ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данное методическое пособие предназначено для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника» и 09.03.02 «Информационные системы и технологии» изучающих курс «Технология создания web-сервисов и web-приложений». Основные задачи курса:

Знать:

- возможности современных инструментальных программных средств для решения задач создания web-сервисов и web-приложений;
- основные современные библиотеки для автоматизации разработки web-ресурсов;
- особенности использования фреймворков для автоматизированной разработки web-ресурсов;
- технические методы разработки структуры сайта с использованием инструментальных средств;
- подходы к организации хранения данных web-ресурсов;
- приемы использования баз данных при организации работы web-ресурсов;
- существующие технологии организации работы клиент-серверных приложений.

Уметь:

- производить установку и настройку программного инструментария для решения задач создания web-ресурсов;
- осуществлять выбор необходимых программных средств для решения прикладных задач по созданию web-ресурсов;
- разрабатывать структуру хранилищ данных (баз данных) для работы web-ресурсов;
- осуществлять установку и настройку клиент-серверной модели для функционирования web-ресурсов;
- применять полученные знания для разработки концепции и структуры данных web-ресурсов в соответствии с предъявляемыми требованиями.

Владеть:

- методами настройки программного обеспечения для решения требуемых задач в рамках разработки web-сервисов и web-приложений;
- навыками работы с программными средствами в рамках разработки web-сервисов и web-приложений

- методиками создания и оптимизации аппаратно-программных компонентов и баз данных в рамках разработки web-сервисов и web-приложений приемами использования различных программных решений в рамках создания web-ресурсов.

Любой сайт состоит из пользовательской и серверной частей. На веб-странице отображается текст, кнопки, панели, изображения и видео. Существует возможность перемещаться по сайту, свободно изучать контент. По сути, все что видит пользователь — frontend сайта: визуализация, интерактивность и понятность интерфейса. При разработке современных веб-приложений уделяется большая роль вопросам дизайна и usability.

За логику, работоспособность и правильное функционирование сайта отвечает серверная часть, которая скрыта от пользователя. Она называется backend сайта, а взаимодействие с сервером, на котором расположена логика сайта осуществляется посредствам запросов по протоколу http и передачей данных.

Для создания полноценного веб-приложения необходимо использовать достаточно большой набор различных программных решений, включающий в себя языки разметки (HTML), каскадные таблицы стилей (CSS), серверные скриптовые языки (PHP, Python), базы данных (MySql), клиентские скриптовые языки (JavaScript), а также готовые библиотеки, что в значительной степени упрощает и автоматизирует процесс разработки (рис 1).

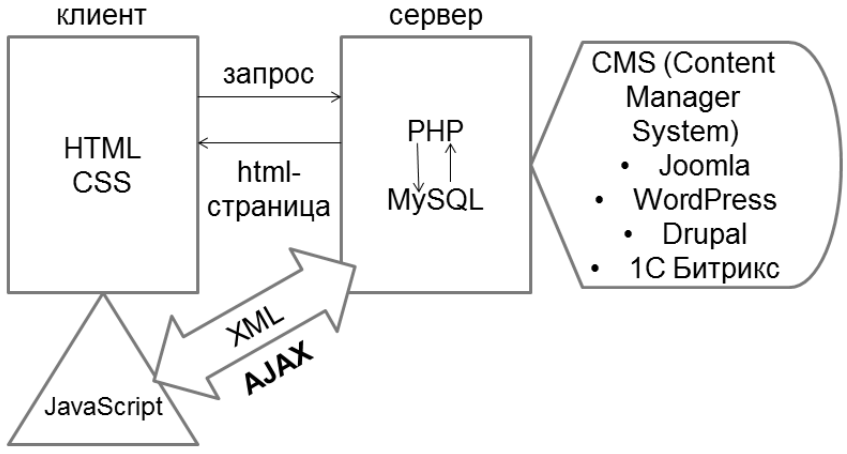


Рисунок 1. Организация построения современных веб-сервисов и веб-приложений

Вопросы разработки базовой структуры сайта на основе HTML, CSS, PHP и баз данных MySQL рассмотрены в учебно-методическом пособии «Разработка

web-сервисов с использованием HTML, CSS, PHP и MYSQL» издательства РТУ МИРЭА – 2019, авторы – Е.В. Кашкин и И.И. Антонова.

В рамках данного курса рассмотрены базовые принципы разработки клиентской части веб-ресурсов средствами JavaScript с применением библиотеки jQuery.

