 — это назначение свойства:

   event.cancelBubble = true

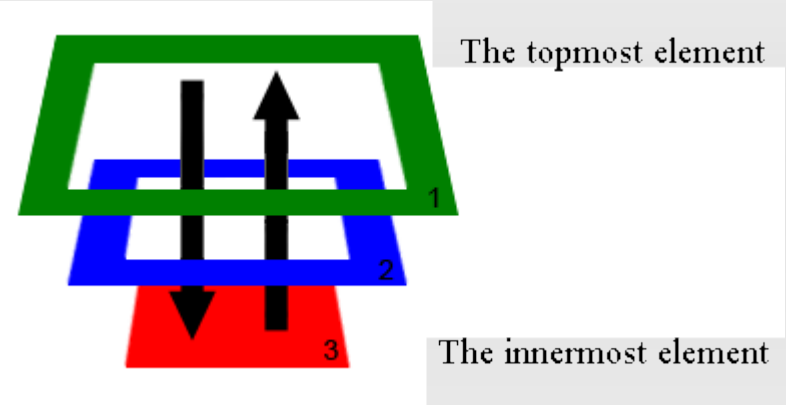
```
element.onclick = function(event) {
  // Кроссбраузерно получить событие
  event = event || window.event;
  // существует ли метод?
  if (event.stopPropagation) {
    // Стандартно:
    event.stopPropagation();
  } else {
    // Вариант IE
    event.cancelBubble = true;
  }
}
```

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. Например, если на ссылке есть два onclick-обработчика, то остановка всплытия для одного из них никак не скажется на другом.

Стадия перехвата

Во всех браузерах, кроме IE<9, есть две стадии прохода события.

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



В соответствии с этой моделью, событие:

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

2) Всплывает вверх - через 3 -> 2 -> 1.

Обычно обработчики событий ничего не знают о стадии перехвата, а начинают работать со всплытия.

Единственный способ поймать событие на стадии перехвата — это 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.

Действия браузера по умолчанию

Многие события влекут за собой действие браузера.

Например, клик по ссылке инициирует переход на новый URL, нажатие на кнопку «отправить» в форме — посылку ее на сервер, и т.п.

Если логика работы обработчика требует отменить действие браузера — это возможно.

Отмена действия браузера

1) Первый способ — это воспользоваться объектом события. Для отмены действия браузера существует стандартный метод event.preventDefault(), ну а для IE<9 нужно назначить свойство event.returnValue = false.

```
element.onclick = function(event) {
  event = event || window.event

  if (event.preventDefault) {  // если метод существует
    event.preventDefault();
  } else { // вариант IE<9:
    event.returnValue = false;
  }
}
```

2) Если обработчик назначен через on..., то return false из обработчика отменяет действие браузера:

```
element.onclick = function(event) {
  ...
  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. Большинство браузеров выделяют такую ссылку пунктирной границей. Можно и указать свой стиль в CSS для псевдоселектора :focus.

Фокусировку нельзя предотвратить из обработчика onfocus, поскольку обработчик вызывается уже после того, как она произошла.

С другой стороны, переход по URL происходит после клика по ссылке, поэтому его можно отменить.

```
<style>
  a:focus { border: 1px solid black }
</style>
<a href="/" onclick="return false" onfocus="return false">
  По клику произойдет фокусировка, но перехода не будет
</a>
```

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.

Эта техника называется делегированием и очень активно применяется в современном JavaScript.

```
<ul id="menu">
  <li><a href="/php">PHP</a></li>
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/flash">Flash</a></li>
</ul>
<script>
  document.getElementById('menu').onclick = function(e) {
    var target = e && e.target || event.srcElement;
    if (target.nodeName != 'A') return;
    var href = target.getAttribute('href');
    alert(href);
    return false // prevent url change
  };
</script>
```

Применение делегирования: действия в разметке

Делегирование позволяет использовать обработчик и для абсолютно разных действий.

Например, нам нужно сделать меню с разными кнопками: «Сохранить», «Загрузить», «Поиск» и т.д. Решить задачу можно путем добавления одного обработчика на всё меню. Все клики внутри меню попадут в обработчик. Но как нам узнать, какую кнопку нажали и как обработать это событие? Эту задачу мы можем решить, добавив каждой кнопке нужный нам метод в специальный атрибут, который назовем data-action (можно придумать любое название, но data-* является валидным в HTML5). Обработчик считывает содержимое атрибута и выполняет метод.

```
<script>
function Menu(elem) {
  this.save = function() { alert('сохраняю'); };
  this.load = function() { alert('загружаю'); };
  var self = this;
  elem.onclick = function(e) {
    var target = e && e.target || event.srcElement;
    var action = target.getAttribute('data-action');
    if (action) {
      self[action]();
    }
  };
}
new Menu(document.getElementById('menu'));
</script>
```

```
<div id="menu">
  <button data-action="save">
    Нажмите, чтобы Сохранить
  </button>
  <button data-action="load">
    Нажмите, чтобы Загрузить
  </button>
</div>
```

Обратите внимание, как используется трюк с var self = this, чтобы сохранить ссылку на объект Menu. Иначе обработчик просто бы не смог вызвать методы Menu, потому что его собственный this ссылается на элемент.

Главный цикл браузера

В каждом окне выполняется только один главный поток, который занимается выполнением JavaScript, отрисовкой и работой с DOM.

Он выполняет команды последовательно и блокируется при выводе модальных окон, таких как alert.

Есть и другие потоки, например для сетевых коммуникаций, поэтому скачивание файлов может продолжаться пока основной поток ждёт реакции на alert. Но это скорее исключение, чем правило.

Асинхронные события

Большинство событий — асинхронны.

Когда происходит асинхронное событие, оно попадает в очередь.

Внутри браузера существует главный внутренний цикл, который проверяет очередь и обрабатывает события, запускает соответствующие обработчики и т.п.

События добавляются в очередь в порядке поступления. Иногда даже сразу пачкой.

Например, при клике на элементе генерируется несколько событий:

1) Сначала mousedown — нажата кнопка мыши.
2) Затем mouseup — кнопка мыши отпущена.
3) Так как это было над одним элементом, то дополнительно генерируется click

Все асинхронные события обрабатываются последовательно.

Основной поток, когда освобождается, смотрит в очередь, берёт оттуда событие, обрабатывает, затем получает из очереди следующее и так далее. Каждое событие обрабатывается полностью отдельно от других.

Вложенные (синхронные) события

Некоторые события обрабатываются синхронно, то есть поток выполнения переходит в их обработчик тут же.

Как правило, синхронными будут события, которые инициируются вызовом метода. Исключением здесь является Internet Explorer (включая 10), в котором почти все события асинхронные.

Пример: событие **onfocus** Когда посетитель фокусируется на элементе, возникает событие onfocus. Но фокусировку можно вызвать и явно, вызовом метода elem.focus().

```
<input type="button" value="Нажми меня">
<input type="text" size="60">
<script>
  var button = document.body.children[0];
  var text = document.body.children[1];
  button.onclick = function() {
    text.value += ' ->в onclick ';
    text.focus(); // вызов инициирует событие onfocus
    text.value += ' из onclick-> ';
  };
  text.onfocus = function() {
    text.value += ' !focus! ';
  };
</script>
```

событие onfocus — синхронное. Его обработчик onfocus не ждёт завершения onclick, а выполняется тут же:

```
->в onclick  !focus!  из onclick->
```

setTimeout

```
timeout_id = window.setTimeout(func|code, delay)
```

1) func|code - Функция или строка кода для выполнения
2) delay - Задержка в миллисекундах, т.е 1000 это 1 секунда

Этот метод выполняет код(или функцию), указанный в первом аргументе, асинхронно, с задержкой в delay миллисекунд.В каком виде указывать первый параметр - в виде строки кода или функции - разницы нет.Следующие два вызова работают одинаково:

```
// первый аргумент - строка
setTimeout('alert("прошла секунда")', 1000)
// первый аргумент - функция
function second_passed() {
alert("прошла секунда")
}
setTimeout(second_passed, 1000)
```

При указании строки кода - интерпретатор динамически создает анонимную функцию с телом из данной строки. Но более правильным считается объявление функции в явном виде, например так:

```
setTimeout(function() { alert('0.5 секунды') },
                    500)
```

Вы можете отменить выполнение setTimeout при помощи clearTimeout, используя для этого идентификатор таймаута.

```
var timeout_id = setTimeout(...)
clearTimeout(timeout_id)
```

setInterval

```
intervalID = window.setInterval(func|code, delay)
```

1) func|code - Функция или строка кода для выполнения
2) delay - Интервал в миллисекундах, т.е 1000 это 1 секунда

В отличие от метода setTimeout, setInterval выполняет код много раз, через равные промежутки времени, пока не будет остановлен при помощи clearInterval. В каком виде указывать первый параметр - разницы нет. При указании строки кода - динамически создается функция с телом code.

Следующие вызовы почти одинаковы:

```
// (1)
setInterval('alert("прошла секунда")', 1000)

// (2)
function sec() {
 alert("прошла секунда")
}
setInterval(sec, 1000) // использовать функцию
```

```
setInterval(function() { alert(...) }, 1000)
```

Вы можете отменить выполнение setInterval при помощи clearInterval, используя для этого идентификатор intervalID.

```
var intervalID = setInterval(...)
clearInterval(intervalID)
```

Задача 14

Измените интерфейс для создания многоуровнево списка.

1) Создайте кнопку для добавления элемента списка(при нажатии на кнопку должны появляться запросы родительского элемента и текста для элемента)
2) К каждому элементу списка добавьте кнопку удаления элемента

className

Свойство className соответствует HTML-атрибуту class.

Пример его использования (добавление класса):

```
<body class="class1 class2">
<script>
  alert(document.body.className);
  document.body.className += ' class3';
</script>
</body>
```

Так как классы перечислены через пробел, то для удаления можно производить строковые операции, например с использованием регулярных выражений, а можно просто разбить строку по пробелу.

classList

Свойство classList поддерживается во всех современных браузерах, в IE начиная с IE10, но его можно эмулировать в IE8+.

Оно предоставляет удобный интерфейс для работы с классами элемента:

1) **elem.classList.contains(cls)** — проверяет, есть ли у элемента класс cls
2) **elem.classList.add/remove(cls)** — добавляют/удаляют класс cls
3) **elem.classList.toggle(cls)** — добавляет класс cls, если его нет, а если есть — удаляет.

style

Свойство style дает доступ к стилю элемента. Это свойство можно как читать, так и править.

С помощью него можно изменять большинство CSS-свойств, например element.style.width='100px' работает так, как будто у элемента есть атрибут style="width:100px". Также, как и в CSS, вам нужно указывать единицы измерения, например px.

Для свойств, названия которых состоят из нескольких слов, используется CamelCase нотация:

background-color => backgroundColor

z-index => zIndex

border-left-width => borderLeftWidth

```
element.zIndex = 10000;
element.style.cssFloat

// введите пустое значение,
//чтобы сбросить цвет
document.body.style.backgroundColor
                = prompt('background color?', 'green')
```

Чтобы убрать поставленный стиль, присваивают style пустую строку: elem.style.width=''.

Например, для того чтобы спрятать элемент можно присвоить: **elem.style.display = "none"**.

style.cssText

Свойство style является специальным объектом, который нельзя присваивать как строку. Запись div.style="color:blue" работать не будет. Для решения такой задачи стандарт предусматривает специальное свойство style.cssText. Именно оно позволяет поставить стиль целиком в виде строки.

