МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.М. ГОСТИН, А.Н. САПРЫКИН

ИНТЕРНЕТ-ТЕХНОЛОГИИ

Часть 2

Учебное пособие

Рязань 2017

УДК 004.43

Интернет-технологии. Часть 2: учеб. пособие / А.М. Гостин, А.Н. Сапрыкин; Рязан. гос. радиотехн. ун-т. Рязань, 2017. 64 с.

Приведены основные сведения о языке JavaScript. Рассмотрены основные принципы создания интерактивных Web-страниц с помощью JavaScript.

Предназначено для бакалавров и магистров очной, очно-заочной и заочной форм обучения направления 09.00.00 – "Информатика и вычислительная техника".

Ил. 6. Библиогр.: 4 назв.

Язык JavaScript, веб-страница

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра САПР вычислительных средств Рязанского государственного радиотехнического университета (зав. кафедрой В.П. Корячко)

Г о с т и н Алексей Михайлович
С а п р ы к и н Алексей Николаевич

Интернет-технологии

Редактор Р.К. Мангутова
Корректор С.В. Макушина
Подписано в печать 30.09.17. Формат бумаги 60х84 1/16.
Бумага писчая. Печать трафаретная. Усл. печ. л. 4,0.
Тираж 50 экз. Заказ .
Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.

1. Событийное программирование JavaScript

1.1. Синтаксис языка JavaScript

Язык JavaScript является объектно-ориентированным интерпретатором, встроенным в браузер, и служит для обработки программных сценариев страницы. Сценарии (или скрипты) обеспечивают динамическую обработку событий и взаимодействие пользователя с интерфейсом страницы. Спецификация языка описывается стандартом ECMAScript и постоянно развивается консорциумом W3C.

В отличие от языка Java, язык JavaScript не компилирует скрипты, а выполняет их непосредственно, что накладывает ряд особенностей, например возможно использование переменных в скриптах без их предварительного объявления и т.д. Также для обеспечения безопасности клиентских приложений язык JavaScript не может совершать прямые файловые операции ввода-вывода, не имеет прямого доступа к функциям операционной системы. Возможность работы с файлами у скриптов ограничена пределом "песочницы" браузера (отведенного каталога). С помощью технологии JavaScript можно выполнять запросы к серверу (в том числе без перезагрузки страницы, используя технологию AJAX), иметь программный доступ ко всем элементам страницы, вкючая CSS стили (с помощью объектной модели документа - DOM), управлять поведением браузера (с помощью объектной модели браузера - BOM) и т. д.

На стороне клиента сценарии JavaScript обеспечивают эффективное взаимодействие пользователя с интерфейсом страницы и страницы - с сервером. Программы на JavaScript могут выполнять первичную фильтрацию передаваемых на сервер данных, принимать от сервера данные в форматах XML и JSON, осуществлять предобработку форм, генерацию элементов пользовательского интерфейса и другие операции.

В основе синтаксиса языка JavaScript лежит интерпретируемый код, заключенный в функции-обработчики. Код скриптов обычно загружается вместе с загрузкой страницы и размещается в специальном контейнерном теге:

```
 <script>
alert('Hello, world!');
</script>
```

Контейнер `<script>` чаще всего размещается в заголовке страницы `<head>` и сразу же интерпретируется. Иногда скрипты

размещают в других местах страницы, например в подвале, чтобы их интерпретация и выполнение не задерживали загрузку страницы. В настоящее время для асинхронной загрузки скриптов используется атрибут без содержания `async`.

Часто скрипты размещают в отдельных текстовых файлах с расширением `.js`, которые подключают к контейнеру `<script>` с помощью атрибута `src`:

```
<script src="js/form.js"></script>
```

В данном примере скрипт содержится в файле `form.js`, размещенном в локальном каталоге `js` относительно текущего корня веб-сервера. В качестве значения также может быть указан абсолютный URL адрес скрипта:

```
<script async src="http://www.google.com/analytics.
js"> </script>
```

Синтаксис сценариев на языке JavaScript включает набор команд, разделенных символом ; (точка с запятой):

```
alert('Hello'); alert('world!!!');
// два текстовых сообщения
```

Переменные

Для объявления переменных в JavaScript используется ключевое слово `var`:

```
var <идентификатор> [=<значение>];
```

Идентификатор представляет собой имя переменной, состоящее из латинских букв, цифр, символов: *$* (денежный знак) и _ (подчеркивание), начинающееся не с цифры. В качестве идентификаторов нельзя использовать зарезервированные команды языка. При объявлении нескольких переменных идентификаторы разделяются запятыми. Пример:

```
var author = 'Ivan', age = 25, message_12 =
'Hello!';
```

При описании переменных регистр букв имеет значение. Традиционно в JavaScript заглавными буквами объявляются константы:

```
var GREEN = '#00FF00';
```

Если при инициализации переменной ключевое слово `var` не указано, то такая переменная является *глобальной*.

В языке JavaScript определено шесть типов данных.

1. *Число* (number) представляет собой целое или действительное число в обычной или экспоненциальной форме записи. Кроме того, существует два специальных значения - `NaN` (не число) и `Infinity` (бесконечность). На представление числа в JavaScript отводится 32 бита (4 байта). Примеры:

```
23; 3.14; -8; 6e-3; 0.006; NaN.
```

Кроме десятичных чисел, в программах на JavaScript можно использовать шестнадцатеричные, восьмеричные и двоичные числа. Для этого их предваряют префиксами: `0x` (шестнадцатеричное), `0` (восьмеричное), `0b` (двоичное). Примеры чисел: `0xffff; 0777; 0b1111.`

2. *Строка* (string) представляет собой любую последовательность символов, заключенных в кавычки или апострофы. Символьного типа данных в языке нет. Строка может содержать специальные символы типа "*\n*" (перевод строки) или быть пустой "". В JavaScript используется внутреннее представление строк в формате Unicode. Примеры строк: `"программа", 'Hello', 'A'`.

3. *Булевый тип* (boolean) представляет логический тип данных. Переменные этого типа принимают одно из двух значений: `true` (истина) или `false` (ложь).

4. *Пустой тип* (null) – отдельный тип данных, означает: значение неизвестно. Пример: `age = null;`

5. *Неопределенный тип* (undefined) – отдельный тип данных, означает: значение не присвоено. Пример: `var age;`

6. *Объект* (object) – представляет собой объект Javascript. Объект объявляется с помощью фигурных скобок и может состоять из полей и методов. Пример:

```
var user = { name: "Василий" };
```

Комментарии

Комментарии в языке JavaScript указываются в стиле C++.

Однострочный комментарий указывается с помощью символов `//` (двойной слеш) и действует до конца строки. Пример:

```
var age; // undefined – значение не присвоено
```

Многострочный комментарий указывается внутри последовательности символов `/*` *и* `*/` и действует на все многострочное содержимое:

```
/*  ...комментарий... */
```

Операторы

Язык JavaScript имеет C-подобные операторы и команды.

Оператор присвоения служит для присвоения значений переменным:

```
c = a + b; // присвоение суммы a и b
```

Возможно последовательное применение нескольких операторов присвоения. Оператор присвоения правоассоциативен (выполняется в последовательности справа налево). Пример:

```
c = d = x; // сначала d = x, потом c = d
```

Арифметические операторы: `+ (сложение)`, `– (вычитание)`, `* (умножение)`, `/ (деление)`, `% (остаток от деления)` подобны используемым в других языках. Все арифметические операторы являются бинарными и левоассоциативными. Оператор `+ (сложение)` в зависимости от типов аргументов может использоваться как для сложения чисел, так и для объединения (конкатенации) строк.

Операторы сравнения: `< (меньше)`, `> (больше)`, `== (равно)`, `!= (не равно)`, `<= (меньше или равно)`, `>= (больше или равно)`, `=== (строго равно)`, `!== (строго не равно)`. Последние два оператора отличаются от обычного равенства и неравенства тем, что при их выполнении над аргументами не производится преобразование типов. Пример:

```
c = (2 == '2'); // результат c = true
c = (2 === '2'); // результат c = false
```

Операторы сравнения имеют логический тип возвращаемого значения и часто используются в условных конструкциях языка.

Логические операторы: `&& (логическое И, конъюнкция)`, `|| (логическое ИЛИ, дизъюнкция)`, `! (логическое НЕ, отрицание)`. Логические операторы применяются к логическим выражениям и имеют логический тип возвращаемого значения: `true` или `false`. Пример:

```
c = !(2*2 == 4); // результат c = false
c = (5 < 2) || (3 > 2); // результат c = true
```

Унарные операторы представляют собой операторы с одним аргументом: `+ (унарный плюс)`, `– (унарный минус)`, `++ (инкремент)`, `-- (декремент)`, `! (логическое отрицание)`, `~`

(побитовое НЕ).

Операторы инкремент (увеличение на 1) и декремент (уменьшение на 1) могут быть инфиксные (ставятся перед аргументом) или постфиксные (ставятся после аргумента). Это влияет на порядок вычислений. Остальные унарные операторы являются инфиксными. Пример:

```
c = 2; a=c++; // a = 2, c = 3
c = 2; a=++c; // a = 3, c = 3
```

Побитовые операторы: & (побитовое И), | (побитовое ИЛИ), ^ (побитовое исключающее ИЛИ), ~ (побитовое НЕ), >> (сдвиг вправо), << (сдвиг влево), >>> (сдвиг вправо с заполнением 0).

Побитовые операторы, в отличие от логических, применяются к операндам побитно. Они имеют числовой тип возвращаемого значения. Операторы сдвига являются аналогами умножения (деления) на число 2 в степени, равной количеству сдвигаемых бит. Оператор сдвига вправо с заполнением 0 в отличие от обычного сдвига при выполнении добавляет нули слева. Пример:

```
c = 5 & 6; // c = 4 (0b101 & 0b110 = 0b100)
c = -9 >> 2; // c = -3 (0xfffffffd)
c = -9 >>> 2; // c = 1073741821 (0x3ffffffd)
```

Побитовые операции выполняются после операций сравнения.

Операторы с присвоением: += (сложение), -= (вычитание), *= (умножение), /= (деление) и т.д. являются сокращенной формой записи двух операторов: присвоения и действия. Пример:

```
c += 2; // то же, что c = c + 2
```

Приведение простых типов

В языке JavaScript может использоваться явное и неявное преобразование простых типов. Преобразования типов используются для строк, чисел и логических переменных.

Функция typeof(x) позволяет определить тип переменной. Результатом ее выполнения является строка, содержащая тип. Пример:

```
typeof(3); // "number"
```

В случае сложения строки с числом, если строка может быть преобразована в число, выполняются преобразование и сложение ее как числа, в противном случае выполняется операция объединения (конкатенация). Пример:

```
c = 3 + '2'; // результат c = 5
c = 'help' + 4; // результат c = 'help4'
```

Для явного приведения к строке также может использоваться конструктор `String(x)`, например: `String(null); // строка "null"`.

Оператор `+ (унарный плюс)` используется для приведения аргумента к числу:

```
c = +'3' + 2; // результат c = 5
```

Для преобразования строк в числа можно использовать специальные функции мягкого преобразования. Основная их особенность заключается в том, что строка в этом случае преобразуется посимвольно в число до тех пор, пока это возможно. Таким образом достигается положительный результат.

Функция `parseInt(<строка>[,<основание>])` преобразует строку к целому числу по заданному основанию (по умолчанию, к десятичному). Например:

```
c = parseInt("3px"); // результат c = 3
d = parseInt("ff",16); // результат d = 255
```

Функция `parseFloat(<строка>)` преобразует строку к действительному числу. Например:

```
c = parseFloat("3.14"); // результат c = 3.14
```

Для явного приведения к числу также можно использовать конструктор `Number(x)`. Например:

```
Number("3"); // число 3.
```

Для того чтобы узнать, является ли значение данного выражения числом, используется функция `isNaN(x)`, возвращающая логическое значение:

```
isNaN("25 руб."); // true (не число).
```

При выполнении приведения к числу значения `null` и `false` преобразуются в число `0`, значение `true` – в `1`, значение `undefined` – в `NaN`.

Оператор `!! (двойное отрицание)` используется для приведения аргумента к логическому значению, например: `alert(!!"0"); // true`

При использовании логического типа строка `""`, а также числа `0`, `NaN`, `undefined` и `null` интерпретируются как `false`, остальные

как `true` (в том числе и строка `"0"`).

Для явного приведения к логическому типу также можно использовать конструктор `Boolean(x)`, например: `Boolean("1"); // true`.

Приоритет операций

В JavaScript существует 19 приоритетов операций.

Основные из них (в порядке убывания):

1) группировка операторов, заключенных в скобки `()`;
2) операторы доступа к элементам: `.` (точка), `[]`;
3) вызов функции `()`, команда `new`;
4) унарные операторы: `+`, `–`, `++`, `––`, логическое и побитовое отрицание: `!`, `~`;
5) умножение, деление и остаток: `*`, `/` и `%`;
6) сложение и вычитание: `+` и `–`;
7) побитовые сдвиги: `>>`, `<<`, `>>>`;
8) операторы сравнения: `<`, `>`, `<=`, `>=`, `==`, `!=` и т.д.;
9) побитовое И `&`;
10) побитовое ИЛИ `|`;
11) логическое И `&&`;
12) логическое ИЛИ `||`;
13) присвоение: `=`, `+=`, `–=` и т.д.