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVASCRIPT

JavaScript - предназначен для написания сценариев для активных HTML-страниц. Язык JavaScript не имеет никакого отношения к языку Java. Java разработан фирмой SUN. JavaScript - фирмой Netscape Communication Corporation. Первоначальное название - LiveScript. После завоевания языком Java всемирной известности LiveScript из коммерческих соображений переименовали в JavaScript.

JavaScript не предназначен для создания автономных приложений. Скрипт на JavaScript встраивается непосредственно в исходный текст HTML-документа и интерпретируется браузером по мере загрузки этого документа. С помощью JavaScript можно динамически изменять текст загружаемого HTML-документа и реагировать на события, связанные с действиями посетителя или изменениями состояния документа или окна.

Важная особенность JavaScript - объектная ориентированность. Программисту доступны многочисленные объекты, такие, как документы, гиперссылки, формы, фреймы и т.д. Объекты характеризуются описательной информацией (свойствами) и возможными действиями (методами).

Для добавления скрипта JavaScript на страницу необходимо создать в HTML документе блок SCRIPT.

Пример:

```
!DOCTYPE HTML>
<html>
  <head>
    <script language="javascript">
      //Здесь располагается скрипт javascript
    </script>
  </head>
  <body>
  </body>
</html>
```

Типы данных JavaScript

Имя переменной не должно совпадать с зарезервированными ключевыми словами JavaScript: abstract, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, extends, false, final, finally, float, for, function, goto, if, implements, import, in, instanseof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, super, sinchronized, switch, this, throw, throws, transient, true, try, typeof, var, void, wich.

Все переменные в JavaScript объявляются с помощью ключевого слова var. При объявлении тип переменной не указывается. Этот тип присваивается переменной только тогда, когда ей присваивается какое-либо значение.

Пример:

```
var a;
var a=13;
var b="Строка";
```

Типы данных, поддерживаемых языком:

- **number** для любых чисел: целочисленных или чисел с плавающей точкой, целочисленные значения ограничены диапазоном ±253;
- **bigint** для целых чисел произвольной длины;
- **string** для строк. Строка может содержать один или больше символов, нет отдельного символьного типа;
- **boolean** для true/false;
- **null** для неизвестных значений – отдельный тип, имеющий одно значение null;
- **undefined** для неприсвоенных значений – отдельный тип, имеющий одно значение undefined;
- **object** для более сложных структур данных;
- **symbol** для уникальных идентификаторов.

Оператор typeof возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку. У него есть два синтаксиса:

- Синтаксис оператора: typeof x
- Синтаксис функции: typeof(x)

Другими словами, он работает со скобками или без скобок. Результат одинаковый.

Вызов typeof x возвращает строку с именем типа.

Пример:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Операторы

PHP поддерживает следующие операции над числовыми данными

- + сложение
- - вычитание
- * умножение
- / деление
- ++ инкремент
- -- декремент

Для строковых переменных возможна операция конкатенации – «склеивания» строк. Она реализуется через оператор +.

Пример:

```
<script language="javascript">
      var a="Привет";
      var b="Вася";
      var c=a+" "+b;// Строка "Привет Вася"
</script>
```

Вывод-вывод через диалоговые окна

Самым простым способом обеспечить ввод/вывод данных в скрипте, написанном на JavaScript является использование диалоговых окон. Существует 3 типа таких окон: alert, confirm и prompt (рис. 2)