```
<div>Button</div>
<script>
  var div = document.body.children[0];

  div.style.cssText="color: red !important; \
    background-color: yellow; \
    width: 100px; \
    text-align: center; \
    blabla: 5; \
  ";

  alert(div.style.cssText);
</script>
```

Браузер разбирает строку style.cssText и применяет известные ему свойства. При установке style.cssText все существующие свойства style перезаписываются. Поэтому, по возможности, во избежание конфликта, присваивают более конкретные подсвойства style: style.color, style.width и т.п, а style.cssText используют для более короткой записи, когда это заведомо безопасно.

Получение информации о style

Свойство style позволяет читать информацию о стиле, но лишь ту, которая доступна напрямую из свойства/атрибута "style". Если же стиль получен из CSS, особенно когда задано составное свойство (margin вместо margin-top), то style «не знает» об этом.

```
<body style="color:red">
  Красный текст
  <script>
    alert(document.body.style.color); // red
  </script>
</body>
```

```
<style>
  body { margin: 10px }
</style>
<body>
  <script>
    alert(document.body.style.marginTop);
//в большинстве браузеров ничего не выведет
  </script>
</body>
```

```
//если сначала присвоить свойство
// через style, а затем прочесть
<body>
  <script>
    document.body.style.margin = '20px';
    alert(document.body.style.marginTop);
// 20px!
  </script>
</body>
```

```
//Свойство в стиле может быть показано не в
//точности, как его назначали, а как его
//воспринял браузер.
<body style="color:#abc">
  <script>
    alert(document.body.style.color);
// rgb(170, 187, 204)
  </script>
</body>
```

getComputedStyle и currentStyle

Свойство style дает доступ только к той информации, которые хранится в elem.style.

Он не скажет ничего об отступе, если он появился в результате наложения CSS или встроенных стилей браузера:

```
<style>  body { margin: 10px } </style>
<body>
  <script>
    alert(document.body.style.marginTop); // пустая строка
  </script>
</body>
```

Для того, чтобы получить текущее «вычисленное» значение свойства, используется метод window.getComputedStyle.

getComputedStyle(element, pseudo)

element

Элемент, значения для которого нужно получить

pseudo

Указывается, если нужен стиль псевдо-элемента, например "::before". Пустая строка означает сам элемент.

CSS-метрики width/height

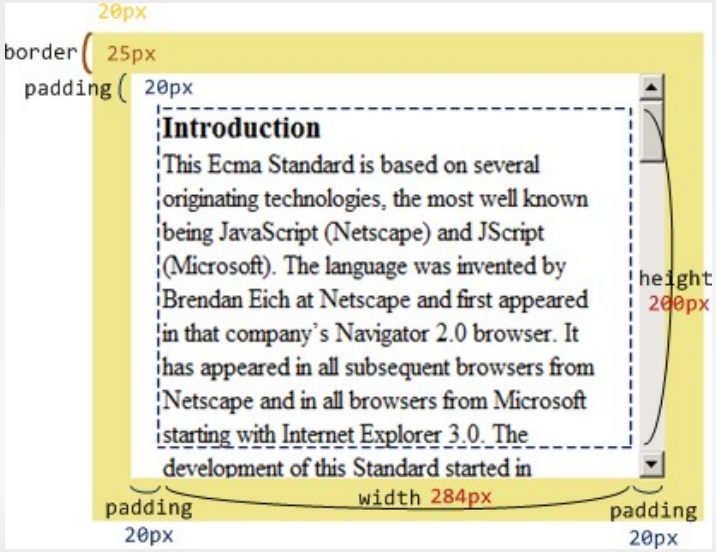
CSS-высоту и ширину width/height можно установить с помощью elem.style и извлечь, используя getComputedStyle()/currentStyle:

```
style = window.getComputedStyle ? getComputedStyle(elem, '') : elem.currentStyle;

alert(style.width); // вывести CSS-ширину
```

Они, при стандартном значении CSS-свойства box-sizing, относятся к размеру внутренней части элемента, которая лежит внутри padding. На рисунке нижнее поле padding заполнено прокруткой, но оно всё равно подразумевается и не входит в высоту height.

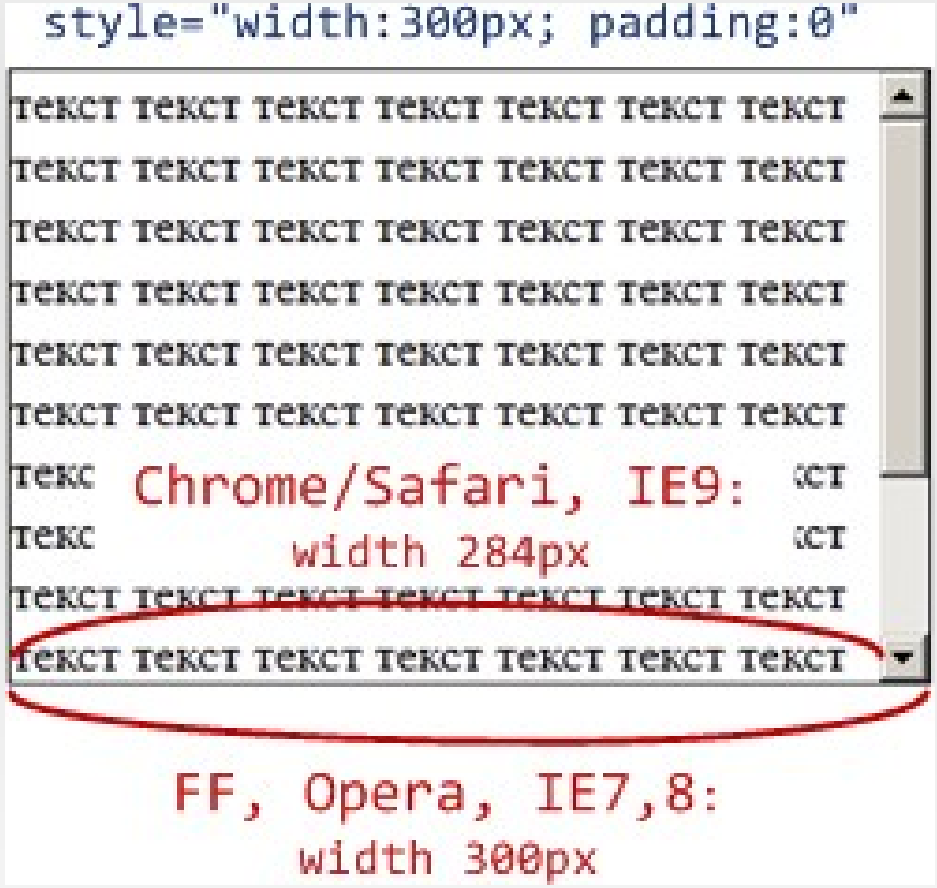
Заметим, что эти метрики нельзя использовать для надежного получения ширины и высоты произвольного элемента!

```
<span>Привет!</span>
<div>Мир!</div>
<script>
  var span = document.body.children[0];
  var div = document.body.children[1];