Сначала выполняются операции с высшим приоритетом. Пример:

```
c = 2 + 4*6 / 3*2 | 1 << 2;  //  2 + (24/3*2) | 4 = (2 + 16) | 4 = 22
```

Ввод – вывод значений

Пользователь может взаимодействовать с JavaScript программой с помощью модальных окон. Модальные окна прерывают работу программы до тех пор, пока пользователь не введет данные или не нажмет соответствующую кнопку. В JavaScript используется три вида функций для вывода модальных окон.

1. Функция `alert` выводит информационное сообщение. Формат:

```
alert(<сообщение>);
```

Пример:

```
alert("Сообщение"); // вывод информационного сообщения
```

Результат выполнения примера показан на рис. 1.

Рис. 1. Пример работы функции `alert`

Текстовое окно используется для оповещения пользователя и для отладки. При нажатии кнопки `OK` или клавиши `<Esc>` окно исчезает.

2. Функция `prompt` выводит окно с сообщением и полем ввода. Формат:

```
<переменная> = prompt(<сообщение>,
<значение_по_умолч>);
```

Пример:

```
var year = prompt("Сколько вам лет?", 100);
```

Результат выполнения примера показан на рис. 2.

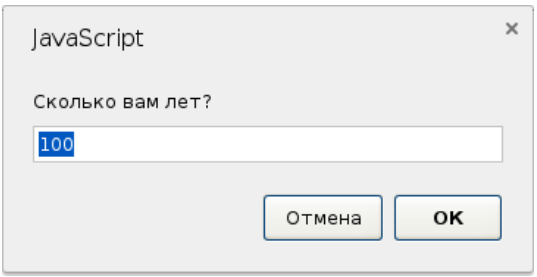

Рис. 2. Пример работы функции `prompt`

Введенное в поле ввода значение присваивается указанной переменной. Второй параметр задает значение поля по умолчанию. Если он не используется, рекомендуется задать пустую строку `""`. После нажатия клавиши `OK` функция возвращает введенное текстовое значение. При нажатии кнопки `<Отмена>` или клавиши `<Esc>` возвращается текстовое значение `"null"`.

3. Функция `confirm` выводит запрос подтверждения с двумя кнопками. Формат:

```
<переменная> = confirm(<запрос>);
```

Пример:

```
var admin = confirm("Вы – администратор?");
```

Результат выполнения примера показан на рис. 3.

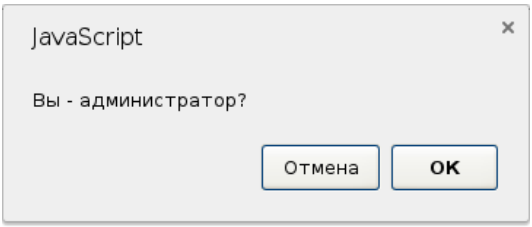

Рис. 3. Пример работы функции `confirm`

При нажатии на кнопку `OK` функция возвращает значение `true`, в ином случае – `false`.

Управляющие конструкции

Язык JavaScript имеет C-подобные управляющие конструкции.
Простой условный оператор (если-то) имеет вид:

```
if (<выражение>) <действие>;
```

Если выражение в круглых скобках истинно, то выполняется действие. Пример:

```
if (a < 0) alert("Число a – отрицательное!");
```

При необходимости выполнения нескольких действий по условию используются операторные скобки `{ }`. После операторных скобок точка с запятой не ставится. Пример:

```
if (a < 0) { a =-a; alert("Модуль числа " + a);  }
```

Другой тип условного оператора (если-то-иначе) имеет вид:

```
if (<выражение>) <действие1> else <действие2>;
```

Если выражение истинно, то выполняется `действие1`, иначе – выполняется `действие2`. Пример:

```
if (x < 0) alert('x – отрицательное'); else
alert('x – положительное');
```

Обратите внимание, что перед оператором `else` должна стоять точка с запятой. При необходимости в операторе можно использовать

операторные скобки, а также использовать вложенные и составные конструкции:

```
if (d > 0) {
   x1 = (-b+Math.sqrt(d))/(2*a);
   x2 = (-b-Math.sqrt(d))/(2*a);
   alert("корни x1="+x1+" x2="+x2);
} else if (d = 0) {
   x1 = -b/(2*a);
   alert("корень x1="+x1);
} else  alert("корней нет");
```

Функция извлечения квадратного корня относится к глобальному объекту Math.

Оператор *присвоения с условием* имеет вид:

```
<переменная> = (<условие>) ? <выражение1>:
<выражение 2>
f = (x > 0)? x: -x; // функция модуля f=|x|
```

Этот оператор является сокращенным вариантом конструкции:

```
if (x>0) f = x; else f = -x;
```

Оператор выбора применяется при проверке нескольких условий:

```
switch (<переменная>) {
    case <значение1>: <действие1>;
    ...
    case <значениеN>: <действиеN>;
    [default: <оператор по умолчанию>;]
}
```

Если переменная принимает одно из заданных значений, то выполняется соответствующее действие. Пример:

```
switch (a) {
    case 0: alert('ложь'); break;
    case 1: alert('истина'); break;
    default: alert('значение не установлено');
}
```

Команда `break` прекращает вычисления после вывода сообщения и производит выход из оператора `switch`. Если она не указана, то происходит переход к следующей строке оператора `switch`. Условие `default` срабатывает при выборе любого другого значения переменной a, кроме 0 и 1.

Оператор параметрического цикла имеет вид:

```
for ([<начало>]; [<условие>]; [<шаг>]) <тело
цикла>;
```

Параметр `<начало>` задает начальное условие, параметр `<условие>` – условие выполнения цикла, параметр `<шаг>` – изменение в каждом цикле. Пример:

```
for (i = 1, s = 1; i <= 5; i++)  s*=i; //  значение
5! = 1*2*3*4*5 = 120
```

В данном примере перед началом цикла выполняется два действия: `i=1` и `s=1`. Цикл выполняется за 5 итераций, в каждой из которых переменная `i` увеличивается на 1. Тело цикла может состоять из нескольких операций:

```
s = a = 0;
dx = 0.01;
for (i = 0; i < 100; i++) {  // интеграл x2dx от 0
до 1
     x = a + i * dx;
     s += x*x*dx;
} // значение s = 0.32835
```

Оператор цикла с предусловием имеет вид:

```
while (<условие>) <тело цикла>;
```

Тело цикла выполняется, если только выполняется заданное условие:

```
s = x = 0;
while (x <= 100) {    // сумма ряда 1/x, где x
изменяется от 1 до 100
    x++;
    s += 1 / x;
}  // значение s = 5.1873...
```

Оператор цикла с постусловием имеет вид:

```
do <тело цикла> while (<условие>);
```

Тело цикла продолжает выполняться, если выполняется заданное условие:

```
s = x = 0;
do {    // сумма ряда 1/x от 1 до 1/100
    x++;
    s += 1 / x;
} while (x <= 100);  // значение s = 5.1873...
```

Цикл с постусловием отличается от цикла с предусловием тем, что выполняется хотя бы один раз.

Оператор *прерывания цикла* `break` позволяет досрочно выйти из цикла:

```
var sum = 0;
while (true) {  // бесконечный цикл
    var value = +prompt("Введите число", '');
    if (!value) break;    // выход, если число не
введено
    sum += value;
}  //  sum – сумма введенных чисел
```

Оператор *продолжения цикла* `continue` позволяет продолжить цикл сначала:

```
for (i = 0; i < 10; i++) {
     if (i % 2 == 0) continue;
     alert(i);  //  вывод нечетных значений
}
```

В случае если нужно прервать/продолжить выполнение сразу нескольких циклов, вместе с командами `break/continue` используются метки, представляющие собой идентификатор с символом : (точка с запятой). Метка ставится перед началом прерываемого/продолжаемого цикла:

```
c = 0;
label:  for (i = 0; i < 3; i++) {
     for (j = 0; j < 3; j++) {
          c += i + j;
          if (c > 10) break label;
     }
}  // значение c = 0+1+2+1+2+3+2 = 11
```

Массивы

Простой массив в языке JavaScript – это переменная, содержащая однотипные элементы. Для создания массива и обращения к элементам используются квадратные скобки `[ ]` и индекс (порядковый номер) элемента. Нумерация элементов массива начинается с 0. Пример:

```
var empty = [ ]; // пустой массив
var odd = [0, 2, 4, 6, 8]; // массив из 5 чисел
var list = ["Иван", "Андрей", "Наталья"]; // массив
из 3 слов
var c = odd[2];  // значение c = 4;
```

```
alert(list[1]);  // вывод "Андрей"
```

Поскольку массив является встроенным объектом типа `Array`, для объявления можно использовать его конструктор:

```
var a = new Array(2, 3, 4); // массив из 3 чисел
```

Многомерный массив объявляется с помощью вложенных квадратных скобок `[ ]`. Элементы такого массива адресуются двумя (или более) подряд идущими индексами. Пример:

```
var a = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]; //
матрица 3x3
alert(a[1][1]); // вывод элемента 5
```

К элементам массива также можно обращаться с помощью именованных ключей. Такой массив называется ассоциативным. Ассоциативный массив объявляется с помощью фигурных скобок `{ }`, имена ключей завершаются символом `:` (двоеточие):

```
var earth = { name: "Земля", period: 365.25,
radius: 6378};
```

Ассоциативный массив, так же как и обычный, является объектом типа `Array` и может быть объявлен с помощью конструктора:

```
var earth = new Array();
```

К элементам ассоциативного массива можно обратиться по имени – свойству через символ . (точка) или квадратные скобки [ ]. Обращение по числовому индексу не разрешается:

```
earth.name = "Земля";
earth.radius = 6378;
alert(earth.name); // вывод Земля
alert(earth["radius"]);  // вывод 6378
```

Для перебора всех ключей ассоциативного массива используются цикл `for` и команда `in`:

```
for (key in earth) {
    alert("Ключ "+key+" значение " + earth[key]);
}
```

Функции

Функции являются основными строительными блоками программы. Часть функций, такие как вышеуказанные

`alert(<сообщение>)` или `Math.sqrt(x)`, являются встроенными в объектную модель JavaScript и не требуют объявления.

Пользовательские функции требуют обязательного объявления:

```
function <имя_функции> ([<аргументы>,...]) {
   <тело функции>
}
```

Функция вызывается по ее имени, например:

```
// объявление функции
function sum(a, b) {  // сумма двух чисел
   return (a+b); // возвращаемое значение
}
var c = sum(2,3);  // вызов функции
alert("результат:" + c);  // результат: 5
```

При вызове функции `sum` переменным `a` и `b` присваиваются значения `2` и `3`. Команда `return` возвращает значение функции в основную программу. Возвращаемое значение присваивается переменной `c` и выводится на экран.

Если аргументы не заданы, то они принимают значения `undefined`. Пример:

```
var c = sum(2);  // аргумент b опущен
alert("результат:" + c); // результат: NaN (2 +
undefined)
```

Необязательные аргументы должны размещаться в конце списка. Для задания значения аргумента по умолчанию используется оператор || (ИЛИ):

```
// объявление функции
function sum(a, b) { // сумма двух чисел
   b = b || 0;  // аргумент по умолчанию, значение
0
   return (a+b);
}
var c = sum(2);  // вызов функции
alert("результат:" + c); // результат: 2
```

Если команда `return` указана без параметра, то функция возвращает значение `undefined`. Команда `return` может быть указана в любом месте функции.

Все переменные, объявленные внутри функции, являются *локальными*. Область их действия распространяется только на заданную функцию, из основной программы они не видны. Аргументы

функции также являются локальными переменными.

Переменные, объявленные вне функции, являются *глобальными*. Область их действия распространяется на всю программу, они видны из любой части программы. Локальные переменные перекрывают глобальные. Пример:

```
var a = 0; // глобальная переменная
var b = 1; // глобальная переменная
// описание функции
function f(c) { // локальная переменная
   var a = 2; // локальная переменная
   alert("a=" + a); // вывод локальной переменной
a=2
   alert("b=" + b); // вывод глобальной переменной
b=1
   alert("c=" + c); // вывод локальной переменной
c=3
}
f(3);  // вызов функции, c присваивается 3
alert("a=" + a); // вывод глобальной переменной a=0
alert("b=" + b); // вывод глобальной переменной b=1
alert("c=" + c); // ошибка: глобальная переменная c
не определена
```

Другой способ объявления функции – *функциональное выражение*:

```
<имя_функции> = function(<аргументы>,...) {
   ... тело функции ...
}
```

Например:

```
var sum = function(a, b) { // сумма двух переменных
    return (a+b);
}
var c = sum(2,3);
alert(c); // вывод 5
```

Вызываются функциональные выражения так же, как и обычные функции. Функциональные выражения должны объявляться до их использования, для обычных функций это правило не обязательно.