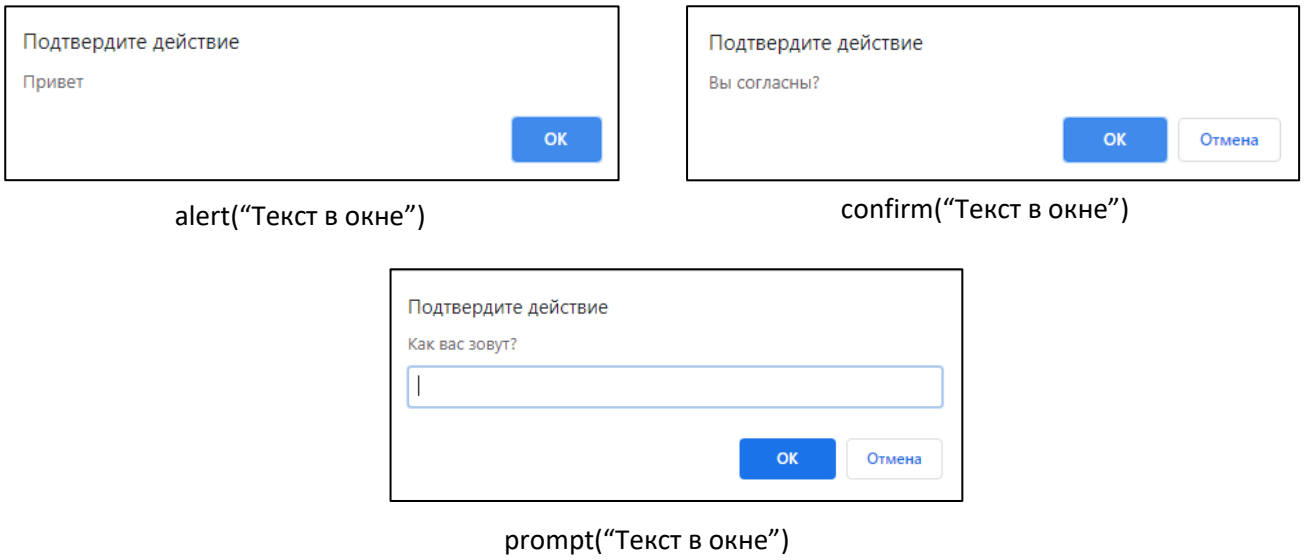


Рисунок 2. Диалоговые окна alert, prompt, confirm

В качестве возвращаемого значения confirm использует true/false (в зависимости какая кнопка была нажата: да или нет), prompt – возвращает введённую строку.

Управляющие конструкции. Ветвление

Базовая конструкция «Ветвление» подразумевает выбор одного из двух вариантов – истинного и ложного (рис. 3).

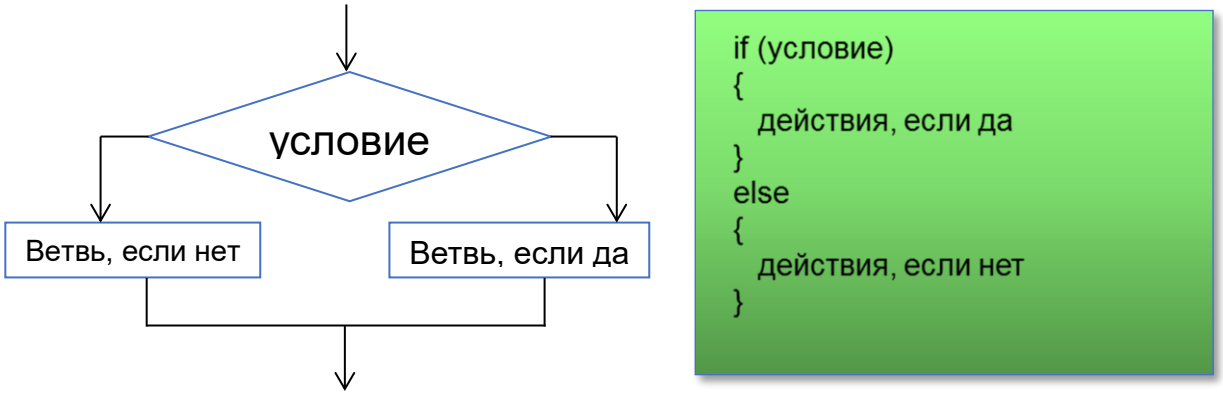


Рисунок 3. Блок-схема и синтаксис ветвления

При этом блок else не является обязательным, т.е. его можно не прописывать, если нет в этом необходимости.

Для построения условия необходимо применять условные операторы.

- больше
- < меньше
- == равно
- != не равно
- >= больше либо равно
- <= меньше либо равно

При этом возникает возможность создавать простые условия, например $a>3. Однако, часто возникает необходимость составлять более сложные условия, включающие в себя несколько простых условий. Например, проверить правильность введенного логина и пароля. При этом два этих отдельных условия должны выполняться как одно. В этом случае необходимо использовать логические операторы – &&(И), ||(ИЛИ).