  alert( getComputedStyle(span, '').width );  // auto
  alert( getComputedStyle(div, '').width );  // ..px
</script>
```



Документ выше выводит width у span правильно, ведь это inline-элемент, и в блочной модели CSS его размер зависит от содержимого.

Полоса прокрутки



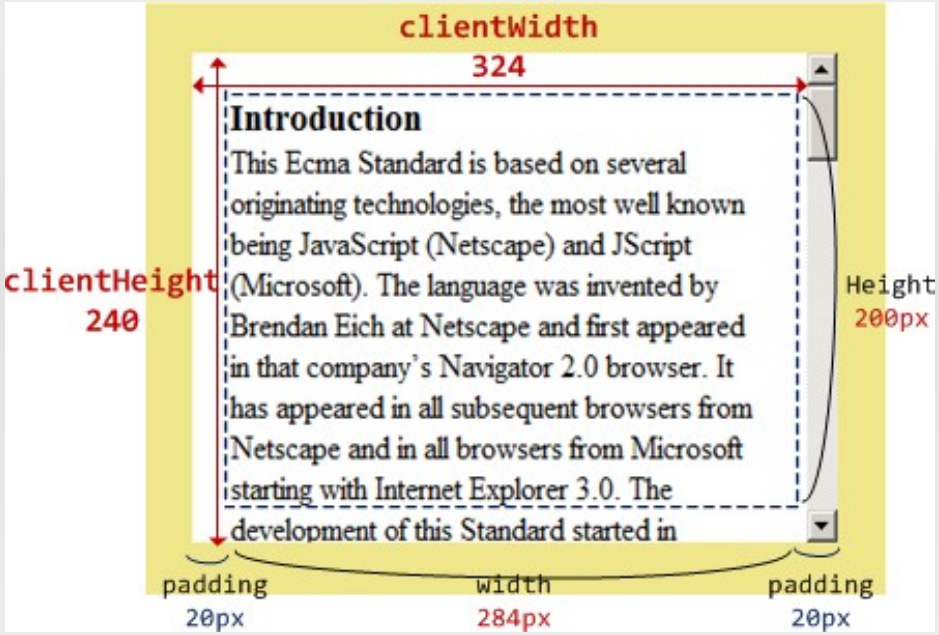
При наличии вертикальной полосы прокрутки — она забирает себе часть ширины элемента.

На рисунке ниже в CSS указано width:300px. А вот getComputedStyle может возвращать 300px или 284px

Ширина полосы прокрутки обычно составляет 16px, 18px, в зависимости от браузера и операционной системы. Бывает и 0 для полупрозрачной прокрутки, не отъедающей место (Chrome под MacOS). В примере подразумевается система Windows, поэтому внутренняя область будет уже не 300px, а 284px.

Ширина текста при наличии прокрутки уменьшается до 284px.

JS метрики clientWidth/Height



В JavaScript существует ряд дополнительных свойств, содержащих размеры элементов. Мы будем называть их «метриками».Метрики JavaScript, в отличие от свойств CSS, содержат числа, всегда в пикселях и без единиц измерения на конце.

clientWidth/Height - Размер клиентской зоны, а именно: внутренняя область плюс padding.

Общая ширина внутри рамки — это

284 (width) + 20(padding left) + 20 (padding right) = 324.

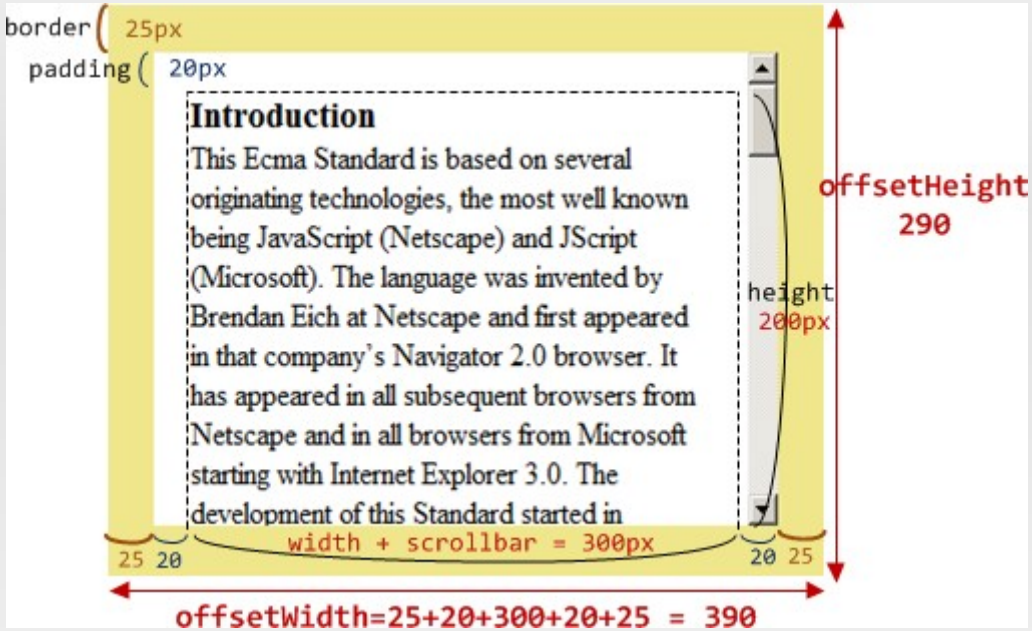
Получаем:

clientWidth = 284(width) + 2*20(padding) = 324

clientHeight = 200(height) + 2*20(padding) = 240

Если padding нет, то clientWidth/Height покажет реальный размер области данных, внутри рамок и полосы прокрутки.

JS метрики offsetWidth/Height

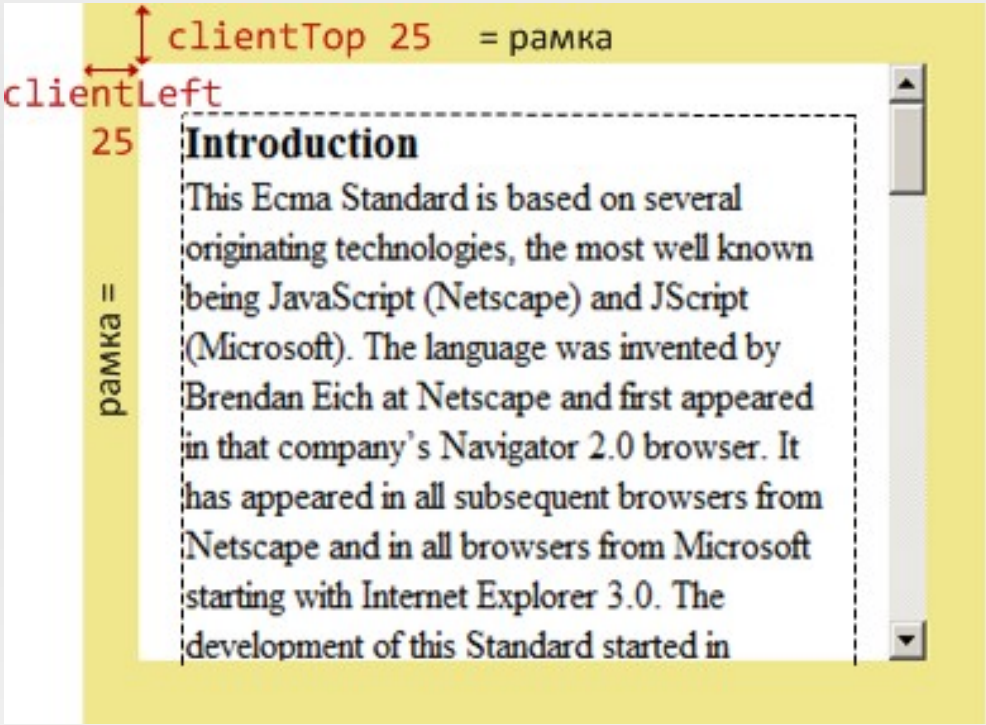


offsetWidth/Height - Внешняя ширина/высота блока, полный размер, включая рамки, исключая внешние отступы margin.

1) offsetWidth = 390 — внешняя ширина блока
2) offsetHeight = 290 — внешняя высота блока

Эти свойства показывают внешние ширину и высоту блока, то как блок выглядит снаружи.

JS метрики clientTop/Left

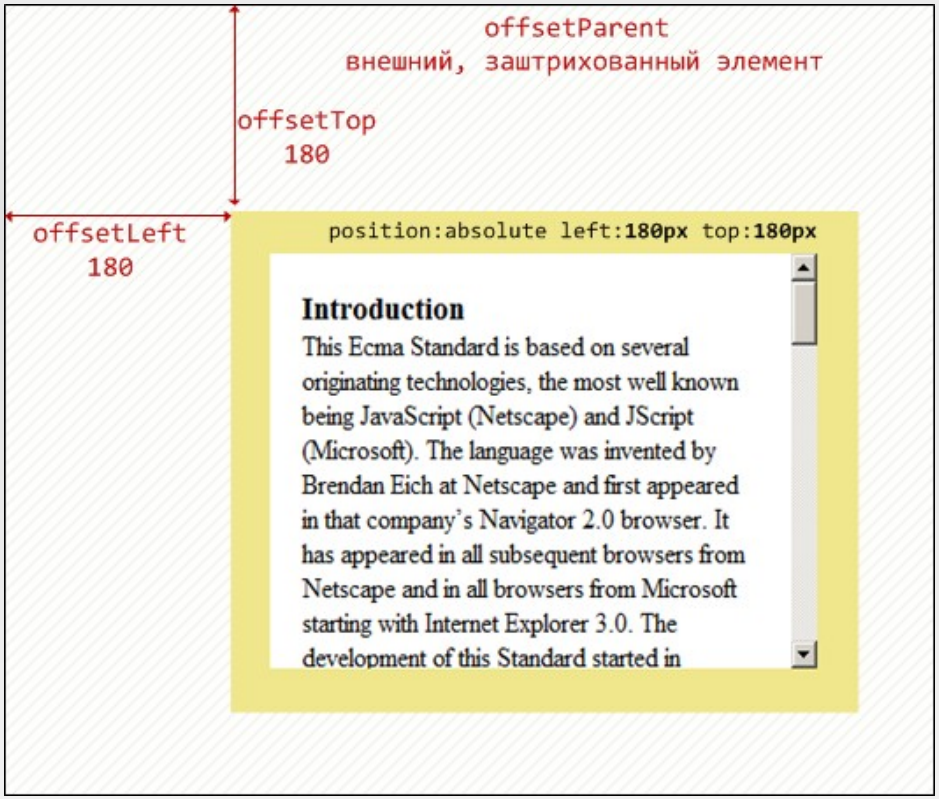


clientTop/Left - Отступ клиентской области от внешнего угла блока.

Другими словами, это ширина верхней/левой рамки(border) в пикселях.

1) clientLeft = 25 — ширина левой рамки
2) clientTop = 25 — ширина верхней рамки

JS метрики offsetParent, offsetLeft/Top



offsetParent — это родительский элемент в смысле отображения на странице.

Например, если элемент позиционирован абсолютно, то таким родителем offsetParent является ближайший позиционированный элемент (т.е. свойство position не равно static), или BODY, если таковой отсутствует.

Свойства offsetLeft/Top задают смещение относительно offsetParent.

```
<div style="position: relative">
  <div style="position: absolute; left: 180px;
top: 180px">...</div>
</div>
```

Невидимые элементы

Метрики для невидимых элементов равны нулю.

Координаты и размеры в JavaScript устанавливаются только для видимых элементов.

Они равны 0 для элементов с display:none или тех, которые вне DOM. Кстати, offsetParent для таких элементов тоже null.

Это дает нам замечательный способ для проверки, виден ли элемент:

```
function isHidden(elem)
  return !elem.offsetWidth && !elem.offsetHeight;
}
```

1) Работает, даже если родителю элемента установлено свойство display:none.
2) Работает для всех элементов, кроме TR, с которым возникают некоторые проблемы в разных браузерах.
3) Считает элемент видимым, даже если позиционирован за пределами экрана или имеет свойство visibility:hidden.
4) «Схлопнутый» элемент, например пустой div без высоты и ширины, будет считаться невидимым.

Координаты

В браузере есть три координатные системы.

1) Относительно документа pageX/pageY

   Точка начала координат лежит в левом верхнем углу страницы.

2) Относительно окна clientX/clientY

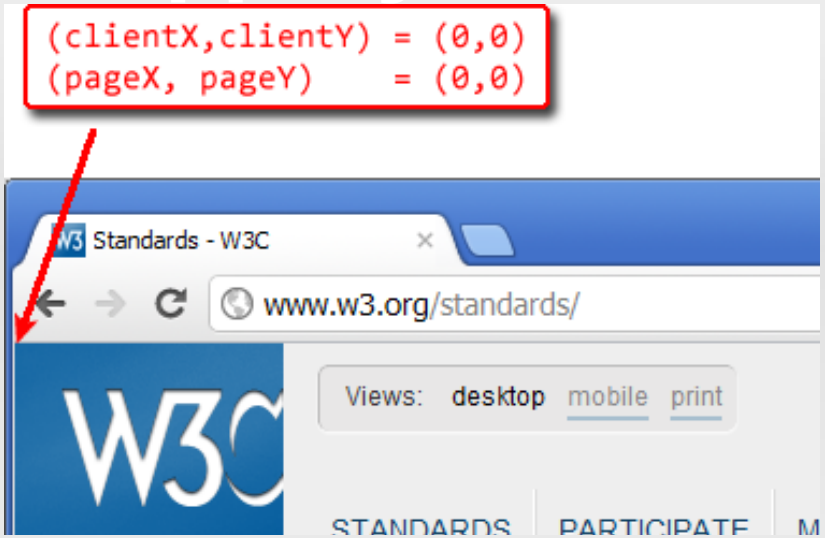
   Начало координат лежит в левом верхнем углу текущей видимой области.

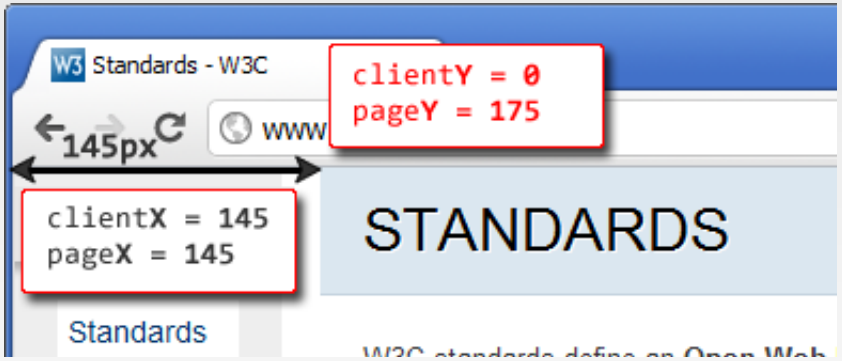
3) Относительно экрана screenX/screenY

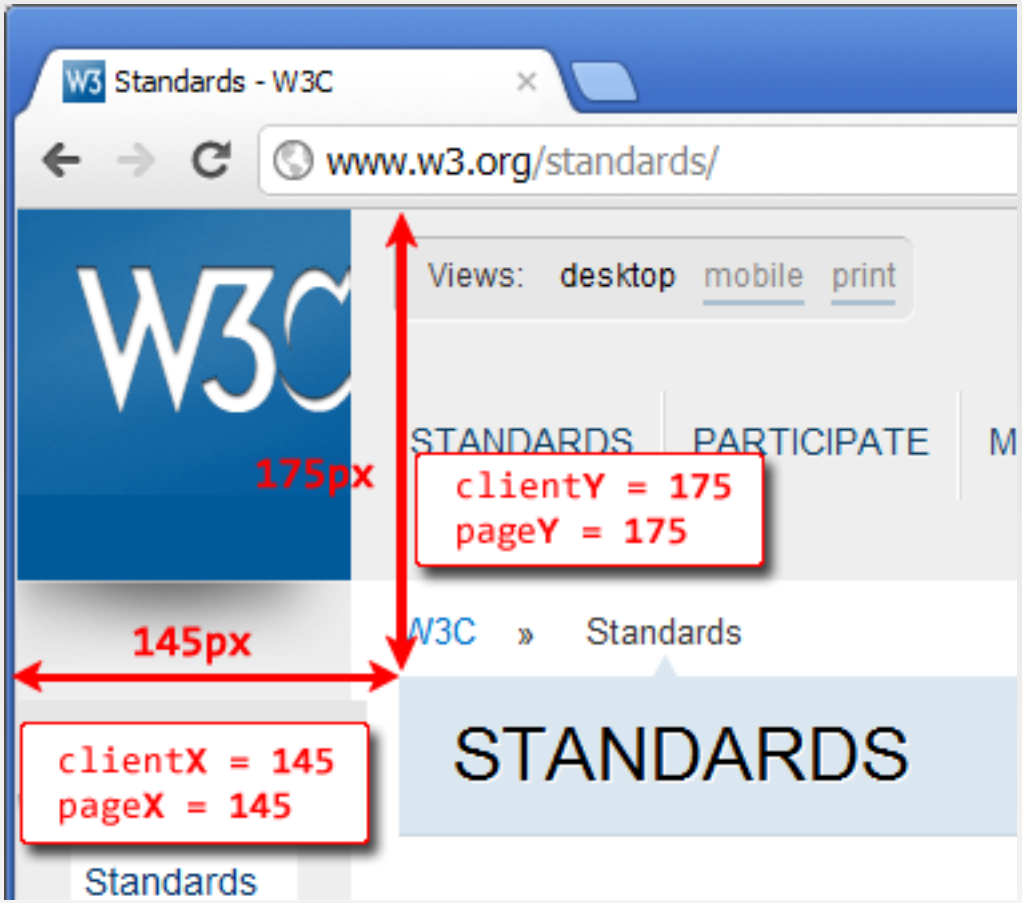
   Начало координат — левый-верхний угол экрана.

Координатами элемента, для краткости, называются координаты его левого верхнего угла.

Координаты относительно окна и документа







Итак, координаты pageX,pageY не меняются при прокрутке, в отличие от clientX,clientY. Можно сказать и по-другому. Разность pageY-clientY — это в точности размер текущей прокрученной области.

Координаты в окне: elem.getBoundingClientRect()

Он возвращает размеры прямоугольника, который охватывает элемент, в виде объекта со свойствами: top, left, right, bottom.

Эти 4 числа — координаты верхнего левого и нижнего правого углов элемента относительно окна.

Координаты elem.getBoundingClientRect() — относительно окна, а не документа.

Например, если вы прокрутите эту страницу так, что кнопка будет становить ближе к верху окна, то ее top-координата будет стремиться к 0, потому что он считается относительно окна.

Чтобы посчитать ее относительно документа, нужно добавить к ней величину прокрутки.

Координаты в документе

Готовой функции для координат элемента в документе нет. Но её можно написать самим.

Наша функция getCoords(elem) будет брать результат elem.getBoundingClientRect() и прибавлять текущую прокрутку документа.

Результат: объект с координатами {left: .., top: ..}

```
function getCoords(elem) {
    // (1)
    var box = elem.getBoundingClientRect();
    var body = document.body;
    var docEl = document.documentElement;
    // (2)
    var scrollTop = window.pageYOffset || docEl.scrollTop || body.scrollTop;
    var scrollLeft = window.pageXOffset || docEl.scrollLeft || body.scrollLeft;
    // (3)
    var clientTop = docEl.clientTop || body.clientTop || 0;
    var clientLeft = docEl.clientLeft || body.clientLeft || 0;
    // (4)
    var top  = box.top +  scrollTop - clientTop;
    var left = box.left + scrollLeft - clientLeft;
    // (5)
    return { top: Math.round(top), left: Math.round(left) };
}
```

Получение элемента по координатам:elementFromPoint(x,y)

Обратная задача — «получить элемент по координатам в документе» появится при работе с событиями мыши.

Для её решения есть метод document.elementFromPoint(clientX, clientY).

Параметры clientX/clientY — координаты относительно окна.

Код функции подсветки:

```
/* x,y - координаты относительно окна */
function highlightRedXY(x,y) {
  var el = document.elementFromPoint(x, y);
  el.style.backgroundColor = (el.style.backgroundColor == 'red') ? '' : 'red';
}
```

Обратите внимание — при прокрутке страницы на тех же координатах будут новые элементы. Это как раз потому, что координаты в этой функции — относительно окна, а не документа.

Координаты на экране screenX/screenY

Координаты относительно экрана screenX/screenY отсчитываются от его левого-верхнего угла. Имеется в виду именно весь экран, а не окно браузера.

Такие координаты редко используются. Но они могут быть полезны, например, при работе с мобильными устройствами.

Общая информация об экране хранится в глобальной переменной screen

```
// общая ширина/высота
alert( screen.width + ' x ' + screen.height );

// доступная ширина/высота (за вычетом таскбара и т.п.)
alert( screen.availWidth + ' x ' + screen.availHeight);

// есть и ряд других свойств screen (см. документацию)
```

Координаты левого-верхнего угла браузера на экране хранятся в window.screenX, window.screenY (не поддерживаются IE<9):

```
alert("Браузера находится на " + window.screenX + "," + window.screenY);
```

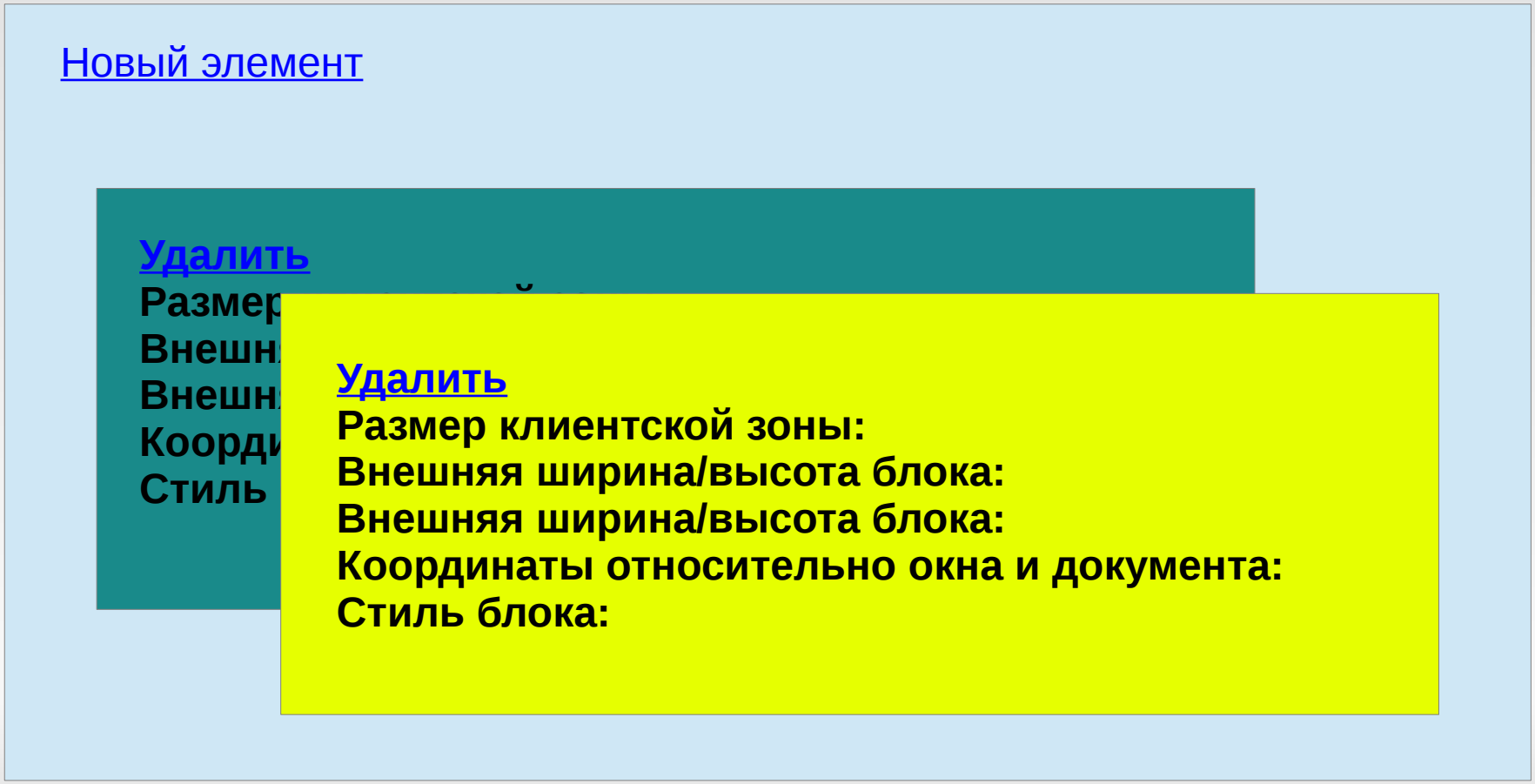
Они могут быть и меньше нуля, если окно частично вне экрана.

Координаты DOM-элемента относительно экрана получить нельзя. Браузер не предоставляет свойств и методов для этого.

Задача 15

Реализуйте заполнение окна документа блоками с произвольно заданными параметрами (геометрическими характеристиками, цветом, положением на экране). Заданые параметры должны отображаться внутри каждого блока.

Реализуйте возможность интерактивного изменения установленных параметров.



Типы событий мыши

Простые события

1) **mousedown** Кнопка мыши нажата над элементом.
2) **mouseup** Кнопка мыши отпущена над элементом.
3) **mouseover** Мышь появилась над элементом.
4) **mouseout** Мышь ушла с элемента.
5) **mousemove** Каждое движение мыши над элементом генерирует это событие.

Комплексные события

6) **click** Вызывается при клике мышью, то есть при mousedown, а затем mouseup на одном элементе
7) **contextmenu** Вызывается при клике правой кнопкой мыши на элементе.
8) **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. То есть, обработчики вызовутся в порядке mousedown -> mouseup -> click.

Каждое событие обрабатывается независимо. Например, при клике события mouseup + click возникают одновременно, но обрабатываются последовательно. Сначала полностью завершается обработка mouseup, затем запускается click.

Получение информации о кнопке: which/button

При обработке событий, связанных с кликами мыши, бывает важно знать, какая кнопка нажата.

Для получения кнопки мыши в объекте event есть два свойства: which и button. Они хранят кнопку в численном значении, хотя и с некоторыми браузерными несовместимостями.

На практике они используются редко, т.к. обычно обработчик вешается либо onclick — только на левую кнопку мыши, либо oncontextmenu — только на правую.

По стандарту, у мышиных событий есть свойство which, которое работает одинаково во всех браузерах, кроме IE<9, имеет следующие значения:

1) which == 1 - левая кнопка
2) which == 2 - средняя кнопка
3) which == 3 - правая кнопка

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:

1) shiftKey
2) altKey
3) ctrlKey
4) metaKey (для Mac)

Например, кнопка ниже сработает только на Ctrl+Shift+Клик:

```
<button>Ctrl+Shift+Кликни меня!</button>
<script>
  document.body.children[0].onclick = function(e) {
    e = e || window.event;;
    if (!e.ctrlKey || !e.shiftKey) return;
    alert('Ура!');
  }
</script>
```

Координаты мыши относительно окна: clientX/Y

Есть отличное кросс-браузерное свойство clientX(clientY), которое содержит координаты относительно window.

При этом, например, если ваше окно размером 500x500, а мышь находится в центре, тогда и clientX и clientY будут равны 250.

Если прокрутите вниз, влево или вверх не сдвигая при этом мышь, то значения clientX/clientY не изменятся, потому что они считаются относительно окна, а не документа.