Функциональное выражение, которое не присваивается переменной, называется анонимной функцией. Анонимная функция вызывается в том месте, где она описана. Пример:

```
function ask(question, yes, no) {  // описание
функции
```

```
   if (confirm(question)) yes();
   else no();
}
ask("Вы согласны?",  // вызов функции
   function() { alert("Да"); },    // анонимная
функция
   function() { alert("Нет"); } // анонимная
функция
);
```

Объекты

Для описания пользовательского объекта используются фигурные скобки `{ }`, в которых перечисляются свойства подобно ассоциативному массиву:

```
person = { name: "Иван", family: "Иванов", age: 20
};
```

Также для описания объекта можно использовать встроенный класс Object:

```
person = new Object();
person.name = "Иван";
person.family = "Иванов";
person.age = 20;
```

Для инициализации объектов можно использовать функцию-конструктор, в которой можно также описать публичные свойства и методы класса с помощью специальной ссылки на класс `this`:

```
function Person(name, family, age) {
   this.name = name;
   this.family = family;
   this.age = age;
   this.getName = function() { alert(this.name); }
}
```

После описания класса экземпляры объектов можно задавать с помощью команды `new`:

```
var alex = new Person("Алексей", "Сидоров", 22);
alert(alex.age);  // вывод 22
alex.getName();  // вывод Алексей
```

Для добавления нового свойства заданного объекта достаточно указать его в программе:

```
alex.group = 144;  // появилось новое свойство
group
```

Если свойства не существует, то при обращении к нему вернется значение `undefined`:

```
alert(alex.sename);  // undefined
```

Для проверки существования свойства можно использовать команду `in`:

```
if ("sename" in alex) alert("Отчество:" +
alex.sename);
```

Для удаления свойств заданного объекта можно использовать команду `delete`:

```
delete alex.group;  // удалено свойство group
```

Для удаления объекта целиком используется указатель на пустую строку `null`:

```
alex = null;  // удаление объекта alex
```

Для добавления новых свойств и методов, общих для всех объектов класса, можно их явно указать в программе с помощью свойства `prototype`:

```
Person.prototype.group = 144;  // у всех объектов
свойство group=144
Person.prototype.show = function() {  // новая
функция show()
   alert("ФИО:" + this.family + " " + this.name);
}
```

Простое функциональное наследование класса можно указать в конструкторе, вызвав функцию `call` базового класса:

```
function Student(name, family, age, group) {
   Person.call(this, name, family, age); //
конструктор базового класса
   this.group = group;
   this.show = function() { alert("Студент группы
"+group); }
}
var alex = new Student("Алексей", "Сидоров", 22,
144);
alex.show(); // вывод Студент группы 144
```

В данном примере класс `Student` наследует свойства и методы вышеописанного класса `Person`, добавляя новое свойство `group` и метод `show()`.

Также можно указать отношение наследования между классами с помощью свойства `prototype`:

```
function Student(name, family, age, group) {
   Person.apply(this, arguments); // вызов
конструктора базового класса
   this.group = group;
}
Student.prototype =
Object.create(Person.prototype);
var alex = new Student("Алексей", "Сидоров", 22,
144);
```

Конструктор базового класса `Person` вызывается с помощью специальной функции `apply`, которая передает ей ссылку на текущий класс `this` и полный список аргументов `arguments`. Обратите внимание, что перед `Object` отсутствует команда `new`.

Копирование объектов, а также массивов осуществляется по ссылке. Это означает, что при присвоении объекта нескольким переменным и изменении одной из них будут изменены и все остальные, так как объект, на который они указывают, изменен. Пример:

```
var alex = new Person("Алексей", "Сидоров", 22); //
конструктор
var green = alex; // присвоение green
green.age = 25;  // изменение возраста
alert(alex.age); //  вывод 25 (объект изменен)
```

Таким образом, для осуществления полноценного копирования необходимо использовать явное присвоение всех элементов объекта:

```
for (key in alex) {
    green[key] = alex[key];
}
```

Приватные свойства и методы в классе описываются с помощью локальных переменных и вложенных функций в конструкторе:

```
function Person(name, family, age) { // конструктор
   var kinder = false; // приватное свойство
   this.name = name; // публичное свойство
   this.family = family; // публичное свойство
   this.age = age;  // публичное свойство
   function setKinder() {  // приватный метод
       kinder = (age < 16);
   }
```

```
    this.isKinder = function() {  // публичный метод
        setKinder();  // вызов приватного метода
        if (kinder) alert("Да");
        else  alert("Нет");
    }
}
var alex = new Person("Алексей", "Иванов", 22);
alex.isKinder(); // вывод Нет
```

Обработка исключений

При выполнении некоторых операций могут возникнуть ошибки времени выполнения. Для их перехвата используется механизм обработки исключений. Механизм обработки исключений реализован с помощью операторов:

```
try { <программа> } catch(e) { <обработка> } [
finally { <завершение> } ]
```

Блок `<программа>` представляет собой исходный код, в котором осуществляется перехват ошибки. Блок `<обработка>` служит для обработки ошибки. Необязательный блок `<завершение>` представляет собой команды, выполняемые после блоков `<программа>` и `<обработка>` в любом случае. Пример:

```
try {
   var x = prompt("введите число");
   var q = x*a;
   alert("Результат:"+q);
} catch(e) {
   alert(e.message);
}
```

При попытке выполнить программу будет выведено сообщение *a is not defined*, поскольку переменная *a* не определена. Описание ошибки хранится в свойстве `message` объекта `Error`.

Генерацию исключений можно вызвать принудительно с помощью команды `throw`:

```
try {
   var x = prompt("введите число");
   if (x<0) throw "Число должно быть больше 0";
   var q = Math.sqrt(x);
   alert("Корень:"+q);
} catch(e) {
   alert(e);
}
```

В этом случае в переменную e записывается непосредственно генерируемое сообщение.

Объекты JavaScript

Язык JavaScript является объектно-ориентированным и использует объекты для своей работы. Фактически массивы, строки и даже числа являются встроенными объектами `Array`, `String`, `Number` и `Boolean`. Встроенными объектами также являются даты `Date`. Кроме того, существуют глобальные объекты, такие как `Math`, `JSON` и др.

Объект Number. Представляет собой обычное число. Основные методы:

`Number(<параметр>)` – представляет собой конструктор числа. В качестве параметра может передаваться любое значение. Пример:

```
var a = new Number("12.24"); // a = 12.24
```

`toFixed(n)` – преобразование к действительному числу с `n` значащими цифрами после точки:

```
var a = 3.1415926;
a.toFixed(3); // 3.142
```

`toExponential(n)` – преобразование к числу в экспоненциальной форме с `n` значащими цифрами после точки:

```
var a = 86421;
a.toExponential(3); // 8.642e+4
```

`toPrecision(n)` – преобразование числа с `n` общим количеством значащих цифр:

```
var a = 123.456;
a.toPrecision(4); // 123.5
```

`toString(x)` – преобразование числа в строковое представление в системе счисления по основанию `x`:

```
var a = 12;
a.toString(16); // символ c
```

Объект String. Представляет собой текстовую строку, заключенную в кавычки или апострофы. При необходимости строка может содержать управляющие последовательности: *\n* – перевод строки, *\DDD* – десятичный код символа, *\xHH* – шестнадцатеричный код символа, *\uHHHH* – символ в кодировке Unicode, *\'* –

экранирование апострофа, \" – экранирование кавычки, \\ – экранирование обратной косой черты. Основные методы и свойства:

String(<параметр>) – представляет собой конструктор строки. В качестве параметра инициализации может передаваться любое значение. Пример:

```
var str = new String(12.45);
alert(str.length); // вывод 5
```

Свойство length определяет длину строки в символах:

```
"Привет!".length; // 7
```

Обращение по индексу строки [i] возвращает символ на позиции i. Нумерация позиций начинается с 0:

```
"Привет!"[3]; // символ "в"
```

Метод charCodeAt(i) возвращает десятичный код символа строки на позиции i (в кодировке Unicode):

```
"Привет!".charCodeAt(3); // 1074
```

fromCharCode(x1,...,xN) – возвращается текстовая строка с кодами символов x1,...,xN:

```
String.fromCharCode(1055,1088,1080,1074,1077,1090,3
3); // Привет!
```

concat(<строка>) – сложение (конкатенация) строк. Действует как оператор + (плюс):

```
"Иван".concat("Иванович"); // ИванИванович
```

indexOf(<подстрока>[, i]) – поиск позиции подстроки в заданной строке, начиная с позиции *i* (по умолчанию 0) от начала строки. Если подстрока не найдена, функция возвращает значение -1:

```
alert("Математика".indexOf("а")); // вывод 1
```

lastIndexOf(<подстрока>[, i]) – поиск позиции подстроки в заданной строке, начиная с позиции i (по умолчанию 0) от конца строки. Если подстрока не найдена, функция возвращает значение -1:

```
alert("Математика".lastIndexOf("а")); // вывод 9
```

substr(i [, <длина>]) – возвращает подстроку строки начиная от позиции i заданной длины. В случае если длина не указана или превышает длину строки, подстрока обрезается до конца

строки.

```
var a = "Математика";
alert(a.substr(2,4)); // вывод тема
```

`substring(i [, j])` – возвращает подстроку строки начиная от позиции `i` до `j` (не включая его). Если позиция `j` не указана, то возвращает подстроку от позиции `i` до конца.

```
var a = "Математика";
alert(a.substring(2,6)); // вывод тема
```

`slice(i [, j])` – подобно функции `substring` возвращает подстроку строки начиная от позиции `i` до `j` (не включая его). В отличие от `substring`, может иметь отрицательное значение второго параметра `j` – в этом случае, индекс отсчитывается от конца строки:

```
var a = "Математика";
alert(a.slice(2,-4)); // вывод тема
```

`split(<разделитель>[, n])` – разбивает строку на массив из `n` элементов по заданному разделителю. Количество элементов массива – необязательный параметр. Если в качестве разделителя указывается пустая строка, то исходная строка разбивается на массив отдельных символов. Пример:

```
var a = "до конца";
var b = a.split(" "); // Массив ["до", "конца"]
var c = a.split("",4); // Массив ["д","о"," ","к"]
```

`toLowerCase()` – приводит строку к нижнему регистру:

```
var a = "Привет";
alert(a.toLowerCase()); // вывод: привет
```

`toUpperCase()` – приводит строку к верхнему регистру:

```
var a = "мир";
alert(a.toUpperCase()); // вывод: МИР
```

Оборачивающие методы позволяют содержимое строки обернуть в HTML теги:

`big()` – увеличение шрифта (`<big>строка</big>`);
`small()` – уменьшение шрифта (`<small>строка</small>`);
`bold()` – полужирный (`<b>строка</b>`);
`italic()` – курсив (`<i>строка</i>`);
`sub()` – подстрочный индекс (`<sub>строка</sub>`);
`sup()` – надстрочный индекс (`<sup>строка</sup>`);

`fontcolor(<цвет>)` – цвет шрифта (`<font color="цвет">строка</font>`);

`fontsize(<разм>)` – размер шрифта (`<font size="разм">строка</font>`);

`link(<ссылка>)` – гиперссылка (`<a href="ссылка">строка</a>`).

Объект Array. Представляет собой массив. Основные свойства и методы:

`Array(<n>)` – конструктор массива из `n` элементов. Если параметр n является числом, то резервируется память для элементов массива. Все элементы принимают значения `undefined`. Для инициализации элементов массива их необходимо передать в качестве параметров:

```
var a = new Array(5); // массив из 5 пустых
элементов
var b = new Array(1,2,3); // массив из 3 элементов,
b = [1,2,3]
```

Обращение к элементам массива производится по их индексу `[i]`. Нумерация элементов начинается с 0:

```
var b = [1, 2, 3];
alert(b[1]); // вывод 2
```

Свойство `length` определяет размер массива в элементах:

```
var b = [1, 2, 3];
alert(b.length); // вывод 3
```

Уменьшение свойства `length` приводит к уменьшению размера массива и освобождению памяти от элементов. Аналогично увеличение свойства `length` резервирует дополнительную память под элементы. Также увеличение свойства `length` происходит автоматически при обращении к несуществующим индексам.

```
var b = [1, 2, 3, 4, 5]; // размер массива 5
b.length = 3; // размер массива 3, b = [1, 2, 3]
alert(b); // вывод 1,2,3
b[4] = 5; // размер массива снова 5, b = [1, 2, 3,
undefined, 5]
alert(b); // вывод 1,2,3,,5
```

Методы, не изменяющие исходный массив:

`concat(<массив>)` – объединение массивов в один. Возвращается объединенный массив. Пример:

```
var a = [1, 2, 3];
var b = new Array("Привет", "мир");
var c = a.concat(b); // c = [1, 2, 3, "Привет",
"мир"]
```

join(<разделитель>) – объединение всех элементов массива в одну строку с указанным разделителем:

```
var a = new Array(1, 2, 3);
var str = a.join(""); // str = "123"
```

slice(i [, j]) – создание нового массива из исходного с индексами элементов от i до j (не включая его). Если параметр j не задан, то новый массив создается до конца исходного массива. При отрицательных значениях второго параметра j индекс отсчитывается от конца исходного массива:

```
var a = [1, 3, 2, 4, 5];
var b = a.slice(1,-1); // b = [3, 2, 4]
```

Методы, изменяющие исходный массив:

push(<элемент>) – добавление элемента в конец массива.

pop()– извлечение последнего элемента массива.

unshift(<элемент>) – добавление элемента в начало массива.

shift()– извлечение первого элемента массива. Примеры:

```
var a = [1, 2, 3];
a.push(5); // добавление элемента в конец, a = [1,
2, 3, 5]
var b = a.pop(); // извлечение последнего элемента,
b = 5, a = [1, 2, 3]
a.unshift(0); // добавление элемента в начало,
a = [0, 1, 2, 3]
var c = a.shift(); // извлечение первого элемента,
b = 0, a = [1, 2, 3]
```