Пример:

```
var log="Вася";
var pass="123";
if ((log=="Вася") && (pass=="123"))
{
    alert(“Привет, Вася”);
}
else
{
   alert(“Мы не знакомы!”);
}
```

Оператор SWITCH

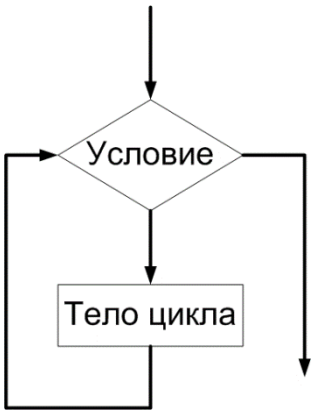
В случае необходимости множественного выбора среди заранее известного набора значений более удобным является оператор SWITCH, предполагающий набор блоков CASE для выполнения действия, если входная переменная получает конкретное значение.

Пример:

```
switch (day)
    {
     case 1: //выполнение при day ==1;
     break; //останов
     case 2: //выполнение при $day ==2;
     break; //останов
     case 3: //выполнение при $day ==3;
     break; //останов
     default: //ничего из предыдущего не  подходит;
     break; //останов
    }
```

Цикл

Данная конструкция позволяет выполнять однотипную операцию некоторое количество раз (пример – заполнение строк таблицы).



Обязательные элементы цикла:

1. Начало цикла (до входа в цикл)
2. Шаг (в теле цикла)
3. Конец цикла (условие)

Рисунок 4. Блок-схема и синтаксис цикла

Порядок работы с циклом:

1. Инициализация "счетчика";
2. Проверка условия;
3. Выполнение инструкций;
4. Изменение "счетчика".

Существуют 2 базовых оператора цикла WHILE и FOR.

Цикл WHILE

Пример:

```
while(условие)
   {
         Действия внутри цикла
        (тело цикла)
   }
```

Действия внутри тела цикла выполняются до тех пор, пока условие истинно.

Пример:

```
var i=1; //Начальное значение
while(i<11) //Условие (Конечное значение)
{
    alert (i);
    i=++; //Шаг
}
```

Результат – вывод чисел от 1 до 10 через диалоговые окна

Цикл FOR

Позволяет указывать начальное значение, условие и шаг непосредственно внутри цикла.

Пример:

```
for(начальное значение ; условие ; шаг)
   {
         Действия внутри цикла
         (тело цикла)
   }
```

При этом блоки начальное значение, условие и шаг можно не заполнять, но в обязательном порядке они должны быть разграничены точкой с запятой.

Пример:

```
for(var i=1;i<11;i++)
{
        alert (i);
}
```

Результат – вывод чисел от 1 до 10 через диалоговые окна

Прерывание цикла. Оператор BREAK

Обычно цикл завершается при вычислении условия в false. Но существует возможность выйти из цикла в любой момент с помощью специальной директивы break в случае наступления определенного события.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выводит через всплывающее окно.

Пример:

```
var sum = 0;

while (true) {

  var value = +prompt("Введите число", "");

  if (!value) break; // (*)

  sum += value;

  }
alert( 'Сумма: ' + sum );
```

Переход к следующей итерации. Оператор CONTINUE

Директива continue – «облегчённая версия» break. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если необходимо пропустить итерацию при каком-либо условии.

Например, цикл использует continue, чтобы выводить только нечётные значения.

Пример:

```
for (var i = 0; i < 10; i++)
{
      // если true, пропустить оставшуюся часть тела цикла
       if (i % 2 == 0) continue;
       alert(i); // 1, затем 3, 5, 7, 9
}
```

Функции

Часто при написании скриптов возникает необходимость повторять одно и то же действие во многих частях программы. Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы. Примеры встроенных функций – это alert(), prompt() и confirm().

Объявление функции

Для создания функций неоходимо использовать объявление функции. Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

Пример:

```
function имя(параметры)
{
  ...тело...
}
```

Вызов функции в программе происходит через вызов по имени функции.

Пример:

```
function showMessage() {
    alert( 'Всем привет!' );
  }

  showMessage();
  showMessage();
```

Параметры функции

Для того, чтобы передать внутрь функции любую информацию, используются параметры (также называемые аргументы функции). В нижеприведённом примере функции передаются два параметра: from и text.

Пример:

```
function showMessage(from, text) { // аргументы: from, text
  alert(from + ': ' + text);
}

showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

Когда функция вызывается в строках (*) и (**), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.

Если параметр не указан, то его значением становится undefined.

Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом. Это не приведёт к ошибке. Такой вызов выведет "Аня: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined.

Если необходимо задать параметру text значение по умолчанию, следует указать его после =

Пример:

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}

showMessage("Аня"); // Аня: текст не добавлен
```

Функция может вернуть результат, который будет передан в вызвавший её код. Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше).