```
<input onmousemove="this.value = event.clientX+':'+event.clientY">
```

Координаты относительно документа: pageX/Y

Обычно, для обработки события нам нужно знать позицию мыши относительно документа, вместе со скроллом. Для этого стандарт W3C предоставляет пару свойств pageX/pageY.

Если ваше окно размером 500x500, а мышь находится в центре, тогда и pageX и pageY будут равны 250. Если вы прокрутите страницу на 250 пикселей вниз, то значение pageY станет равным 500.

Итак, пара pageX/pageY содержит координаты относительно левого верхнего угла документа, вместе со всеми прокрутками.

Эти свойства поддерживаются всеми браузерами, кроме IE<9.

```
<input onmousemove="this.value = event.pageX+':'+event.pageY">
```

События mouseover/mouseout, свойство relatedTarget

Событие mouseover происходит, когда мышь появляется над элементом, а mouseout — когда уходит из него.

В этих событиях мышь переходит с одного элемента на другой. Оба этих элемента 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).

Как вы видите, спецификация W3C объединяет fromElement и toElement в одно свойство relatedTarget, которое работает, как fromElement для mouseover и как toElement для mouseout.

Частота событий mousemove и mouseover

Событие mousemove срабатывает при передвижении мыши. Но это не значит, что каждый пиксель экрана порождает отдельное событие!

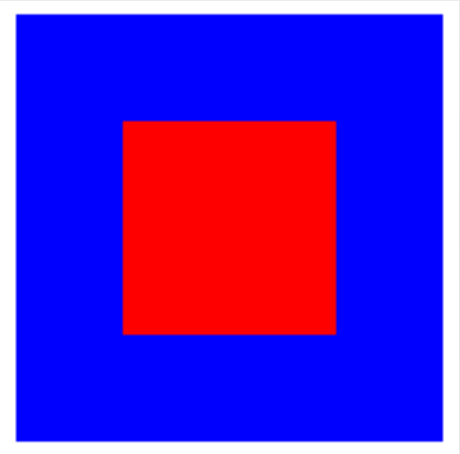
Курсор может быстро перейти над родительским элементом в дочерний, при этом не вызвав событий на родителе, как будто он через родителя никогда не проходил.

Курсор может быстро выйти из дочернего элемента без генерации событий на родителе.

Несмотря на некоторую концептуальную странность такого подхода, он весьма разумен. Хотя браузер и может пропустить промежуточные элементы, он гарантирует, что если уж мышь зашла на элемент (сработало событие mouseover), то при выходе с него сработает и mouseout.

То есть, события позволяют надёжно обрабатывать заход на элемент и уход с него.

mouseout при уходе на потомка



Представьте ситуацию — курсор зашел на элемент. Сработал mouseover на нём. Потом курсор идёт на дочерний… И, оказывается, на элементе-родителе при этом происходит mouseout! Как будто курсор с него ушёл, хотя он всего лишь перешёл на потомка.

Это происходит потому, что согласно браузерной логике, курсор мыши может быть только над одним элементом — самым глубоким в DOM (и верхним по z-index).

Так что если он перешел на потомка — значит ушёл с 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.**

То есть, эти события более интуитивно понятны. 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.**

Прокрутки

Несмотря на то, что колёсико мыши обычно ассоциируется с прокруткой, это совсем разные вещи.

1) При прокрутке срабатывает событие onscroll. Оно произойдёт при любой прокрутке, в том числе через клавиатуру, но только на прокручиваемых элементах. Например, элемент с overflow: hidden в принципе не может сгенерировать onscroll.

2) А событие wheel генерируется над любым элементом при передвижении колеса мыши. При этом не важно, прокручиваемый он или нет. В частности, overflow:hidden никак не препятствует обработке колеса мыши.

Кроме того, событие onscroll происходит после прокрутки, а onwheel — до прокрутки, поэтому в нём можно отменить саму прокрутку (действие браузера).

Основы Drag'n'Drop

Drag’n’Drop — это возможность захватить мышью элемент и перенести его.

Для организации Drag’n’Drop нужно:

1) При помощи события mousedown отследить нажатие на переносимом элементе
2) При нажатии — подготовить элемент к перемещению: дать ему position:absolute. При этом показать его на том же месте, назначив left/top по координатам курсора. Далее отслеживать движение мыши через mousemove и передвигать переносимый элемент по странице. Элемент уже имеет position:absolute, так что это делается путем смены left/top.
3) При отпускании кнопки мыши, то есть наступлении события mouseup — остановить перенос элемента и произвести все действия, связанные с окончанием Drag’n’Drop.

Пример реализации Drag'n'Drop

```
var ball = document.getElementById('ball2');

ball.onmousedown = function(e) {
  var self = this;

  this.style.position = 'absolute';
  moveAt(e);
  document.body.appendChild(this);

  this.style.zIndex = 1000;

  function moveAt(e) {
    self.style.left = e.pageX-25+'px';
    self.style.top = e.pageY-25+'px';
   }

  document.onmousemove = function(e) { moveAt(e); }

  this.onmouseup = function() { document.onmousemove = self.onmouseup = null; }
}

ball.ondragstart = function() {  return false; };
```

События клавиатуры keydown и keyup

События **keydown/keyup** происходят при нажатии/отпускании клавиши и позволяют получить её скан-код в свойстве keyCode.

Скан-код клавиши одинаков в любой раскладке и в любом регистре. Например, клавиша z может означать символ "z", "Z" или "я", "Я" в русской раскладке, но её скан-код будет всегда одинаков: 90.

В действии:

```
<input onkeydown="this.nextSibling.innerHTML = event.keyCode"><b></b>
```

Скан-коды

Для буквенно-цифровых клавиш, скан-код будет равен коду соответствующей заглавной английской буквы/цифры.

Например, при нажатии клавиши S (не важно, каков регистр и раскладка) её скан-код будет равен "S".charCodeAt(0).

С другими символами, например, знаками пунктуации, ситуация тоже понятна. Есть таблица кодов или же можно нажать на нужную клавишу на тестовом стенде и получить код.

Когда-то в этих кодах была масса кросс-браузерных несовместимостей. Сейчас всё проще — таблицы кодов в различных браузерах почти полностью совпадают.

keypress

Событие keypress возникает сразу после keydown, если нажата символьная клавиша, т.е. нажатие приводит к появлению символа.

Любые буквы, цифры генерируют keypress. Управляющие клавиши, такие как Ctrl, Shift, F1, F2.. — keypress не генерируют.

Событие keypress позволяет получить код символа. В отличие от скан-кода, он специфичен именно для символа и различен для "z" и "я".

Код символа хранится в свойствах: charCode и which.

Функция, позволяющая получить код везде.

```
function getChar(event) {
  if (event.which == null) {  // IE
    if (event.keyCode < 32) return null; // спец. символ
    return String.fromCharCode(event.keyCode)
  }
  if (event.which!=0 && event.charCode!=0) { // все кроме IE
    if (event.which < 32) return null; // спец. символ
    return String.fromCharCode(event.which); // остальные
  }
  return null; // спец. символ
}
<input onkeypress="this.nextSibling.innerHTML = getChar(event)+""><b></b>
```

Отмена пользовательского ввода

Появление символа можно предотвратить, если отменить действие браузера на keydown/keypress:

```
<input onkeydown="return false" type="text" size="30">

<input onkeypress="return false" type="text" size="30">
```

Отмена специальных действий

Отменять можно не только символ, а любое действие клавиш.

Например, при отмене Backspace — символ не удалится. А при отмене Page Down — страница не прокрутится.

Конечно же, есть действия, которые в принципе нельзя отменить, в первую очередь — те, которые происходят на уровне операционной системы. Комбинация Alt+F4 инициирует закрытие браузера в Windows, что бы мы ни делали в JavaScript.

В примере ниже, действие браузера отменяется с помощью return false, а вместо него в input добавляется значение в верхнем регистре:

```
<input id="my" type="text" size="2">
<script>
document.getElementById('my').onkeypress = function(e) {
  e = e || window.event;
  // спец. сочетание - не обрабатываем
  if (e.ctrlKey || e.altKey || e.metaKey) return;
  var char = getChar(event || window.event);
  if (!char) return; // спец. символ - не обрабатываем
  this.value = char.toUpperCase();
  return false;
}
</script>
```

Порядок наступления событий

События keydown/keyup возникают всегда. А keypress — по-разному.

Есть три основных варианта нажатия клавиш.

1) **Печатные клавиши [S 1 ,]**

   События keydown keypress – keyup

   Описание Нажатие вызывает keydown и keypress.

   Когда клавишу отпускают, срабатывает keyup.

2) **Специальные клавиши [Alt Esc ⇧]**

   События keydown – keyup

   Описание Нажатие вызывает keydown.

   Когда клавишу отпускают, срабатывает keyup.

3) **Сочетания с печатной клавишей [Alt+E Ctrl+У Cmd+1]**

   События keydown keypress – keyup

   Описание Браузеры под Windows — не генерируют keypress, браузеры под MacOS — генерируют. Если сочетание вызвало браузерный диалог («Сохранить файл», «Открыть» и т.п.), то keypress/keyup не будет.

Автоповтор

При долгом нажатии клавиши возникает автоповтор. По стандарту, должны генерироваться многократные события keydown (+keypress), и вдобавок стоять свойство свойство repeat=true у объекта события.

То есть, поток событий должен быть такой:

```
keydown
keypress
keydown
keypress
..повторяется, пока
клавиша не отжата...
keyup
```

Однако в реальности на это полагаться нельзя, под Firefox(Linux) генерируется и keyup:

```
keydown
keypress
keyup
keydown
keypress
keyup
..повторяется, пока
клавиша не отжата...
keyup
```

Задача 16

Измените интерфейс для создания многоуровнево списка.

1) Добавьте функцию сворачивания/разворачивания ветви списка по клику на элементе, который содержит эту ветвь
2) Добавьте функцию переноса ветвей и элементов с помощью метода Drag'n'Drop

input, textarea

Для большинства типов input значение доступно на чтение-запись в value:

```
input.value = "Новое значение";

textarea.value = "Новый текст";
```

Для элементов textarea также доступно свойство innerHTML, но лучше им не пользоваться: оно хранит только HTML, изначально присутствовавший в элементе. Кроме того, оно не преобразует HTML-сущности.

Например, сравните значения value/innerHTML до того, как что-то ввели (будут разные!) и после (innerHTML не поменяется):

```
<textarea> &lt;a&gt; </textarea>
<script>
  var area = document.body.children[0];
</script>
<button onclick="alert(area.value);
alert(area.innerHTML);">
  Вывести value и  innerHTML
</button>
```

input type="checkbox", input type="radio"

Для этих элементов можно узнать или установить текущее «отмеченное» состояние.

Оно находится в свойстве checked (true/false).