Из-за особенностей реализации сдвиги shift и unshift выполняются значительно медленнее, чем операции со стеком push и pop.

reverse()– изменяет порядок элеметов в массиве на противоположный:

```
var a = [1, 2, 3, 4, 5]
a.reverse(); // массив a = [5, 4, 3, 2, 1]
```

sort(<функция>) – сортировка элементов массива с помощью пользовательской функции сравнения. Если функция не задана, то

используется строковая сортировка по возрастанию:

```
var a = [1, 8, 15, 6, 3];
a.sort(); // массив a = [1, 15, 3, 6, 8]
```

При использовании пользовательской функции сравнения в качестве сортируемых элементов массива используются два ее аргумента (например, `a` и `b`), а ее имя передается в качестве аргумента функции `sort`. Функция сравнения в зависимости от параметров должна возвращать одно из трех значений: меньше 0 – если параметр `b` следует за `a`, равен 0 – если параметры равны, больше 0 – если параметр `a` следует за `b`. Пример:

```
var a = [1, 8, 15, 6, 3];
function comp(a, b) { return (a-b); }
a.sort(comp); // массив a = [1, 3, 6, 8, 15];
```

`splice(i, len [, <список>])` – срез массива, вырезает из исходного массива `len` элементов, начиная от `i`, возвращает их и заменяет их в исходном массиве новым списком элементов, если он задан. Пример:

```
var a = ["в", "час", "вечерний", "прохладный"];
var x = a.splice(1, 2, "день", "осенний");
alert(x); // вывод: час,вечерний
alert(a); // вывод: в,день,осенний,прохладный
```

Объект Date. Представляет собой дату. Основные свойства и методы:

`Date(<дата>)` – конструктор даты. Имеются различные способы инициализации даты:

- строка формата ISO 8601 Extended "YYYY-MM-DDTHH:mm:ss.msZ" (год, месяц, день, T - символ-разделитель, часы, минуты, секунды, миллисекунды, Z - символ для GMT или часовое смещение +hhmm);
- укороченная строка ISO "YYYY-MM-DD" (год, месяц, день);
- число msUTC (число миллисекунд POSIX от 01.01.1970 г. GMT+0);
- массив [YYYY, NM, DD, HH, mm, ss] (NM - порядковый номер месяца начиная с 0);
- укороченный массив [YYYY, NM, DD].

Примеры:

```
var a = new Date("2015-12-26"); // 26 декабря 2015
года
```

```
var b = new Date("2016-04-01T12:30:00"); //
01.04.2016 г., 12 ч 30 мин
var d = new Date(1461917448231); // 29.04.2016 г.,
11:10:48
var c = new Date(2016, 03, 12, 16, 45, 00); //
12.04.2016 г., 16 ч 45 мин
```

Для установки текущего значения даты и времени параметры в конструкторе можно не указывать:

```
var a = new Date(); // текущая дата и время
```

Для извлечения параметров из даты используются следующие методы:

`getFullYear()` – получение номера года (от 1970 и далее);

`getMonth()` – получение порядкового номера месяца (от 0 – января до 11 – декабря);

`getDate()` – получение числа месяца (от 1 до 31);

`getDay()` – получение номера дня недели (от 0 – воскресенья до 6 – субботы);

`getHours()` – получение часа (от 0 до 23);

`getMinutes()` – получение минут (от 0 до 59);

`getSeconds()` – получение секунд (от 0 до 59);

`getMilliseconds()` – получение локального времени в миллисекундах POSIX.

Пример:

```
var a = new Date("2016-04-01T12:30:00");
var year = a.getFullYear(); // 2016
var month = a.getMonth(); // 3
var d = a.getDate(); // 1
var hours = a.getHourse(); // 12
```

Все методы извлекают данные для текущей временной зоны. Для мирового времени GMT+0 имеются аналогичные функции с префиксом UTC, например `getUTCHourse()` и т.д. Кроме того, существуют методы, использующие мировое время:

`getTime()` – получение мирового времени в миллисекундах POSIX GMT+0;

`getTimezoneOffset()` – возвращение разницы между локальным и мировым временем в минутах:

```
var h = new Date().getTimezoneOffset(); // -180 для
Москвы (3 часа)
```

Для установки параметров даты используются соответствующие

методы с префиксом `set`:

`setFullYear(YYYY)` – установка номера года;

`setMonth(NM)` – установка порядкового номера месяца (0 - январь);

`setDate(DD)` – установка числа месяца;

`setDay(day)` – установка номера дня недели (0 - воскресенье);

`setHours(HH)` – установка часа;

`setMinutes(mm)` – установка минут;

`setSeconds(ss)` – установка секунд;

`setMilliseconds(ms)` – установка локального времени в миллисекундах POSIX.

При установке значений в общем случае остальные исходные параметры даты не изменяются. Однако если значения выходят за пределы диапазона, то параметры автоматически корректируются. Пример:

```
var a = new Date(); // текущее дата и время
a.setFullYear(2015); // установлен 2015 год
a.setMonth(1); // установлен месяц февраль
a.setDate(29); // поскольку в 2015 году февраль не
високосный,
alert(a); // будет выведена дата 01.03.2015 и
текущее время
```

Все методы устанавливают данные для текущей временной зоны. Для мирового времени GMT+0 имеются аналогичные функции с префиксом UTC, например `setUTCHourse(HH)` и т.д. Кроме того, имеется метод, использующий мировое время:

`setTime(msUTC)` – установка мирового времени в миллисекундах POSIX GMT+0.

Для строкового преобразования даты и времени используются следующие методы:

`toString()`– преобразование даты и времени в текстовую строку. Вид строки зависит от версии браузера, например:

```
var a = new Date("2016-04-01"); // установка даты
alert(a.toString()); // вывод Fri Apr 01 2016
03:00:00 GMT +0300 MSK
```

`toDateString()`– преобразование только даты в текстовую строку.

`toTimeString()`– преобразование только времени в текстовую строку.

`toLocaleString([<язык>, <опции>])` – преобразование

даты и времени в текстовую строку в локализованном формате. Язык и ассоциативный массив опций – необязательные параметры, устанавливающие формат строки (не поддерживается Microsoft Internet Explorer). Пример:

```
var a = new Date("2016-04-01"); // установка даты
var ops = {
   day: 'numeric',
   month: 'long',
   year: 'numeric'
}
alert(a.toLocaleString("ru",ops)); // вывод 1
апреля 2016 г.
```

`Date.parse(<дата>)` – возвращает число миллисекунд POSIX на заданную дату. Дата указывается в вышеуказанном формате ISO 8601 Extended. Пример:

```
var a = Date.parse("2016-04-01T12:45:12+0300");
alert(a); // вывод 1459503912000
```

При использовании объектов `Date` в числовом контексте возвращается количество миллисекунд. Это свойство часто используется для анимации элементов страницы, однако лучше применить отдельный метод `now()`:

`Date.now()`– возвращает число миллисекунд POSIX. Этот метод работает быстрее, чем `getTime()`, так как при этом сам объект `Date` не создается. Его можно использовать даже для профилирования программ. Пример:

```
var a = Date.now();
for (var x = 1, s = 1; x < 100000; x++)  s +=
1/(x*x); // сумма ряда 1/x2
var b = Date.now();
alert("сумма:"+s+", "+(b - a)+" ms"); // 2.644...,
3 ms
```

Обратите внимание, что оператор `new` перед `Date` в примере не ставится.

Управление временем. Профилирование

В программах, требующих профилирования, вместо объекта `Date` лучше использовать специальное свойство `performance` (не поддерживается Microsoft Internet Explorer):

`performance.now()`– возвращает число миллисекунд POSIX. Метод работает быстрее, чем `Date.now()`, и точнее в 1000 раз:

```
var a = performance.now();
for (var x = 1, s = 1; x < 100000; x++)  s +=
1/(x*x); // сумма ряда 1/x2
var b = performance.now();
alert("сумма:"+s+", "+(b - a)+" ms"); // 2.644...,
2.315 ms
```

Для управления временем в JavaScript имеется несколько полезных глобальных методов, которые с успехом применяются при анимации объектов.

`<переменная> = setTimeout(<функция>, <задержка>)` – устанавливает пользовательскую функцию, которая будет выполнена с заданной задержкой (в миллисекундах). Пример:

```
var func = function() { alert("1 секунда"); };
setTimeout(func, 1000); // вывод сообщения через 1
секунду
```

`clearTimeout(<переменная>)` – отменяет выполнение пользовательской функции, установленной методом `setTimeout`. Переменная представляет собой идентификатор таймера `setTimeout`. Пример:

```
var timer = setTimeout(function() { alert("1
секунда") }, 1000);
clearTimeout(timer);
```

В данном случае сообщение не будет выведено.

`<переменная> = setInterval(<функция>, <задержка>)` – устанавливает пользовательскую функцию, которая будет периодически выполняться с заданной задержкой (в миллисекундах). Пример:

```
var time = 0;
var func = function() {
    time++;
    alert(time+" секунда");
};
setInterval(func, 1000); // вывод сообщения каждую
секунду
```

В отличие от метода `setTimeout`, пользовательская функция в данном примере будет выполняться больше одного раза, увеличивая каждый раз счетчик секунд на 1.

`clearInterval(<переменная>)` – отменяет выполнение пользовательской функции, установленной методом `setInterval`. Переменная представляет собой идентификатор таймера

`setInterval`. Пример:

```
var time = 0;
var func = function() {
    time++;
    alert(time+" секунда");
    if (time >= 5) clearInterval(timer); // отмена
выполнения
};
var timer = setInterval(func, 1000); // вывод 5
сообщений через секунду
```

Объект Math. Является глобальным объектом. Представляет собой библиотеку, которая включает наиболее распространенные математические константы и функции. Основные свойства:

`E` - константа **e** (2.71828...);

`PI` - константа **π** (3.14159...);

Основные методы:

`abs(x)` – модуль числа x;

`acos(x)` – арккосинус числа x (угол в радианах);

`asin(x)` – арксинус числа x (угол в радианах);

`atan(x)` – арктангенс числа x (угол в радианах);

`ceil(x)` – округление числа x до целого в большую сторону;

`cos(x)` – косинус угла x (в радианах);

`exp(x)` – экспонента числа x;

`floor(x)` – округление числа x до целого в меньшую сторону;

`log(x)` – натуральный логарифм числа x;

`max(a, b)` – максимальное значение a и b;

`min(a, b)` – минимальное значение a и b;

`pow(x, y)` – вычисление функции x в степени y;

`random()` – вычисление случайного числа в диапазоне от 0 до 1;

`round(x)` – округление числа x до ближайшего целого;

`sin(x)` – синус угла x (в радианах);

`sqrt(x)` – квадратный корень числа x;

`tan(x)` – тангенс угла x (в радианах);

Все константы и функции возвращают значения с двойной точностью. Примеры:

```
var sqare = r * r * Math.PI; // площадь круга с
радиусом r
var y = Math.sin(x); // функция y = sin(x);
```

Отладка программы

Для отладки JavaScript программы и вывода значений переменных обычно используется функция `alert`, выводящая текстовое сообщение. Однако в некоторых случаях вывод окна влияет на поведение сайта и такой способ является нежелательным.

Гораздо более удобным и гибким инструментом является вывод с помощью встроенного метода `console.log(<сообщение>)`. С помощью этого метода можно выводить сообщения в отладочную панель браузера (для этого она должна быть предварительно включена). Пример:

```
console.log("Текстовое сообщение");
```

Для включения отладочной панели в большинстве браузеров используется комбинация клавиш <Ctrl> + <Shift> + <J>. Кроме этого, для отладки программ в браузере Mozilla Firefox используется расширение Firebug, которое включается клавишей <F12>.

Для вывода значений объектов и массивов программы также можно использовать метод `console.dir(<переменная>)`. В отличие от метода `console.log`, этот способ позволяет вывести структуру объектной переменной со всеми значениями полей и наименованиями методов.

1.2. Объектная модель браузера

JavaScript использует стандартную объектную модель, вершиной которой является объект `window` (рис. 4). Он также является главным объектом всей иерархии JavaScript.

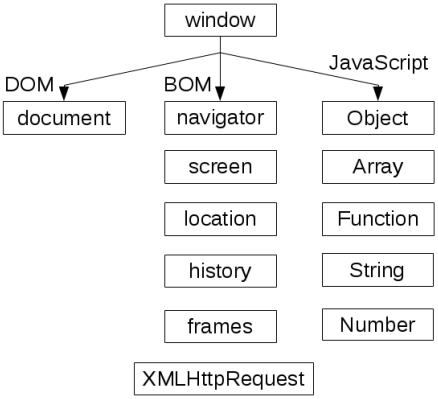

Рис. 4. Объектная модель браузера

Объектная модель браузера (Browser Object Model, BOM) состоит из следующих основных объектов:

1) `window` – окно браузера. Основные свойства:

`window.innerWidth` – ширина окна браузера;

`window.innerHeight` – высота окна браузера;

`window.opener` – ссылка на родительское окно;

`window.status` – содержимое панели статуса браузера;

`window.screenX` и `window.screenY` – координаты левого верхнего угла окна браузера относительно экрана;

`window.pageXOffset` и `window.pageYOffset` – координаты смещения скроллинга HTML страницы в браузере.

Основные методы объекта `window`:

`window.open(<URL>, <win> [, params])` – открытие нового окна, где

`URL` – адрес отображаемого HTML документа;

`win` – имя создаваемого окна (или _blank – для нового окна);

`params` – текстовая строка с параметрами окна.

К объекту окна можно также обратиться с помощью инициализированной переменной. Пример:

```
var
params="menubar=yes,location=yes,resizable=yes,scro
llbars=yes,
status=yes,width=500,height=450";
var win = window.open("http://yandex.ru",
"Yandex", params);
win.focus(); // получить фокус нового окна
```

`window.close()` – закрытие окна;

`window.blur()` – сделать окно неактивным;

`window.focus()` – сделать окно активным;

`window.print()` – распечатать содержимое окна;

`window.moveTo(x,y)` – перемещение ЛВУ окна в позицию с координатами (`x,y`);

`window.resizeTo(w,h)` – изменение размера окна, где `w` – ширина, `h` – высота.

Поскольку `window` является глобальным объектом, то к нему относятся и все глобальные методы, такие как `alert()`, `prompt()` и др. Например:

```
window.alert("Сообщение"); // вывод сообщения
```

2) `navigator` – объект свойств браузера и операционной

системы. Основные свойства:

`navigator.appName` – имя браузера;

`navigator.appVersion` – версия браузера;

`navigator.userAgent` – полная информация о браузере;

`navigator.platform` – информация об операционной системе пользователя;

3) `screen` – объект экрана. Основные свойства:

`screen.width` – ширина экрана;

`screen.height` – высота экрана;

`screen.colorDepth` – глубина цвета;

4) `location` – URL адрес загруженной HTML страницы. Основные свойства:

`location.href` – строка URL;

`location.hostname` – домен URL адреса;

`location.pathname` – путь относительно корня сайта;

`location.search` – строка запроса.

Основные методы объекта `location`:

`location.assign(<url>)` – загрузить документ по данному URL;

`location.reload([true])` – перезагрузить документ по текущему URL. Если установлен параметр `true`, то документ перезагружается с сервера, иначе – браузер будет использовать свой кэш.

`location.replace(<url>)` – замена текущего документа на страницу по указанному URL. В отличие от использования `assign()`, страница не сохраняется в истории браузера;

`location.toString()` – строковое представление URL;

5) `history` – объект списка посещенных страниц браузером. Основные свойства:

`history.length` – количество элементов списка.

Основные методы объекта `history`:

`history.back()` – загрузить предыдущий URL;

`history.forward()` – загрузить последующий URL;

`history.go(<index>)` – перейти на указанный URL в списке по индексу `<index>`. Например, выполнение команды `history.go(-1)` аналогично команде `history.back()` (перейти на предыдущий адрес).

1.3. Доступ к элементам страницы

Объект `document` хранит объектную структуру HTML документа (Document Object Model, DOM). С помощью него можно получить доступ к отдельным HTML тегам, добавлять и изменять их содержимое, а также определять поведение этих элементов.

Основные свойства объекта `document`:

`document.cookie` – возвращает ключики, связанные с HTML документом. Это ассоциативный массив вида *ключ=значение*, хранящийся в браузере пользователя. Обычно используется для хранения пользовательских настроек для каждого конкретного сайта или отдельной страницы;

`document.head` – возвращает тег заголовка <`head`>;

`document.title` – возвращает заголовок страницы <`title`>;

`document.forms` – возвращает массив, содержащий все формы на странице;

`document.images` – возвращает массив изображений на HTML странице;

`document.links` – возвращает массив гиперссылок на HTML странице;

`document.lastModified` – возвращает время последнего изменения документа;

`document.documentElement` – возвращает корневой тег документа <`html`>;

`document.body` – возвращает тег основного тела документа <`body`>.

Основные методы объекта `document`:

`document.createElement(<тег>)` – создает HTML элемент по имени тега;

`document.getElementById(<Id>)` – возвращает HTML элемент по его идентификатору (атрибуту `id`);

`document.getElementsByName(<имя>)` – возвращает массив HTML элементов по имени (атрибуту `name`);

`document.getElementsByTagName(<тег>)` – возвращает массив HTML элементов по имени тега;

`document.getElementsByClassName(<класс>)` – возвращает массив HTML элементов по имени класса (атрибуту `class`);

`document.querySelector(<селектор>)` – возвращает первый подходящий HTML элемент по его CSS селектору. Параметр

`<селектор>` описывается так же, как в стилевой таблице. Также он может включать несколько CSS селекторов, разделенных запятыми. Пример (возвращает элемент `div` с атрибутом `id="test"`):

```
var el = document.querySelector("div#test");
```

`document.querySelectorAll(<селектор>)` – возвращает массив HTML элементов по их CSS селектору;

`document.write(<строка>)` и `document.writeln(<строка>)` – запись HTML строки в поток вывода браузера.

Пример:

```
<div id="test"></div>
<script>
      var dd = document.getElementById("test");
      dd.innerHTML = "Тестовые данные";
</script>
```

После выполнения скрипта тег `<div>` будет иметь вид:

```
<div id="test">Тестовые данные</div>
```

Свойства и методы элементов DOM

Для каждого HTML элемента в DOM создается соответствующий объект. Все объекты DOM имеют набор стандартных свойств и методов и образуют древовидную иерархию (рис. 5). Кроме того, некоторые объекты (например, объекты формы) имеют дополнительные свойства и методы, определяющиеся их назначением.

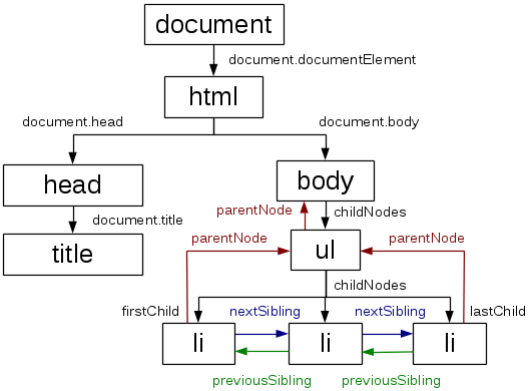

Рис. 5. Древовидная иерархия свойств и методов объекта DOM

Общие основные свойства элементов DOM:

`attributes` – возвращает массив атрибутов;
`childNodes` – возвращает массив потомков;
`parentNode` – возвращает родительский элемент;
`firstChild` – возвращает первого потомка;
`lastChild` – возвращает последнего потомка;
`nextSibling` – возвращает последующий соседний элемент;
`previousSibling` – возвращает предыдующий соседний элемент;
`nodeName` или `tagName` – имя узла (тега);
`nodeType` – тип узла (1 – элемент, 2 – атрибут, 3 – текстовой узел);
`nodeValue` или `data` – получить (установить) значение текстового узла;
`innerHTML` – HTML содержимое элемента (без тега);
`outerHTML` – HTML содержимое элемента (вместе с тегом);
`textContent` – текстовое содержимое элемента (без тегов);
`className` – получить/установить имя класса элемента;
`id` – получить/установить идентификатор элемента;
`style` – получить/установить CSS стиль элемента

Текстовой узел является прямым потомком элемента, поэтому для обращения к его содержимому необходимо обратиться к свойству `childNodes`:

```
var text = document.body.childNodes[0].data;
alert(text); // вывод содержимого тега <body>.
```

Общие основные методы элементов DOM:

`getAttribute(<атрибут>)` – получить значение атрибута;
`setAttribute(<атрибут>,<значение>)` – установить значение атрибута;
`removeAttribute(<атрибут>)` – удалить атрибут;
`appendChild(<элемент>)` – добавление элемента потомка;
`removeChild(<элемент>)` – удаление элемента потомка;
`hasChildNodes()` – возвращает `true`, если узел имеет потомков;
`focus()` – делает элемент активным;
`blur()` – делает элемент неактивным;
`click()` – активирует обработчик нажатием правой кнопки мыши на элементе.

Пример использования методов DOM:

```
<ul id="test"></ul>
<script>
       var dd = document.getElementById("test");
       var el = Array();
       for (var i=0; i<3; i++) {
       el[i] = document.createElement("li");
              el[i].innerHTML = "тест " + (i+1);
              dd.appendChild(el[i]);
       }
</script>
```

После выполнения скрипта список `<ul>` на HTML странице будет иметь вид:

```
<ul id="test">
 <li>тест 1</li>
 <li>тест 2</li>
 <li>тест 3</li>
</ul>
```

Специальные свойства элементов DOM

Кроме общих свойств многие элементы DOM имеют также специальные свойства. Использование этих свойств позволяет обращаться к HTML атрибутам соотвествующих тегов, получать и устанавливать их значения. Например, для обращения к файлу изображения, устанавливаемому с помощью атрибута `src` тега `img`, используется соответствующее свойство `img.src` и т.д.:

```
<img src="picture1.png" alt="изображение"
onclick="change()">
<script>
 function change() { // функция изменения
изображения
       document.images[0].src = "picture2.png";
 }
</script>
```

Для элементов HTML таблиц используются следующие свойства и методы:

`table.rows` – возвращает массив строк таблицы;

`table.createCaption()` – создает тег заголовок таблицы `<caption>`;

`table.deleteCaption()` – удаляет тег заголовок таблицы `<caption>`;

`table.createTHead()` – создает тег шапку таблицы `<thead>`;

`table.deleteTHead()` – удаляет тег шапку таблицы `<thead>`;

`table.createTFoot()` – создает тег подвал таблицы `<tfoot>`;

`table.deleteTFoot()` – удаляет тег подвал таблицы `<tfoot>`;

`table.insertRow(i)` – добавляет строку таблицы в позицию `i`;

`table.deleteRow(i)` – удаляет `i-ю` строку из таблицы;

`tr.cells` – возвращает массив ячеек таблицы;

`tr.rowIndex` – возвращает текущую позицию строки в таблице;

`tr.insertCell(i)` – добавляет ячейку строки в позицию `i`;

`tr.deleteCell(i)` – удаляет `i-ю` ячейку из строки;

`td.cellIndex` – возвращает текущую позицию ячейки в строке.

Пример:

```
var table = document.getElementById("idTable");
var row = table.insertRow(0); // вставка пустой
строки <tr> в начало
var cell1 = row.insertCell(0); // вставка пустой
ячейки <td> в позицию 0
cell1.innerHTML = "новая ячейка";
```

Использование стилей

Свойство `style` элемента DOM является объектом и позволяет изменять внешний вид элемента в соответствии с CSS правилами. Большинство CSS атрибутов представлены соответствующими строковыми свойствами элемента `style`. Для обращения к этим свойствам используется следующее правило: если CSS атрибут состоит из нескольких слов, разделенных дефисом, то соответствующее свойство `style` также будет составлено из этих слов, но записанных подряд. При этом все слова, следующие после дефиса, должны начинаться с заглавной буквы. Например, CSS атрибуту `font-weight` для указанного элемента в JavaScript будет соответствовать свойство `element.style.fontWeight`, атрибуту `text-align` будет соответствовать свойство `element.style.textAlign` и др. Используя свойство `style` вместе с обработкой событий, можно сделать динамическую стилизацию страницы, например скрыть или показать некоторые элементы на странице в зависимости от выбора того или иного пункта меню, изменить цветовую гамму элементов и др.

Пример использования объекта `style`:

```
document.body.style.backgroundColor = "red"; //
установить красный фон
var dd = document.getElementById("test");
dd.style.visibility = "hidden"; // скрыть элемент с
идентификатором test
```

Обработка событий

Управление событиями на странице является одной из важнейших возможностей JavaScript, необходимых для создания интерактивного интерфейса веб-приложений.

События – это функции-обработчики, которые могут быть привязаны к элементам HTML страниц. Обработчики выполняются после того, как произойдет их активация. Разные типы событий имеют разные активирующие действия.

Основные виды событий:

`onblur` – элемент стал неактивным;
`onchange` – содержимое элемента изменено;
`onclick` – на элементе произведен щелчок мыши;
`ondblclick` – на элементе произведен двойной щелчок мыши;
`onerror` – при загрузке элемента произошла ошибка;
`onfocus` – элемент стал активным;
`onkeydown` – нажата клавиша;
`onkeypress` – произошло нажатие на клавиатуру;
`onkeyup` – нажатая клавиша отпущена;
`onload` – элемент полностью загружен;
`onmousedown` – на элементе нажата клавиша мыши;
`onmouseout` – курсор мыши вышел за пределы элемента;
`onmouseover` – курсор мыши наведен на элемент;
`onmouseup` – на элементе отпущена клавиша мыши;
`onreset` – форма сброшена;
`onresize` – изменен размер документа;
`onscroll` – содержимое элемента прокручено;
`onselect` – текст элемента выделен;
`onsubmit` – форма передана на сервер;
`onunload` – страница закрыта (для тега `<body>`).

Чтобы задать обработчик, необходимо связать его с HTML элементом. Для этого существует несколько способов.

1. Через соответствующий атрибут HTML тега `on<событие>`:

```
<div id="test" onclick="alert('Клик')">Нажми
```

```
меня</div>
```

Данный способ наиболее простой, но смешивает JavaScript код с HTML разметкой, поэтому он недостаточно эффективный.