```
if (input.checked) {
  alert("Чекбокс выбран");
}
```

Обратите внимание на разницу значения атрибута и свойства. Атрибут имеет значение «как указано в HTML», а свойство — логическое, в соответствие со стандартом.

```
<input type="checkbox" checked>
<script>
var input = document.body.children[0];
alert(input.checked); // true
alert(input.getAttribute('checked')); // пустая строка
</script>
```

select, option

Элементы типа select, как и input, поддерживают свойство value.

Он обычно возвращает значение (value) выбранной опции, ну а в случае <select multiple> — значение первой из них.

var selectedOptionValue = select.value;

Элемент селекта в JavaScript можно выбрать двумя путями: поставив значение select.value, либо установив свойство select.selectedIndex:

select.selectedIndex = 0; // первый элемент

Установка selectedIndex = -1 очистит выбор.

Опции доступны через select.options.

Если select допускает множественный выбор (атрибут multiple), то значения можно получить/установить, сделав цикл по select.options:

```
<form name="form">
  <select name="genre" multiple>
    <option value="blues" selected>Мягкий
блюз</option>
    <option value="classic">Классика</option>
  </select>
</form>
```

```
<script>
var form = document.body.children[0];
var select = form.elements.genre;
for (var i=0; i<select.options.length; i++) {
  var option = select.options[i];
  if(option.selected) {  alert(option.value);  }
}
</script>
```

Событие change

Событие change происходит при изменении значения элемента формы. По стандарту, оно должно происходить после того, как элемент теряет фокус.

В реальности оно так и происходит для текстовых элементов.

Например, пока вы набираете что-то в текстовом поле ниже — события нет. Но как только вы уведёте фокус на другой элемент, например, нажмёте кнопку — произойдет событие onchange.

```
<input type="text" onchange="alert(this.value)">

<input type="button" value="Кнопка">
```

Для остальных же элементов: select, input type=checkbox/radio событие генерируется при выборе значения, еще до потери фокуса элементом.

Событие propertychange

Это событие происходит только в IE, при любом изменении свойства. Оно позволяет отлавливать изменение значения, как только это измнение произошло.

Если поставить его на checkbox в IE<9, то получится «правильное» событие change:

```
<input type="checkbox"> Чекбокс с "onchange", работающим везде
одинаково
<script>
var checkbox = document.body.children[0];

if("onpropertychange" in checkbox) {
  // если поддерживается (IE)
  checkbox.onpropertychange = function() {
    if (event.propertyName == "checked") { // имя свойства
      alert(checkbox.checked);
    }
  };
} else {
  // остальные браузеры
  checkbox.onchange = function() {
    alert(checkbox.checked);
  };
}
</script>
```

Событие input

Событие input срабатывает тут же при изменении значения текстового элемента и поддерживается всеми браузерами, кроме IE<9.

В IE9 оно поддерживается частично, а именно — не возникает при удалении символов.

Пример использования (не работает в IE<9):

```
<input type="text"> oninput: <span id="result"></span>
<script>
var input = document.body.children[0];

input.oninput = function() {
  document.getElementById('result').innerHTML = input.value;
}
</script>
```

События cut, copy, paste

Эти события используются редко, но иногда бывают полезны. Они происходят при вырезании/вставке/копировании значения в поле.

При этом действие браузера можно отменить, как это сделано в примере ниже:

```
<input type="text"> event: <span id="result"></span>

<script>
var input = document.body.children[0];

input.oncut = input.oncopy = input.onpaste = function(e) {
  e = e || event;
  document.getElementById('result').innerHTML = e.type +' '+input.value;
  return false;
}
</script>
```

Событие submit

Чтобы отправить форму на сервер, у посетителя есть два способа:

1) Первый — это нажать кнопку <input type="submit"> или <input type="image">.

2) Второй — нажать Enter на поле, когда оно в фокусе.

В обоих случаях будет сгенерировано событие submit. Если отменить в нём действие браузера, то форма не отправится на сервер.

Как правило, это используется для проверке данных в форме. Если что-то не так — выводится ошибка и отправка отменяется.

Посмотрим это на живом примере. Оба способа выдадут сообщение, форма не будет отправлена:

```
<form onsubmit="alert('submit!');return false">
  Первый: Enter в текстовом поле <input type="text" value="Текст"><br>
  Второй: Нажать на "Отправить": <input type="submit" value="Отправить">
</form>
</button>
```

Ожидаемое поведение:

3) Перейдите в текстовое поле и нажмите Enter, будет событие, но форма не отправится на сервер благодаря return false в обработчике.

4) То же самое произойдет при клике на <input type="submit">.

Метод submit

Чтобы отправить форму на сервер из JavaScript — нужно вызвать на элементе формы метод form.submit().

При этом само событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он выполнил все проверки.

Это используют, в частности, для искусственной генерации и отправки формы.

Событие focus blur

Событие focus вызывается тогда, когда пользователь фокусируется на элементе, например кликает на INPUT.

Событие blur вызывается, когда фокус исчезает, например посетитель кликает на другом месте экрана.

По умолчанию, не все элементы поддерживают эти события. Перечень элементов немного разнится от браузера к браузеру, например, список для IE описан в MSDN. Но есть элементы, которые поддерживают фокусировку всегда. В первую очередь это те, с которыми посетитель может взаимодействовать: INPUT, SELECT, A и т.п.

С другой стороны, элементы для форматирования, такие как DIV, SPAN, TABLE и другие — по умолчанию не поддерживают это событие, т.е. на них нельзя «фокусироваться». Впрочем, существует способ включить фокусировку и для них.

```
<input type="text">
<style>
.highlight { background: #FA6; }
</style>
<script>
var input = document.getElementsByTagName('input')[0];
input.onfocus = function() {   this.className = 'highlight'; }
input.onblur = function() {   this.className = ''; }
</script>
```

Метод elem.focus()

На элементе, который поддерживает фокусировку, можно сфокусироваться программно.

Например, можно показать форму и тут же сфокусироваться на ее поле, чтобы посетитель мог вводить текст:

```
<form style="display:none" name="form">
  Искать <input style="text" name="search">
</form>
<input type="button" value="Показать форму поиска" onclick="showForm()">
<script>
function showForm() {
  var form = document.forms.form;
  form.style.display = 'block';
  form.elements.search.focus();
}
</script>
```

Событие blur и метод elem.blur()

Событие blur — это противоположность focus. Оно срабатывает, когда элемент теряет фокус. Зачастую это событие используется для проверки («валидации») значения. Метод elem.blur() заставляет элемент потерять фокус, если таковой имеется.

```
<label>Введите ваш возраст: <input type="text" name="age"></label>
<div id="error">здесь будет ошибка, если вы введёте не число</div>
<style> .error { background: red; } </style>
<script>
var input = document.getElementsByName('age')[0];
var errorHolder = document.getElementById('error');
input.onblur = function() {
  if (isNaN(this.value)) { // введено не число
    // показать ошибку
    this.className = 'error';
    errorHolder.innerHTML = 'Вы ввели не число. Исправьте, пожалуйста'
  }
}
input.onfocus = function() {
  // сбросить состояние "ошибка", если оно есть
  if (this.className == 'error') {
    this.className = '';
    errorHolder.innerHTML = '';
  }
}
</script>
```

События "onload", "onbeforeunload" и "onerror"

Браузер позволяет отслеживать загрузку внешних ресурсов — скриптов, ифреймов, картинок и других.

Для этого есть два события:

1) onload — если загрузка успешна.
2) onerror — если при загрузке произошла ошибка.

Кроме того, обработчик window.onerror вызывается при ошибках JavaScript, которые не были пойманы try..catch.

onload

Рассмотрим задачу. В браузере работает сложный интерфейс и, чтобы создать очередной компонент, нужно загрузить его скрипт с сервера. Но, после подгрузки нужно запустить функцию создания компонента. Как определить, что скрипт загрузился?

Событие onload - сработает, когда скрипт загрузился и выполнился.

```
var script = document.createElement('script');
script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js"
document.documentElement.appendChild(script);
script.onload = function() {
  alert(jQuery);
}
```

onerror

Что, если загрузка скрипта не удалась? Например, такого скрипта на сервере нет (ошибка 404).

Любые ошибки загрузки (но не выполнения) скрипта отслеживаются обработчиком onerror:

```
var script = document.createElement('script');
script.src = "http://ajax.googleapis.com/404.js"
document.documentElement.appendChild(script);

script.onerror = function() {
  alert("Ошибка: " + this.src);
}
```

IE<9: onreadystatechange

В IE для отслеживания загрузки есть другое событие: onreadystatechange. Оно срабатывает многократно, при каждом обновлении состояния загрузки.

Текущая стадия процесса находится в script.readyState:

1) **loading** В процессе загрузки.
2) **loaded** Получен ответ с сервера — скрипт или ошибка. Скрипт на фазе loaded может быть ещё не выполнен.
3) **complete** Скрипт выполнен.

```
var script = document.createElement('script');
script.src = "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js";
document.documentElement.appendChild(script);

script.onreadystatechange = function() {
  alert(this.readyState); // loading -> loaded -> complete — без ошибки
                          // loading -> loaded — с ошибкой

}
```

Стадии могут пропускаться. Если скрипт в кэше браузера — он сразу даст complete.

Нет особой стадии для ошибки. В примере выше это видно, обработка останавливается на loaded.

window.onerror

Обработчик window.onerror вызывается в случае ошибки, выпавшей из всех блоков try..catch:

Аргументы обработчика:

1) message - Сообщение об ошибке.
2) source - Файл
3) lineno - Номер строки с ошибкой

```
<script>
window.onerror = function(message, source, lineno) {
  alert("Ошибка:"+message +"\n" +
    "файл:" + source + "\n" +
    "строка:" + lineno);
};
</script>
<body>
  <button onclick="alert(blablabla);">alert(blablabla)</button>
</body>
```

window.onload

Обработчик window.onload срабатывает, когда загружается вся страница, включая ресурсы на ней — стили, картинки, ифреймы и т.п.

Пример ниже выведет alert лишь после полной загрузки окна, включая IFRAME, стиль, картинки:

```
<iframe src="http://example.com/" style="height:60px"></iframe>
<link type="text/css" rel="stylesheet" href="/some.css">
<img src="/some.png">
<script>
  window.onload = function() {
    alert('Документ и все ресурсы загружены')
  }
</script>
```

window.onunload window.onbeforeunload

Когда человек уходит со страницы или закрывает окно, срабатывает window.unload. В нём можно, например, закрыть вспомогательные popup-окна, но отменить сам переход нельзя.

Если посетитель инициировал переход на другую страницу или нажал «закрыть окно», то обработчик onbeforeunload может приостановить процесс и спросить подтверждение.

Для этого ему нужно вернуть строку, которую браузеры покажут посетителю, спрашивая — нужно ли переходить.