2. Через свойство элемента DOM `on<событие>` в JavaScript:

```
document.getElementById("test").onclick =
function() { alert('Клик') }
```

Этот способ позволяет вынести всю обработку в JavaScript, но при этом на каждое событие можно задать только один обработчик. Способ используется в большинстве простых программ.

3. С помощью отдельного метода, рекомендованного консорциумом W3C: `addEventListener(<событие>,<обработчик>[,<фаза>])`:

```
document.getElementById("test").addEventListener("c
lick", message);
function message() { alert('Клик') }
```

Это наиболее гибкий способ задания обработчиков, он используется для сложных приложений. Для удаления обработчика можно использовать метод `removeEventListener(<событие>, <обработчик>[,<фаза>])`:

```
document.getElementById("test").removeEventListener
("click", message);
```

Если параметр `<фаза>` установлен в `true`, то при срабатывании во вложенных элементах обработчик можно вызвать в фазе "захвата", иначе - в фазе "всплывания" (см. ниже).

В старых версиях браузера Microsoft Internet Explorer (до 9) вместо `addEventListener` и `removeEventListener` использовались аналогичные функции `attachEvent` и `removeEvent`.

За информацию о произошедших событиях отвечает специальный DOM объект `events`, который можно передать в функцию обработчика и использовать его свойства. События хранятся в отдельных атрибутах `events`. Список атрибутов событий:

`altKey` – нажата клавиша `<Alt>` во время вызова события;

`button` – нажата клавиша мыши (0 – левая, 1 – средняя, 2 – правая) во время вызова события;

`clientX` – горизонтальные координаты указателя мыши относительно границ документа во время вызова события;

`clientY` – вертикальные координаты указателя мыши

относительно границ документа во время вызова события;

`ctrlKey` – нажата клавиша `<Ctrl>` во время вызова события;

`screenX` – горизонтальные координаты указателя мыши относительно границ экрана во время вызова события;

`screenY` – вертикальные координаты указателя мыши относительно границ экрана во время вызова события;

`shiftKey` – нажата клавиша `<Shift>` во время вызова события;

`target` – возвращает элемент DOM, который вызвал событие;

`type` – возвращает имя события.

Используя эти атрибуты, из объекта events можно извлечь много полезной информации о его состоянии. Объект events всегда передается в функцию обработчика первым параметром. Например:

```
function message(event) {
 alert("Имя события:" + event.type);
}
```

Порядок срабатывания событий

На одно и то же событие может реагировать не только указанный DOM элемент, но и элементы, в которые он вложен. Это свойство часто используется при обработке большого количества вложенных однотипных элементов. В этом случае обработчик можно повесить на родительский тег и ловить все события, происходящие в потомках. После отработки событий потомков будут отработаны события на родительском теге. Этот способ называется фазой "всплытия", он действует в браузерах по умолчанию. Если какой-либо обработчик хочет остановить всплытие и не выпускать событие наружу, для объекта `events` используется метод `stopPropagation()`:

```
element.onclick = function(event) {
 event.stopPropagation()
}
```

При использовании другого способа обработки - фазы "захвата" события будут вызываться и обрабатываться в обратной последовательности: от родительского тега к потомкам.

Консорциумом W3C определена универсальная модель обработки вложенных событий, в которой сначала используется фаза "захвата", а затем "всплытия". Таким образом, разработчик веб-приложения может самостоятельно определить, в какой фазе должен выполняться тот или иной обработчик, с помощью установки параметра `<фаза>` функции `addEventListener`.

Для некоторых событий определены стандартные действия браузера, например вызов контекстного меню по нажатию правой кнопки мыши. Для отмены действия браузера по умолчанию используется метод `preventDefault()`. После этой команды можно задавать свой обработчик события:

```
element.onclick = function(event) {
 event.preventDefault();
 // код обработчика события
}
```

1.4. Обработка форм

Существует несколько способов взаимодействия пользовательского интерфейса веб-приложения (frontend) с серверной частью веб-приложения (backend). Например, с помощью параметров URL адреса (`GET` запрос);

- с помощью элементов форм;
- с помощью асинхронного запроса AJAX.

В первом случае для взаимодействия пользователя с веб-приложением используются гиперссылки, содержащие параметры URL строки, например:

```
<a
href="http://domain.ru/index.php?login=ivan&passwor
d=1234"> Авторизация</a>
```

В данном случае взаимодействие с веб-приложением осуществляется с помощью `GET` запроса по адресу `http://domain.ru/index.php` с параметрами `login=ivan` и `password=1234`. Гиперссылки используются в случае, когда все параметры имеют фиксированные значения.

Для взаимодействия пользователя с веб-приложением с помощью интерактивных элементов форм: списков, текстовых полей ввода, кнопок и др., последние размещаются или прикрепляются на форме. В этом случае может использоваться как `GET` (по умолчанию), так и `POST` метод запроса. URL адрес приложения указывается с помощью атрибута `action`.

Перед отправкой формы на сервер данные можно проверить на корректность ввода с помощью специальной функции - обработчика, связанного с кнопкой ввода. Этот способ называется валидацией содержимого формы, производимой, как правило, на стороне клиента.

Доступ к свойствам и методам элементов форм осуществляется

с помощью интерфейса DOM:

```
document.<имя_формы>.<имя_элемента>.<свойства>
```

Данные формы, введенные пользователем, отправляются на сервер с помощью метода `submit()`, применяемого к форме. Метод вызывается автоматически при нажатии кнопки типа `submit` или при нажатии клавиши `<Enter>` в текстовом поле ввода. Также метод `submit()` можно вызвать программным способом в обработчике формы. Пример валидации формы:

```
<form name="form1"
action="http://domain.ru/index.php">
  <label>Логин <input type="text"
name="login"></label>
  <label>Пароль <input type="password"
name="passwd"></label>
  <input type="button" value="Ввод" onclick="go()">
</form>
<script>
    function go() {
  if (document.form1.login.value=="") {
     alert("логин не введен");   return false; }
  if (document.form1.passwd.value=="") {
     alert("пароль не введен");   return false; }
  document.form1.submit(); // отправить данные на
сервер
    }
</script>
```

Основные свойства и методы объекта `form`:

`form.elements` – массив всех элементов формы;

`form.length` – количество элементов формы;

`form.name` – имя формы;

`form.<имя_элемента>` – обращение к элементу формы;

`form.action` – адрес URL программы обработчика на сервере;

`form.submit()` – отправка данных формы на сервер;

`form.reset()` – сброс элементов формы в исходное состояние.

Для всех элементов форм можно обратиться к родительскому объекту формы с помощью ссылки: `<имя_элемента>.form`

Объект ввода `input` содержит следующие основные свойства и методы:

`input.type` – тип элемента ввода;

`input.value` – значение элемента ввода;

`input.size` – размер элемента ввода;

`input.name` – имя элемента ввода;

`input.accept` – значение атрибута `accept` (для `<input type="file">`);

`input.checked` – значение атрибута `checked` (для `<input type="checkbox">` и `<input type="radio">`);

`input.defaultChecked` – возвращает `true`, если элемент выбран по умолчанию (для `<input type="checkbox">` и `<input type="radio">`);

`input.maxLength` – значение атрибута `maxlength` (для `<input type="text">` и `<input type="password">`);

`input.readOnly` – возвращает `true`, если содержимое поля нельзя изменить (для `<input type="text">` и `<input type="password">`);

`input.select()` – выделение текста элемента (для `<input type="text">` и `<input type="password">`).

Кроме того, к строковым свойствам элементов формы можно применять строковые функции или операции сравнения, например:

```
if (document.form1.passwd.length < 6) {
    alert("Слишком короткий пароль");  return
false;
}
if (document.form1.passwd.value !=
document.form1.passwd2.value) {
    alert("Пароли не совпадают");  return false;
}
```

Объект ввода текста `textarea` содержит следующие основные свойства и методы:

`textarea.value` – текст элемента ввода;

`textarea.name` – имя элемента ввода;

`textarea.defaultValue` – текст элемента ввода по умолчанию;

`textarea.disabled` – значение атрибута `disabled`;

`textarea.readOnly` – значение атрибута `readonly`;

`textarea.rows` – количество строк элемента (значение атрибута `rows`);

`textarea.cols` – количество позиций элемента (значение атрибута `cols`);

`textarea.select()` – выделение текста элемента.

Объект списка `select` содержит следующие основные свойства и методы:

`select.length` – количество пунктов в выпадающем списке;

`select.multiply` – значение атрибута `multiply`;

`select.name` – имя объекта списка;

`select.options` – массив всех элементов списка;

`select.selectedIndex` – позиция выбранного элемента списка (начиная с 0);

`select.add(<элемент>[,<позиция>])` – добавление элемента в выпадающий список (по умолчанию элемент добавляется в конец списка);

`select.remove(<позиция>)` – удаление элемента из выпадающего списка.

Элемент списка `option` содержит следующие основные свойства и методы:

`option.index` – текущая позиция элемента в списке (начиная с 0);

`option.selected` – значение атрибута `selected`;

`option.value` – значение элемента списка.

Пример валидации списка:

```
<form name="form1"
action="http://domain.ru/index.php">
  <label>Владение языками
    <select name="language" multiple>
 <option value="EN">английский</option>
 <option value="DE">немецкий</option>
 <option value="FR">французский</option>
  </select></label>
  <input type="button" value="Ввод" onclick="go()">
</form>
<script>
    function go() {
 var select = document.form1.language;
 var c = false; // признак выбора
 for (var i=0; i<select.length; i++) { // цикл по
всем опциям
        if (select.options[i].selected) c = true;
 }
 if (!c) {  alert("Ни один язык не выбран");
return false; }
 document.form1.submit(); // отправить данные на
сервер
    }
</script>
```

1.5. Асинхронная передача данных

Основой взаимодействия пользователя с веб-приложением является его интерфейс, который формируется на основе элементов HTML страниц. В период развития интернет-технологий в конце прошлого века основной проблемой являлось обновление части данных на странице. Для корректного отображения интерфейса всю HTML страницу приходилось перегружать целиком, нагружая при этом браузер, сервер и каналы передачи данных, что значительно замедляло работу веб-приложения. Использование технологии фреймов для этой цели себя не оправдало из-за сложности организации взаимодействия фреймовых окон и громоздкости всей конструкции. Технология встраиваемых страниц SSI (*Server Side Including*) оказалась неудобной в настройке и конфигурировании, и с развитием веб-программирования надобность в ней отпала. Одновременно с этим появилась насущная необходимость обновления отдельных элементов без перезагрузки всей HTML страницы.

Для частичного обновления информации на странице был разработан стандарт обмена данными AJAX (*Asynchronous JavaScript And Xml*). Для обмена данными с сервером с помощью AJAX используется JavaScript объект `XmlHttpRequest`, который умеет отправлять запрос и получать ответ с сервера.

Обмен данными с сервером в AJAX может быть как синхронным, так и асинхронным. При синхронном обмене обработчик будет ожидать ответа от сервера после каждого запроса. При асинхронном обмене после запроса выполнение JavaScript сценариев на стороне клиента не прерывается, а ответ от сервера вызывает событие `onreadystatechange`, которое затем перехватывается клиентским обработчиком. Синхронная обработка данных в веб-приложениях применяется крайне редко.

Основные методы объекта `XmlHttpRequest`:

`new()` – создание объекта. Используется в качестве инициализации AJAX.

`open(<метод>,<адрес>,<асинхр>[,<логин>,<пароль>])` – настройка параметров соединения с сервером. Параметр `<метод>` задает один из двух методов запроса: `GET` или `POST`. `<адрес>` представляет собой URL адрес запроса. Параметр `<асинхр>` задает способ синхронизации: `true` – асинхронный, `false` – синхронный. Дополнительные параметры `<логин>` и `<пароль>` при необходимости задают аутентификационные данные доступа к серверу. Метод не открывает соединение. Пример:

```
var xhr = new XMLHttpRequest(); // создается HTTP
запрос
xhr.open('GET', 'http://domain.ru/phones.json',
true); // асинхронный метод
```

`send([<запрос>])` – отправка запроса на сервер. Дополнительный параметр `<запрос>` является телом запроса и используется с методом `POST`. В случае использования метода `GET` параметры запроса передаются в URL строке метода `open()`. Пример:

```
var xhr = new XMLHttpRequest(); // создается HTTP
запрос
xhr.open('GET', 'http://domain.ru/phones.json',
true);
xhr.send(); // запрос отправляется на сервер
```

`abort()` – прерывание выполнения запроса.

`onreadystatechange()` – асинхронное событие, вызываемое несколько раз в ходе сеанса запроса, по числу возможных состояний. Событие связывается с обработчиком ответа. Текущее состояние запроса определяется свойством `readyState`:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState != 4) return; // запрос не
завершен
  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText); //
запрос неуспешен
  } else {
    // запрос успешен...
    alert(xhr.responseText); // вывод ответа
сервера
  }
}
```

`setRequestHeader(<имя>,<знач>)` – устанавливает заголовок запроса, где `<имя>` – наименование заголовка, `<знач>` – его значение. Отменить установленный заголовок нельзя, так же как и установить некоторые виды заголовков, контролируемых браузером. Пример:

```
xhr.setRequestHeader('Content-Type',
'application/json');
```

`getResponseHeader(<имя>)` – получает заголовок ответа, где `<имя>` – наименование заголовка (кроме заголовка `Set-Cookie`):

```
xhr.getResponseHeader('Content-Type');
```

`getAllResponseHeaders()` – получает все заголовки ответа, разделенные символами перевода строки. Заголовок возвращается в виде одной строки.

Основные свойства объекта `XmlHttpRequest`:

`readyState` – текущее состояние запроса (0 – начальное состояние, 1 – вызвана функция open, 2 – получены заголовки ответа, 3 – загружается тело ответа, 4 – запрос завершен). Обработка ответа сервера допустима только в случае завершенного запроса (`readyState=4`);

`status` – код ответа сервера (стандартные коды: 200 – запрос успешен, 302 – документ перемещен, 401 – клиент не авторизован, 403 – запрос отклонен, 404 – документ не найден, 500 – внутренняя ошибка сервера и т.д.);

`statusText` – текстовое описание статуса сервера;

`responseText` – текстовой ответ сервера (содержание запрашиваемого документа) ;

`responseXML` – ответ сервера в формате XML (если запрашивался документ в формате XML) ;

`timeout` – задает максимальную продолжительность асинхронного запроса (в миллисекундах). При превышении этого времени генерируется событие `ontimeout`, например:

```
xhr.timeout = 30000;
xhr.ontimeout = function() {
   alert('Запрос превысил 30 секунд')
}
```

Преимущества AJAX

Экономия трафика. Использование AJAX позволяет значительно сократить трафик при работе с веб-приложением, так что вместо загрузки всей страницы загружается только изменившаяся часть или только данные в формате JSON или XML.

Уменьшение нагрузки на сервер. AJAX позволяет снизить нагрузку на сервер в несколько раз за счет уменьшения трафика и числа HTTP соединений.

Улучшение отклика интерфейса. При загрузке частичных данных через AJAX пользователь сразу получает результат своих действий без обновления страницы.

Недостатки AJAX

Отсутствие интеграции с браузерами. Динамически создаваемые страницы не регистрируются браузером в истории

посещения страниц, поэтому не работает кнопка «Назад», также невозможно сохранение закладок на просматриваемые материалы. Эти проблемы решаются с помощью скриптов.

Динамически загружаемое содержимое недоступно поисковикам. Старые методы индексации содержимого и учета статистики сайтов становятся неактуальными из-за того, что поисковые машины не могут выполнять JavaScript. Эта проблема решается с помощью организации альтернативного доступа к содержимому сайта для поисковых машин.

Усложнение клиентской части веб-приложений. Перераспределяется логика обработки данных - происходит выделение и частичный перенос на сторону клиента процессов получения и форматирования данных. В целом это усложняет контроль целостности форматов и типов и приводит к усложнению и удорожанию веб-приложений.

Требуется включенный JavaScript в браузере. JavaScript может быть выключен из соображений безопасности, кроме того, AJAX страницы труднодоступны неполнофункциональным браузерам и поисковым роботам.

Риск несанкционированного использования кроссдоменных запросов. Результат работы AJAX запроса может являться JavaScript кодом. Если методы объекта XMLHttpRequest действуют только в пределах одного домена, то для тега `<script>` такого ограничения нет, что может привести к различным коллизиям.

В целом AJAX удобен для программирования интерактивных веб-приложений, административных панелей и других инструментов, которые используют динамические данные.

1.6. Форматы обмена данными

Для обмена данными между серверной и клиентской частями веб-приложения наиболее часто используются следующие форматы: XML (*eXtensible Markup Language*) и JSON (*JavaScript Object Notation*). Данные могут быть приняты от сервера в результате AJAX запроса и объекта `XmlHttpRequest`.

Документ XML представляет собой дерево, состоящее из узлов – элементов (рис. 6). Для доступа к XML узлам используются вышеописанные функции DOM, применяемые ранее к тегам HTML страниц. Одним из важных достоинств представления данных в виде XML является его универсальность. Стандарт обмена данными в формате XML используется при разработке сложных модульных веб-приложений, в стеке технологий Java, при проектировании и

использовании веб-сервисов, в серверном протоколе SOAP. Большинство языков программирования содержит встроенные библиотеки обработки XML документов, поддерживающие интерфейс DOM – парсеры.

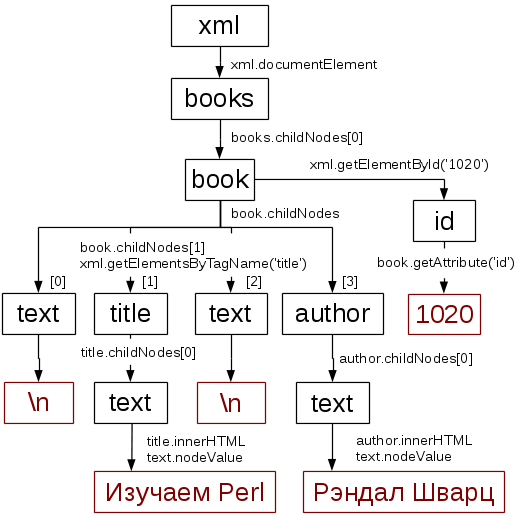

Рис. 6. Дерево XML-документа

Недостаток использования формата XML заключается в некоторой избыточности передаваемых данных (каждый XML узел представляет собой пару текстовых тегов), а также трудоемкости и ресурсоемкости разбора данных XML формата с помощью функций DOM или SAX (Simple API XML).

Пример использования XML данных

Содержание файла books.xml на сервере:

```
<?xml version="1.0"?>
<books>
  <book id="1020">
        <title>Изучаем Perl</title>
        <author>Рэндал Шварц</author>
  </book>
  <book id="1021">
        <title>HTML, скрипты и стили</title>
        <author>Дунаев В.</author>
  </book>
</books>
```

Обработка подгружаемого списка из XML файла:

```
<ol id="books" onclick="select()">Список книг</ol>
<script>
function select() {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://domain.ru/books.xml',
false);
    xhr.send(); // выполнение запроса
    if (xhr.status != "200") {      // обработка
ошибки
 alert("Ошибка сервера"); return false;
    }
    var xml = xhr.responseXML;
    var ol = document.getElementById("books");
    var book = Array();
    for(var i=0; i < xml.childNodes.length; i++) {
 var title  =
xml.getElementsByTagName("title")[i].innerHTML;
 var author =
xml.getElementsByTagName("author")[i].innerHTML;
 book[i] = document.createElement("li");
 book[i].innerHTML = author.italics() + title;
 ol.appendChild(book[i]);
    }
}
</script>
```

В результате после нажатия на текстовой элемент мы получаем вывод списка книг из заданного XML файла.

Использование формата JSON

Альтернативой использования стандарта XML является представление данных в формате JSON. Это удобный краткий текстовой формат описания JavaScript объектов. Данные, передаваемые от серверной части веб-приложения, конвертируются в объектные переменные JavaScript и могут непосредственно использоваться в программных обработчиках на стороне клиента.

Данные в формате JSON (в соответствии с RFC 4627) представляют собой набор терм, содержащих:

- JavaScript объекты { ... };
- массивы [...];
- строки, заключенные в двойные кавычки;
- числа;
- логические значения true/false;

• пустое значение null.

Основой представления JSON данных являются JavaScript объекты, содержащие свойства и их значения. Содержание файла books.json на сервере:

```
{
    "book": [
  {      "id": 1020,
         "title": "Изучаем Perl",
         "author": "Рэндал Шварц"
  },
  {      "id": 1021,
         "title": "HTML, скрипты и стили",
         "author": "Дунаев В."
  }
    ]
}
```

В отличие от объектов JavaScript, все атрибуты JSON данных должны представлять собой текстовые строки, заключенные в кавычки. Апострофы не допускаются. Такое представление короче и проще, чем XML, и потому завоевало большую популярность у разработчиков. Для чтения и разбора данных JSON в языке JavaScript используется глобальный объект `JSON`.

Основные методы объекта JSON:

`JSON.parse(<строка>[,<функция>(ключ,значение)])` – преобразование текстовой строки JSON в объект JavaScript. Дополнительный параметр `<функция>(ключ, значение)` вызывает функцию для каждого свойства объекта, которая должна вернуть измененное значение.

`JSON.stringify(<объект>[,<массив>])` – преобразование (сериализация) объекта JavaScript в текстовую строку формата JSON. Дополнительный параметр `<массив>` содержит массив заданных свойств для сериализации объекта.

Пример использования данных JSON:

```
<ol id="books" onclick="select()">Список книг</ol>
<script>
function select() {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://domain.ru/books.json',
false);
    xhr.send(); // выполнение запроса
    if (xhr.status != "200") {      // обработка
ошибки
```

```
 alert("Ошибка сервера"); return false;
    }
    var json_str = xhr.responseText;
    var ol = document.getElementById("books");
    var obj = JSON.parse(json_str,
function(key,val) {
        if (key=="author") val = val.italics();
        return val;
 }
    );
    var book = Array();
    for(var i=0; i < obj.book.length; i++) {
 var title  = obj.book[i].title;
 var author = obj.book[i].author;
 book[i] = document.createElement("li");
 book[i].innerHTML = author + title;
 ol.appendChild(book[i]);
    }
}
</script>
```

После нажатия на текстовой элемент мы, как и в предыдущем примере, получаем вывод списка книг из заданного файла формата JSON.

2. Библиотека JQuery

Одной из самых популярных JavaScript библиотек, использующихся в разработке веб-приложений, является библиотека JQuery. Если CSS отделяет визуализацию от структуры HTML, то JQuery отделяет поведение от структуры HTML документа. Эта библиотека позволяет значительно упростить и ускорить написание JavaScript кода, сделать его более прозрачным и понятным. jQuery позволяет создавать анимацию, обработчики событий, значительно облегчает выбор элементов в DOM и создание AJAX запросов. Также эта библиотека является кроссбраузерной, что позволяет не заботиться о кроссбраузерной совместимости JavaScript кода.

Для использования библиотеки JQuery в своих проектах необходимо скачать ее с официального сайта https://code.jquery.com и добавить ее на HTML страницу веб-приложения. Существует два варианта библиотеки JQuery: для разработчиков (uncompressed) и для готовых приложений (minified). Последний вариант имеет минимальный объем кода за счет удаления комментариев и пробелов из исходников и отличается от первого дополнительным расширением .min в имени файла.

Для добавления библиотеки JQuery на HTML страницу веб-приложения используется вставка скрипта в заголовок страницы в виде отдельного файла, например (для предварительной загрузки и локального использования):

```
<script src="jquery-2.2.4.min.js"></script>
```

или (для использования ссылки Google):

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/2
.2.4/jquery.min.js">
</script>
```

В приведенном примере используется сжатая версия 2.2.4 библиотеки JQuery. Версии периодически обновляются, поэтому за этим нужно следить. На официальном сайте также содержится подробное описание функций API и возможностей, предоставляемых библиотекой.

Один из наиболее частых вариантов использования библиотеки JQuery, позволяющий избежать преждевременное выполнение кода до полной загрузки страницы, с использованием оборачивающей безымянной функции:

```
$(document).ready(function() {
  // обработка данных
});
```

Вся работа с JQuery производится с помощью функции `$()`, что является синонимом `JQuery`.

Работу с jQuery можно разделить на 2 типа:

• Получение jQuery объекта с помощью функции `$()`. Например, передав в нее CSS-селектор, можно получить jQuery объект всех элементов HTML, попадающих под критерий, и далее работать с ними с помощью различных методов jQuery объекта. В случае если метод не должен возвращать какого-либо значения, он возвращает ссылку на jQuery объект, что позволяет вести цепочку вызовов методов.

• Вызов глобальных методов объекта `$`, например итераторов для массива.

Таким образом, стандартный синтаксис выполнения команд JQuery:

```
$(<селектор>).<метод>(<параметры>)
```

Например, вызов `$()` функции со строкой селектора `CSS`

возвращает объект jQuery, содержащий некоторое количество элементов HTML страницы. Эти элементы затем можно обработать методами jQuery:

```
$(document).ready(function() {
  $("div.test").add("p.quote").addClass("blue").slid
eDown("slow");
});
```

В данном примере находятся все элементы div с классом test, а также все элементы p с классом quote. Затем к ним добавляется класс blue и применяется метод slideDown – визуальный слайдер с параметром slow. В результате возвращается ссылка на исходный объект $("div.test").

Все методы в JQuery делятся на следующие группы:

- методы для манипулирования DOM;
- методы для оформления элементов;
- методы для создания AJAX запросов;
- методы для создания эффектов;
- методы для привязки обработчиков событий.