```
window.onbeforeunload = function() {
  return "Данные не сохранены. Точно
перейти?";
};
```

Особенности:

1) Событие не поддерживается Opera
2) Firefox игнорирует текст, а показывает своё сообщение. Это сделано в целях безопасности.

Обработчики на IMG, IFRAME, LINK

Поддержка событий для других типов ресурсов:

1) IMG Подерживают onload/onerror во всех браузерах.

```
var img = document.createElement('img');
img.onload = function() { alert("OK "+this.src };
img.onerror = function() { alert("FAIL "+this.src }
img.src = ...
```

2) IFRAME Поддерживает onload во всех браузерах. Это событие срабатывает как при успешной загрузке, так и при ошибке.

```
<iframe src="http://example.com/" style="height:60px"></iframe>
<script>
  document.getElementsByTagName('iframe')[0].onload = function() {
    alert('IFRAME загружен')
  }
</script>
```

3) LINK (стили) Поддерживает onload/onerror в некоторых браузерах.

Событие загрузки документа "onDOMContentLoaded"

Событие onDOMContentLoaded срабатывает при загрузке документа, после выполнения всех тегов SCRIPT.

В отличие от window.onload, оно не ждёт загрузки дополнительных ресурсов, поэтому наступает гораздо раньше.

На момент наступления onDOMContentLoaded, дерево DOM полностью построено и все элементы доступны. Поэтому это событие часто используют для инициализации интерфейса.

```
<script>
  function yelp() {
    alert('DOM готов (а iframe - нет)');
  }

  document.addEventListener( "DOMContentLoaded", yelp, false );
</script>

<iframe src="http://example.com" height="80"></iframe>
```

Если в документе есть теги SCRIPT, то браузер обязан их выполнить до того, как построит DOM. Поэтому событие DOMContentLoaded ждёт загрузки и выполнения таких скриптов.

DOMContentLoaded и стили

Внешние стили никак не влияют на событие DOMContentLoaded. Но есть один нюанс. Если после стиля идёт скрипт, то этот скрипт обязан дождаться, пока стиль загрузится:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // сработает после загрузки style.css
</script>
```

Такое поведение прописано в стандарте. Его причина — скрипт может захотеть получить информацию со страницы, зависящую от стилей, например, ширину элемента.

В результате событие DOMContentLoaded будет также ждать стилей.

Задача 17

Измените интерфейс для создания многоуровнево списка.

1) Добавить на страницу форму, позволяющую добавлять элементы списка. Форма должна содержать поле для ввода номера родительского элемента, поле для ввода содержания для нового элемента, кнопку(при нажатии которой происходит добавление элемента) и подписи (элемент label) к каждому элементу формы

2) Значения, введенные в форму, должны проходить проверку на валидность

jQuery

Библиотека jQuery облегчает манипулирование объектной моделью документа DOM, добавление эффектов и выполнение Ajax-запросов.

1) **Кросс-браузерная совместимость.** Большая часть вашего программного кода будет одинаково функционировать во всех основных браузерах, включая Internet Explorer 6.

2) **Полезные утилиты.** Доступен ассортимент полезных функций: усечение строк, способность с легкостью расширять объекты и т. д.

3) **Пользовательский интерфейс jQuery UI.** (Виджеты accordion, slider, dialog, datepicker)

4) **Подключаемые модули.** Всякий раз, когда вы сталкиваетесь с какой-либо задачей или проблемой, прежде всего стоит проверить, имеется ли подключаемый модуль, который может помочь решить ее.

Подключение jQuery

Следует сообщить HTML -файлу о необходимости включения библиотеки jQuery. Это делается посредством добавления элемента <script> в секцию head веб-документа. Элемент <head> HTML -файла с включенной библиотекой jQuery будет выглядеть следующим образом:

```
<head>
<title>Hello jQuery world!</title>
<script type='text/javascript' src='jquery-1.7-min.js'></script>
<script type='text/javascript' src='script.js'></script>
</head>
```

Первый тег script на странице загружает библиотеку jQuery, а второй указывает на файл script.js, используя который мы будем запускать выполнение собственного кода jQuery.

Или добавить включения прямо перед закрывающим тегом </body>

```
</section>
<script type="text/javascript" src="jquery-1.7-min.js"></script>
<script type="text/javascript" src="script.js"></script>
</body>
</html>
```

Псевдоним jQuery

Благодаря включению библиотеки jQuery в веб-страницу мы получаем доступ к функции под названием jQuery. Посредством этой функции jQuery реализует сотни мощных инструментов. jQuery предусматривает более короткий псевдоним, посредством которого можно получить доступ к библиотеке. Псевдоним выглядит как знак $.

Все методы библиотеки jQuery располагается в собственном пространстве имен jQuery. Это позволяет избежать конфликтов при именовании функций.

Основной повод для того, чтобы применять полное имя вызова jQuery вместо короткого псевдонима $, возникает, когда на одной странице вы задействуете сразу несколько библиотек JavaScript, которые «борются» между собой за контроль над именем функции в виде знака доллара. Этот знак представляет собой распространенное имя функции, встречающееся в некоторых библиотеках, часто применяемых для выборки элементов.

Анализ оператора jQuery

Каждая команда состоит из четырех частей: функции jQuery или ее псевдонима, селекторов, действий (или методов) и параметров.

1) Сначала мы задействуем селектор для выборки или перемещения элементов на веб-странице.

2) Затем указываем действие, которое будет применено к каждому из выбранных элементов.

3) И наконец, вводим параметры, посредством которых сообщаем jQuery, как именно хотим применить указанное ранее действие.

Селектор	Действие	Параметры
jQuery('p')	.css	('color', 'blue')
$('p')	.css	('color', 'blue')

В приведенном примере мы выбрали с помощью селектора все теги абзаца (HTML-теги <p>). Далее мы указали jQuery-действие css, применяемое для модификации CSS-свойства элементов абзаца, которые были выбраны изначально. В завершение мы передаем параметры для настройки значения CSS-свойства color в виде blue. (Все наши абзацы окрасятся в синий цвет).

Количество передаваемых параметров может быть разным. Некоторые действия не требуют никаких параметров, другие могут принимать различные наборы параметров, а третьи предполагают указание другой функции JavaScript для выполняющегося программного кода при наступлении определенного события

document-ready

Почти все, что вы будете делать с применением jQuery, необходимо осуществлять после того, как документ полностью загрузится.

```
$(document).ready(function() {
  alert('Welcome to StarTrackr! Now no longer under police …');
  investigation!);
});
```

При этом jQuery включает сокращенную версию, которая устраняет "(document)" и "ready":

```
$(function() {
alert('Ready to do your bidding!');
});
```

При беглом взгляде может показаться, что в событии document-ready отсутствует структура, с которой мы сталкивались при изучении анатомии операторов jQuery, однако если присмотреться повнимательнее, то можно увидеть, что все необходимые части задействованы: селектором здесь выступает document, действием — ready, а роль параметра играет функция, которая запускает выполнение определенного программного кода (в нашем случае это alert).

Простая выборка

Если нам потребуется произвести выборку любого абзаца, элемента , заголовка <h1> или поля <input>, мы воспользуемся следующими селекторами:

```
$('p')          $('div')          $('h1')          $('input')
```

Для ссылки на имена идентификатора (id) и класса (class) jQuery заимствует некоторые условности из CSS. Чтобы произвести выборку на основе идентификатора id, используйте хеш-символ (#), указывая после него идентификатор элемента, и передавайте все это как одну строку функции jQuery. Аналогично мы можем использовать CSS-селектор class для осуществления выборки на основе атрибута класса class. Для этого необходимо передать функции jQuery строку, состоящую из знака точки (.), после которого указывается имя класса элемента:

```
<table class="data" id="celebs"></table>
<script>
$('#celebs');
$('.data');
</script>
```

Допустим, нам необходимо произвести выборку всех элементов , располагающихся внутри элементов <p>, которые, в свою очередь, содержатся внутри элементов .

```
$('div.fancy p span')
```

Придерживаясь данного шаблона, вы сможете производить выборку почти любых элементов, которые вам потребуются

Фильтры

Фильтр позволяет отсечь определенные элементы и выбирать только те из них, которые требуются в данный момент.

```
$(document).ready(function() {
alert($('#celebs tbody tr:even').length + ' elements!');
});
```

Фильтр прикрепляется к элементу, который будет подвергаться фильтрации (в нашем случае это строки таблицы), и обозначается знаком двоеточия, после которого указывается имя этого фильтра. Фильтр :even позволяет сохранить все четные элементы в выборке, отсекая остальные.

Существует довольно большой массив фильтров селекторов jQuery, которыми можно воспользоваться: :odd, :first, :last, :eq() и др.

Использование множественных селекторов

Последний хитрый прием, связанный с выборкой, заключается в возможности выполнения множественных запросов посредством одного оператора. Это очень удобно, поскольку нам часто будет требоваться применять одинаковое действие к разным элементам в несвязанных частях веб-страницы. Для этого необходимо разделить строки селектора запятыми.

Так, например, если нам нужно произвести выборку каждого абзаца, элемента , заголовка <h1> и поля <input> на веб-странице, то потребуется следующий селектор:

```
$('p,div,h1,input')
```

Просмотр CSS-свойств

Перед тем как начать вносить изменения в CSS -свойства, посмотрим, как к ним можно получить доступ. jQuery позволяет нам сделать это с помощью функции css, например:

```
/*Данный код сгенерирует предупреждение о размере
шрифта первого элемента, отобранного селектором как
соответствующего критериям выборки*/

$(document).ready(function() {
  var fontSize = $('#celebs tbody tr:first').css('font-size');
  alert(fontSize);
});
```

Примечательный аспект извлечения CSS - свойств подобным способом заключается в том, что jQuery позволяет получить вычисляемый стиль элемента. Это означает, что вы получите значение, которое применяется при отображении элемента в браузере пользователя, а не то, что указано в CSS-определении.

Таким образом, если, например, вы задали высоту для элемента div 200 пикселов в CSS-файле, однако его содержимое обусловило то, что данная величина было превышена, jQuery сообщит вам действительную высоту элемента, а не те 200 пикселов, которые вы указывали.