Селекторы

С помощью CSS селекторов выбираются элементы на странице для последующего применения к ним определенных действий. Параметры селекторов в целом соответствуют синтаксису стандарта CSS, хотя и имеют некоторые отличия. Основные виды селекторов, поддерживаемые JQuery:

$("*") – выделить все элементы на странице;

$("p") – выделить все абзацы в тексте;

$(".par") – выделить все элементы с классом class="par";

$("#par") – выделить элемент с идентификатором id="par";

$(this) – выделить текущий элемент;

$("p.par") – выделить все абзацы с классом class="par";

$("p#par") – выделить абзац с идентификатором id="par";

$(".head,.foot") – выделить все элементы с классами class="head" и class="foot";

$("[src]") – выделить все элементы, имеющие атрибут src;

$("[src='1.jpg']") – выделить все элементы с атрибутом src="1.jpg";

$("[src!='1.jpg']") – выделить все элементы с атрибутом src, не равным '1.jpg';

`$("[src^='1.']")` – выделить все элементы с атрибутом `src` и началом `'1.'`;

`$("[src$='.jpg']")` – выделить все элементы с атрибутом `src` и окончанием `'.jpg'`;

`$("[src*='jpg']")` – выделить все элементы с атрибутом `src`, содержащим `'jpg'`;

`$("ol#list li")` – выделить все элементы `li` внутри нумерованного списка `ol` с идентификатором `id="list"`;

`$("p > div")` – выделить все элементы `div` внутри родительского абзаца `p`;

`$(":input")` – выделить все элементы `input`;

`$(":button")` – выделить все элементы `input` с атрибутом `type="button"`;

`$(":text")` – выделить все элементы `input` с атрибутом `type="text"`;

`$(":password")` – выделить все элементы `input` с атрибутом `type="password"`;

`$(":radio")` – выделить все элементы `input` с атрибутом `type="radio"`;

`$(":checkbox")` – выделить все элементы `input` с атрибутом `type="checkbox"`;

`$(":file")` – выделить все элементы `input` с атрибутом `type="file"`;

`$("div:first")` – выделить первый элемент `div`;

`$("div:last")` – выделить последний элемент `div`;

`$("li:even")` – выделить все четные элементы списка `li`;

`$("li:odd")` – выделить все нечетные элементы списка `li`;

`$("li:eq(2)")` – выделить третий элемент списка `li` (нумерация с 0);

`$("li:lt(2)")` – выделить первый и второй элементы списка `li` (индекс меньше 2);

`$("li:gt(1)")` – выделить все элементы списка `li`, начиная с 3 (индекс больше 1);

`$(":header")` – выделить все заголовки `h1` – `h6`;

`$(":animated")` – выделить все анимированные элементы;

`$(":contains('hello')")` – выделить все элементы, содержащие строку `'hello'`;

`$(":empty")` – выделить все элементы, не имеющие потомков;

`$(":hidden")` – выделить все скрытые элементы;

`$(":visible")` – выделить все видимые элементы.

Обработчики событий

После выбора элементов можно задать обработчики событий и привязать их к данным элементам. Общий синтаксис обработчика:

```
$(<селектор>).<событие>(function() { ...код
обработчика... } );
```

Например, для изменения текста раздела `div` по нажатию на кнопку `button` определим следующий обработчик:

```
$(document).ready(function() {
  $(":button").click(function() {
$("div#par").html("новый текст"); });
});
```

Основные методы событий:

`click([<функция>])` – устанавливает обработчик клика мыши (или запускает его при опущенном параметре <функция>);

`dblclick([<функция>])` – устанавливает/запускает обработчик двойного клика мыши;

`mouseenter([<функция>])` – устанавливает/запускает обработчик появления курсора мыши в области элемента (не обладает свойством всплытия);

`mouseleave([<функция>])` – устанавливает/запускает обработчик выхода курсора мыши из области элемента (не обладает свойством всплытия);

`hover(<функция1>[,<функция2>])` – устанавливает обработчики двух событий `mouseenter` и `mouseleave`. Если задан один параметр `<функция1>`, то он является обработчиком двух событий одновременно;

`mousedown([<функция>])` – устанавливает/запускает обработчик нажатия кнопки мыши;

`mouseup([<функция>])` – устанавливает/запускает обработчик отпускания кнопки мыши;

`mousemove([<функция>])` – устанавливает/запускает обработчик перемещения мыши;

`toggle(<функция1>,<функция2>[,<функция3>,...])` – устанавливает переключение между двумя и более обработчиками, вызываемыми по нажатию кнопки мыши;

`trigger(<событие>)` – вызывает заданное событие по его имени. Пример:

```
$("form").trigger("submit"); // отправка данных
формы на сервер;
```

keydown([<функция(объект)>]) – устанавливает/запускает обработчик нажатия клавиши. Код клавиши передается в свойстве объект.which;

keyup([<функция(объект)>]) – устанавливает/запускает обработчик отпускания клавиши. Код клавиши передается в свойстве объект.which;

keypress([<функция(объект)>]) – устанавливает/запускает обработчик ввода символа с клавиатуры. Код символа передается в свойстве объект.which;

focus([<функция>]) – устанавливает/запускает обработчик получения фокуса элемента;

blur([<функция>]) – устанавливает/запускает обработчик потери фокуса элемента;

change([<функция>]) – устанавливает/запускает обработчик изменения элемента. Событие активируется, когда элемент ввода изменяет свое значение после получения фокуса. Пример:

```
$(":password").change( function() {
 if ($(this).val().length < 6) {
        alert("Пароль меньше 6 знаков!");
 }
});
```

resize([<функция>]) – устанавливает/запускает обработчик изменения размера элемента. Пример:

```
$(window).resize( function() {
 alert("размер окна изменен");
});
```

scroll([<функция>]) – устанавливает/запускает обработчик прокрутки области просмотра документа;

select([<функция>]) – устанавливает/запускает обработчик выделения текста в текстовом поле. Пример:

```
$(":input").select( function () {
 alert("Текст был выделен");
});
```

submit([<функция>]) – устанавливает/запускает обработчик отправки формы на сервер. Пример:

```
$("form").submit( function () {
 if ($(":password").val().length < 6) {
        alert("Пароль слишком короткий"); return
false;
```

```
 } return true;
});
```

Содержание и стилизация элементов

`text([<текст>])` – получает/устанавливает текстовое значение элемента;

`html([<разметка>])` – получает/устанавливает HTML значение элемента;

`append(<текст>)` – добавляет текст в конец элемента. Пример:

```
$("div").append("текст в конец элемента");
```

`prepend(<текст>)` – добавляет текст в начало элемента;

`after(<текст>)` – добавляет текст после элемента;

`before(<текст>)` – добавляет текст до элемента;

`remove([<класс>])` – удаляет выбранные элементы вместе с потомками. Если указан класс, то удаляются только элементы с указанным классом. Пример:

```
$("p").remove(".test"); // удаление абзацев с
классом test
```

`empty()` – удаляет содержание и потомки выбранного элемента;

`attr(<атрибут>[,<значение>])` – получает/устанавливает значение атрибута выбранных элементов. Если элемент не имеет атрибута, возвращается значение `undefined`. Пример:

```
var title = $("em").attr("title"); //получить
значение атрибута title тега em
$("div").text(title);// установить содержимое тега
div
```

`removeAttr(<атрибут>)` – удаляет заданный атрибут выбранных элементов;

`addClass(<класс>)` – добавляет новый класс к выбранным элементам. Пример:

```
$("em").addClass("red"); // установить класс red
для тега em
```

`hasClass(<класс>)` – проверяет наличие класса для выбранных элементов. Если указанный класс установлен, то возвращается `true`;

`removeClass([<класс>])` – удаляет заданный класс для

выбранных элементов. Если параметр <класс> не задан, то для выбранных элементов удаляются все классы;

toggleClass(<класс>[,<усл>]) – добавляет или удаляет (переключает) класс для выбранных элементов. Необязательный логический параметр <усл> может использоваться в качестве дополнительного условия: true – добавить класс, false – удалить класс. Пример:

```
$("button").click(function() { // добавляет или
удаляет класс red
  $("div").toggleClass("red");
 } ); //
```

css(<свойство>[,<знач>]) – получить/установить CSS свойство для выбранных элементов. Если устанавливается несколько свойств, то параметры могут быть переданы в формате JSON. Пример:

```
$("p").css( { "color": "blue", "background-color",
"yellow" } );
// установить синий цвет шрифта и желтый фон для
абзацев
```

Визуальные эффекты

hide([<скорость>]) – скрыть элемент. Дополнительный параметр <скорость> может принимать значения slow, fast или число в миллисекундах. Пример:

```
$("#hide").click(function(){
    $("p").hide(); // скрыть все вложенные абзацы
});
```

show([<скорость>]) – показать элемент. Метод обратный hide();

toggle([<скорость>]) – переключить видимость элемента. Аналогично применению функций hide и show;

fadeIn([<скорость>]) – проявление элемента. В отличие от метода show() производится анимация свойства прозрачности (opacity);

fadeOut([<скорость>]) – исчезновение элемента. Производится анимация свойства прозрачности (opacity);

fadeTo([<скорость>],[<прозрачность>]) – проявление/исчезновение элемента. Производится анимация свойства прозрачности (opacity) до заданного значения;

fadeToggle([<скорость>]) – поочередное проявление или

исчезновение элемента;

`slideUp([<скорость>])` – плавное скрытие элемента по высоте;

`slideDown([<скорость>])` – плавное разворачивание элемента по высоте;

`slideToggle ([<скорость>])` – плавное разворачивание/сворачивание элемента по высоте. С помощью этого метода можно организовать выпадающее меню:

```
$("#menu").click(function() {
$("#list").slideToggle(500)} );
```

`animate(<свойства>,[<скорость>])` – применение пользовательской анимации. Список CSS свойств задается в виде объекта JavaScript. В отличие от метода `css()`, значения этих свойств могут быть указаны в виде констант `hide`, `show`, `toggle` или относительных единиц. Например:

```
$('div').animate(  // убирать прозрачность и
уменьшить высоту на 50px
     {  opacity: "hide",   height: "-=50"  }, 5000
);
```

Асинхронные запросы

`load(<URL>)` – загрузка содержимого с помощью AJAX запроса. Параметр `<URL>` Например:

```
$("div").click(function(){
    $(this).load('example.html'); // загрузить файл
в текущий блок
});
```

`get(<URL>[,<obj>])` – выполнение GET запроса с помощью AJAX. Дополнительный параметр `obj` содержит параметры GET запроса:

```
$(this).get('example.php', { id: 1}); // параметр
id=1
```

`post(<URL>[,<obj>])` – выполнение POST запроса с помощью AJAX. Дополнительный параметр `obj` содержит параметры POST запроса;

`ajax(<obj>)` – выполнение запроса с помощью AJAX (универсальный метод). Параметр `obj` может содержать следующие поля:

• async – асинхронность запроса, по умолчанию true;

• contentType – тип возвращаемого содержимого, по умолчанию application/x-www-form-urlencoded;

• data – передаваемые данные (строка или объект);

• dataType – тип возвращаемых данных (xml, html, script, json или text);

• type – тип запроса GET или POST;

• url – URL запрашиваемой страницы;

• username – логин для подключения;

• password – пароль для подключения;

• beforeSend – функция-обработчик перед началом запроса;

• error – функция-обработчик ошибки;

• success – функция-обработчик успешного запроса;

• complete – функция-обработчик после завершения запроса.

Пример использования метода AJAX и вывод результата:

```
$(this).ajax( {    url: 'example.json',
    dataType: 'json',
    success( function(data, status) {
        alert('Данные загружены в javascript объект
data');
    });
});
```

getJSON(<URL>[,<obj>]) – выполнение запроса на получение объекта JSON с помощью AJAX. Дополнительный параметр obj содержит параметры запроса.

Сервисные функции

each(<obj>, <func(i[,val])>) – итератор, вызывающий заданную функцию для каждого элемента массива с индексом i или заданным свойством объекта. Пример:

```
var obj = { 'one': 1, 'two': 2 };
$('div').each( obj, function(key, val) {
      alert(key + ': ' + val);
});
```

grep(<obj>, <func(elem[,i])>) – фильтрация элементов массива по заданным критериям. Пример:

```
var a = [1,2,5,9,16,25,8,9,6,3];
a = $.grep(a, function(elem, i) {
      return (elem == i*i); // a = [9,16,16]
});
```

`extend(<res>,<obj1>[,...<objN>])` – объединение двух и более объектов в один результирующий `<res>`. Пример:

```
var a = { 'one': 1 }; var b = { 'two': 2 };
var c = $.extend({}, a, b); // c = { 'one': 1,
'two': 2 };
```

`map(<obj>, <func(elem[,i])>)` – замена каждого элемента массива в соответствии с заданной функцией. Пример:

```
var a = ['a', 'b', 'c'];
a = $.map(a, function(elem) {
      return (elem.toUpperCase()); // a = ['A',
'B', 'C']
});
```

`parseHTML(str)` – преобразование HTML строки в JavaScript объект:

```
html = $.parseHTML('Hello, <br>world!');
```

`parseJSON(str)` – преобразование строки JSON в JavaScript объект:

```
json = $.parseJSON('{ id: 1, name: test }');
```

`parseXML(str)` – преобразование строки XML в JavaScript объект.

Библиотека jQuery в настоящее время активно развивается, в ней появляются новые функции и возможности. Эта библиотека лежит в основе многих проектов и более мощных JavaScript фреймворков, таких как AngularJS, Bootstrap и др.

Библиографический список

1. Дронов В.А. HTML 5, CSS 3 и Web 2.0. Разработка современных Web-сайтов. – СПб.: БХВ-Петербург, 2011. – 416 с.

2. Дунаев В.В. HTML, скрипты и стили. – 3-е изд. – СПб.: БХВ-Петербург, 2011. – 816 с.

3. Прохоренок Н., Дронов В.А. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера. – СПб.: БХВ-Петербург, 2010. – 912 с.

4. Интерактивный учебник по JavaScript. [Электронный ресурс]. URL: https://learn.javascript.ru/ (дата обращения: 01.03.2017).

Оглавление