Настройка CSS-свойств

Добавить CSS-свойство можно разными способами. Наиболее простой из них заключается в повторном использовании оператора jQuery целиком, но с новыми значениями.

```
$('#celebs tbody tr:even').css('background-color','#dddddd');
$('#celebs tbody tr:even').css('color', '#666666');
```

Выполнение этих строк происходит поочередно.

jQuery позволяет настраивать множественные свойства за один подход с помощью литерала объекта.

```
$('#celebs tbody tr:even').css({'background-color': '#dddddd', 'color': '#666666'});
```

Литерал объекта заключен в фигурные скобки, при этом каждый ключ отделен от соответствующего ему значения при помощи знака двоеточия, а каждая пара «ключ/значение» отделяется от прочих запятыми. Он передается в качестве одиночного параметра функции css. Используя подобную методику, вы можете указывать любое количество пар «ключ/значение», нужно лишь разделять их запятыми.

Добавление и удаление классов

Функция addClass может принимать в качестве параметра строку, содержащую имя класса. Вы также можете добавлять за один подход сразу несколько классов, разделяя их имена пробелами точно так же, как при написании HTML-кода:

```
$('div').addClass('class_name');
$('div').addClass('class_name1 class_name2 class_name3');
```

Возможны также ситуации, когда нам потребуется удалить имена классов из элементов. Действие по удалению класса называется соответственно — removeClass. Данная функция используется точно так же, как и addClass, следует лишь передать в качестве параметра (не)нужное имя класса:

```
$('#celebs tr.zebra').removeClass('zebra');
```

Обработчики событий

Обработчики событий называются так, поскольку они выполняют функцию обработки событий. События представляют собой действия (предпринимаемые либо пользователем, либо самим браузером), которые имеют место на веб-странице. Когда событие происходит, мы говорим, что оно инициируется. А когда мы пишем программный код для обработки события (который будет обернут в обычную JavaScript-функцию), то говорим, что событие будет перехватываться.

Мы использовали обработчик событий click для того, чтобы дать указание jQuery скрыть элемент disclaimer, когда пользователь щелкает на кнопке:

```
<input type="button" id="hideButton" value="hide" />
...
$('#hideButton').click(function() {
$('#disclaimer').hide();
});
```

Когда событие происходит, зачастую требуется ссылка на элемент, который его инициировал. Например, мы можем захотеть некоторым образом модифицировать кнопку, на которой только что щелкнул пользовать. Подобная ссылка доступна в программном коде нашего обработчика событий, для чего необходимо воспользоваться ключевым JavaScript-словом this.

```
$('#hideButton').click(function() {    // $(this) обеспечивает более удобный способ
$(this).hide();          //   ссылки на элемент, который инициировал событие, чем его
});                      //повторная выборка.
```

Добавление новых элементов

Фактически любая допустимая HTML-строка, помещаемая внутрь функции jQuery, создается и подготавливается для размещения на веб-странице. Вот пример того, как можно создать простой элемент paragraph:

```
$('<p>A new paragraph!</p>')
```

При написании данного программного кода jQuery выполняет полезные действия: разбирает HTMLна фрагменты объектной модели документа DOMи производит их выборку — подобно тому как это делает обычный селектор jQuery. Это означает, что он тотчас же будет готов для дальнейшей jQuery-обработки. Так, например, для того, чтобы добавить класс к нашему новому элементу, необходимо ввести следующее:

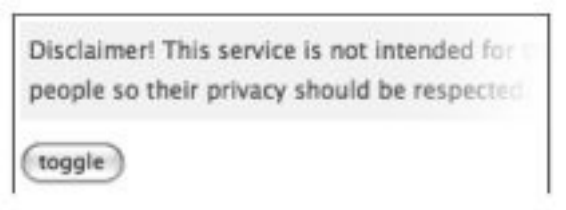
```
$('<p>A new paragraph!</p>').addClass('new');
```

Используя данный метод, вы сможете создавать любые новые элементы посредством jQuery вместо того, чтобы определять их в HTML-разметке.

После создания новых элементов нам потребуется средство для того, чтобы вставить их в нужное место веб-страницы. Для этого предназначены несколько функций jQuery.

InsertAfter, insertBefore

InsertAfter примет выбранные нами текущие элементы (в данном случае это новые элементы, которые мы создали) и вставит их после другого выбранного элемента, который мы передадим ей в качестве параметра.




```
$('<input type="button" value="toggle" id="toggleButton">').insertAfter('#disclaimer');
$('#toggleButton').click(function() {
    $('#disclaimer').toggle();
});
```

Функция insertAfter добавляет новый элемент в виде элемента-брата непосредственно после элемента предупреждающего сообщения disclaimer. Если вы хотите, чтобы кнопка располагалась перед элементом disclaimer, то можете разместить данный элемент перед элементом disclaimer, воспользовавшись функцией insertAfter, либо применить метод insertBefore, что более логично. insertBefore также добавляет новый элемент в виде элемента-брата существующего элемента, однако он будет размещаться непосредственно перед ним.

```
$('<input type="button" value="toggle" id="toggleButton">')
.insertBefore('#disclaimer');
```

prependTo и appendTo

Если вы желаете добавить новый элемент в виде дочернего элемента какого-либо из существующих элементов (то есть чтобы новый элемент располагался внутри существующего элемента), воспользуйтесь функциями prependTo и appendTo.

START! Disclaimer! This service is not intended for the those with criminal intent. Celebrities are kind of like people so their privacy should be respected. **END!**

```
$('<strong>START!</strong>').prependTo('#disclaimer');
$('<strong>END!</strong>').appendTo('#disclaimer');
```

Новые элементы были добавлены в начало (START!) и в конец (END!) текущего div-элемента предупреждающего сообщения disclaimer, а не перед или после него.

remove

Для удаления элементов с использованием jQuery сначала, как обычно, необходимо произвести их выборку с помощью селектора, после чего вызвать метод remove:

```
<p id="no-script">
  We recommend that you have JavaScript enabled!
</p>
$('#no-script').remove();
```

Действие remove удаляет все выбранные элементы из объектной модели документа DOM, а также все обработчики событий и данные, связанные с этими элементами. Действие remove не требует ввода каких-либо параметров, однако вы можете указать выражение, чтобы конкретизировать рамки выборки.

```
$('#celebs tr').remove(':contains("Singer")');
```

Вместо того чтобы удалять все tr в -элементе #celebs, данная строка кода позволит нам удалить только те строки, в которых содержится слово Singer.

text и html.

Действия text и html схожи между собой — оба позволяют настраивать содержимое выбранных элементов. Для этого нужно просто передать строку любой из этих функций.

```
$('p').html('<strong>Warning!</strong> Text has been replaced ... ');
$('h2').text('<strong>Warning!</strong> Title elements can be ...');
```

В обоих приведенных примерах содержание соответствующих элементов будет заменено строкой, которую мы ввели: каждый тег paragraph и h2 на странице будет перезаписан новым содержанием. Абзацы будут содержать текст, выделенный полужирным шрифтом, однако теги <h2> будут включать всю строку точно такой, как было указано, в том числе и теги .

text — для простого текста, а html — для HTML.

fadeIn и fadeOut

Одним из наиболее распространенных (и постоянных) эффектов jQuery является встроенный эффект изменения прозрачности. Чтобы использовать данный эффект в его простейшей форме, достаточно заменить show на fadeIn или hide на fadeOut.

```
$('#hideButton').click(function() {
   $('#disclaimer').fadeOut();
});
```

Существуют также параметры, позволяющие модифицировать данный эффект.

Первый из них предназначен для контроля времени, затрачиваемого на изменение прозрачности. Многие эффекты и анимации jQuery позволяют задавать параметр времени, который можно передавать в виде строки или целого числа. Временной промежуток можно указывать в виде строки, используя одно из следующих предопределенных слов: slow, fast или normal, например: fadeIn('fast').

Если вы хотите получить более детальный контроль над продолжительностью анимации, то можете указать время в миллисекундах: fadeIn(1000).

Переключение эффектов и анимаций

jQuery также включает действие fadeToggle, которое позволит отобразить элемент, если он скрыт, и скрыть его, если он отображается. Кроме того метод toggle, если мы передадим ему параметр со значением промежутка времени, то увидим, что он поддерживает анимацию.

```
$('#toggleButton').click(function() {
$('#disclaimer').toggle('slow');
});
```

Здесь видно, что ширина, высота и непрозрачность всего элемента целиком анимируются. Еще один анимационный эффект из основной библиотеки jQuery, который также включает встроенные действия для переключения, — выдвигание. Он реализуется таким же образом, что и эффект изменения прозрачности, но с применением действий slideDown, slideUp и slideToggle.

```
$('#toggleButton').click(function() {
$('#disclaimer').slideToggle('slow');
});
```

Данный эффект позволяет показывать элементы, как если бы они выдвигались из скрытого отделения, и убирать их. Как и при изменении прозрачности, здесь можно указывать временной промежуток.

Функции callback

Многие эффекты (включая выдвигание и изменение прозрачности) поддерживают специальный параметр, называемый функцией callback. Такие функции представляют собой код, который выполняется после того, как эффект полностью завершится.

```
$('#disclaimer').slideUp('slow',
    function() {
      $('#hideButton').fadeOut();
    }
);
```

В нашем случае функция callback выполняется после того, как элемент закончит выдвигаться.Предупреждающее сообщение уйдет вверх, и сразу после того, как данная анимация завершится, кнопка пропадет из поля зрения.

animate

Animate позволяет анимировать набор CSS-свойств с целью создания ваших собственных эффектов.

```
$('p').animate({
padding: '20px',
fontSize: '30px'
}, 2000);
```

Данный код позволяет анимировать все абзацы на веб-странице, изменяя отступ от его первоначального значения до 20 пикселов и увеличивая размер шрифта до 30 пикселов с периодом 2 с (2000 мс).

При использовании функции animate мы передаем литералы объектов, содержащие свойства, которые мы собираемся анимировать и которые определяются в виде пар «ключ/значение».

Имена свойств должны быть написаны слитно, без разделяющих дефисов, чтобы их могла использовать функция animate. То есть нужно писать backgroundColor вместо background-color и marginLeft вместо margin-left. Значения CSS-свойств можно указывать в пикселах, относительных единицах (em), процентах или пунктах. Например, вы можете указать 100px, 10em, 50% или 16pt. Определяемые вами значения могут соотноситься с текущим значением элемента: все, что вам потребуется сделать, это указать += или -= впереди значения, после чего оно будет добавлено или вычтено из текущего значения элемента.

```
$(this).animate({paddingLeft: '+=15px'}, 200);
```

Плагины

Popular Tags

- ui (202)
- animation (85)
- form (82)
- input (72)
- jquery (64)
- image (49)
- effect (48)
- ajax (43)
- html5 (41)
- scroll (35)

Подключаемые модули jQuery представляют собой превосходный способ создания кода для многоразового использования.

1) **jQuery UI** - библиотека пользовательского интерфейса, представляет собой коллекцию виджетов, эффектов и взаимодействий
2) **Shadowbox1**, **FancyBox2** и **Lightbox2** - применяется для отображения полноценной по размеру версии какого-либо миниатюрного изображения в модальном диалоговом окне(лайтбокс-модули)
3) **Validation** - позволяет с минимальными усилиями внедрять комплексную и конфигурируемую систему проведения проверок в большинство форм.