Вызовов return может быть несколько, например:

Пример:

```
function checkAge(age)
 {
  if (age > 18) {
    return true;
  }
 else
 {
    return confirm('А родители разрешили?');
  }
}

var age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции.

Объектно-ориентированный подход JavaScript

Объектно-ориентированное программирование (ООП) — это шаблон проектирования программного обеспечения, который позволяет решать задачи с точки зрения объектов и их взаимодействий. ООП обычно реализуется с помощью классов или прототипов. Большинство объектно-ориентированных языков (Java, C++, Ruby, Python и др.) используют принцип на основе классов. JavaScript реализует ООП через прототипирование.

Принцип ООП заключается в том, чтобы составлять систему из объектов, решающих простые задачи, которые вместе составляют сложную программу. Объект состоит из приватных изменяемых состояний и функций (методов), которые работают с этими состояниями. У объектов есть определение себя (self, this) и поведение, наследуемое от чертежа, т.е. класса (классовое наследование) или других объектов (прототипное наследование).

Наследование — способ сказать, что эти объекты похожи на другие за исключением некоторых деталей. Наследование позволяет ускорить разработку за счёт повторного использования кода.

В классовом ООП классы являются чертежами для объектов. Объекты (или экземпляры) создаются на основе классов. Существует конструктор, который используется для создания экземпляра класса с заданными свойствами.

Пример:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
  getFullName() {
    return this.firstName + ' ' + this.lastName
  }
}
```

Здесь при помощи ключевого слова **class** создается класс **Person** со свойствами **firstName** и **lastName**, которые хранятся в **this**. Значения свойств задаются в конструкторе, а доступ к ним осуществляется в методе **getFullName()**.

Для создания экземпляра класса **Person** с именем **person** используется ключевое слова **new**.

Пример:

```
var person = new Person('Иван', 'Иванов')
person.getFullName() //> "Иван Иванов"
// Также существует возможность получить доступ напрямую
person.firstName //> "Иван"
person.lastName //> "Иванов"
```

В прототипном подходе классы не используются совсем. Вместо этого объекты создаются из других объектов. Процесс начинается с обобщённого объекта — прототипа. Прототип можно использовать для создания других объектов путём его клонирования или расширять его разными функциями.

Числа, строки, логические переменные (true и false), а также значения null и undefined в JavaScript относятся к простым типам данных. Всё остальное — объекты. Числа, строки и логические переменные похожи на объекты тем, что имеют методы, но в отличие от объектов они неизменны. Объекты в JavaScript имеют изменяемые ключевые коллекции. В JavaScript объектами являются массивы, функции, регулярные выражения, и, конечно, объекты также являются объектами.

По сути, JavaScript — прототипно-ориентированный язык.

Пример:

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

В данном примере создается объект **new** - прототип объекта **Date()**, содержащий в себе все свойства и методы **Date()**.

Массивы

Массив - структура данных, хранящая набор значений, идентифицируемых по индексу или набору индексов, принимающих целые значения из некоторого заданного непрерывного диапазона (рис. 5).

Рисунок 5. Массив

Существует два варианта синтаксиса для создания пустого массива.

Два способа создания массива:

```
var arr = new Array();
var arr = [];
```

Обращение к элементам массива происходит через указание индекса необходимого элемента.

Пример:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива
```

В JavaScript массив – это объект, а следовательно, у него есть встроенные методы (функции) и свойства (переменные). Одним из таких свойств является встроенное значение – количество элементов массива – length.

Пример:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits.length ); // 3
```

Методы pop/push, shift/unshift

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции. На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране. Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

- push добавляет элемент в конец.
- pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху.

Массивы в JavaScript могут работать и как очередь, и как стек. Существует возможность добавлять/удалять элементы как в начало, так и в конец массива (рис. 6).

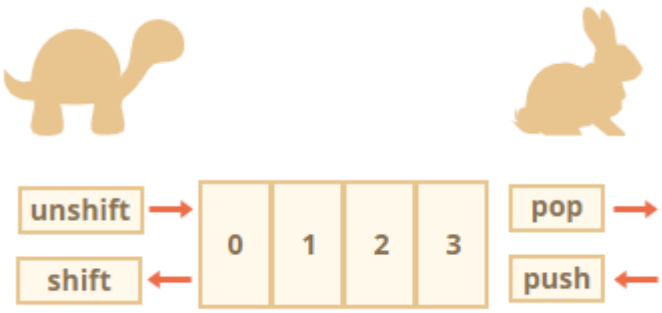


Рисунок 6. Работа с массивом через встроенные методы

Метод **pop** удаляет последний элемент массива и возвращает его:

Пример:

```
var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин
```

Метод **push** добавляет элемент в конец массива:

```
Пример:

    var fruits = ["Яблоко", "Апельсин"];

    fruits.push("Груша");

    alert( fruits ); // Яблоко, Апельсин, Груша
```

Метод **shift** удаляет из массива первый элемент и возвращает его:

```
Пример:

    var fruits = ["Яблоко", "Апельсин", "Груша"];

    alert( fruits.shift() ); // удаляем Яблоко и выводим его

    alert( fruits ); // Апельсин, Груша
```

Метод **unshift** добавляет элемент в начало массива:

```
Пример:

    var fruits = ["Апельсин", "Груша"];

    fruits.unshift('Яблоко');

    alert( fruits ); // Яблоко, Апельсин, Груша
```

Внутреннее устройство массива

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это по сути обычный синтаксис доступа по ключу, как obj[key], где в роли obj представлен arr, а в качестве ключа – числовой индекс.

Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length. Но в основе всё равно лежит объект.

Следует помнить, что в JavaScript существует всего 7 основных типов

данных. Массив является объектом и, следовательно, ведёт себя как объект.

Но то, что действительно делает массивы особенными – это их внутреннее представление. Движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим, так, как это показано на иллюстрациях к этой главе. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро.

Но все они утратят эффективность, если перестать работать с массивом как с «упорядоченной коллекцией данных» и использовать его как обычный объект.

Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, следует использовать их именно в таких случаях. В случае, если необходимы произвольные ключи лучше подойдёт обычный объект {}.

Перебор элементов массива

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

Пример:

```
var arr = ["Яблоко", "Апельсин", "Груша"];

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

Но для массивов возможен и другой вариант цикла, for..of:

Пример:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям
for (var fruit of fruits)
{
  alert( fruit );
}
```

В последнем случае – очередное значение массива копируется в переменную *fruit* и далее в теле цикла с ней возможно провести различные операции. В приставленном примере – вывод на экран через диалоговое окно. Следует отметить, что данный вид цикла подразумевает использование массива только для чтения.

Объектная модель браузера

Веб-страницы бывают статическими и динамическими, последние отличаются тем, что в них используются сценарии (программы) на языке JavaScript.

В сценариях JavaScript браузер веб-разработчику предоставляет множество "готовых" объектов, с помощью которых он может взаимодействовать с элементами веб-страницы и самим браузером. В совокупности все эти объекты составляют объектную модель браузера (**BOM – Browser Object Model**).

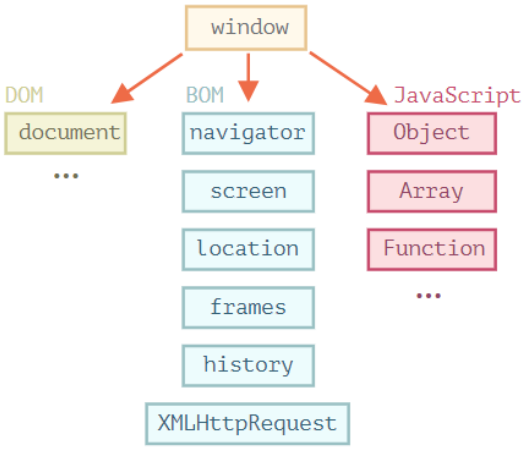


Рисунок 7. Объектная модель браузера

На самом верху этой модели (рис. 7) находится глобальный объект **window**. Он представляет собой одно из окон или вкладку браузера с его панелями инструментов, меню, строкой состояния, HTML страницей и другими объектами. Доступ к этим различным объектам окна браузера осуществляется с помощью следующих основных объектов: **navigator, history, location, screen, document** и т.д. Так как данные объекты являются дочерними по отношению к объекту **window**, то обращение к ним происходит как к свойствам объекта **window**.

Из всех этих объектов, наибольший интерес и значимость для разработчика представляет объект **document**, который является корнем

объектной модели документа (**DOM – Document Object Model**). Данная модель в отличие от объектной модели браузера стандартизована в спецификации и поддерживается всеми браузерами.

Объект document представляет собой HTML документ, загруженный в окно (вкладку) браузера. С помощью свойств и методов данного объекта возможно получить доступ к содержимому HTML-документа, а также изменить его содержимое, структуру и оформление.

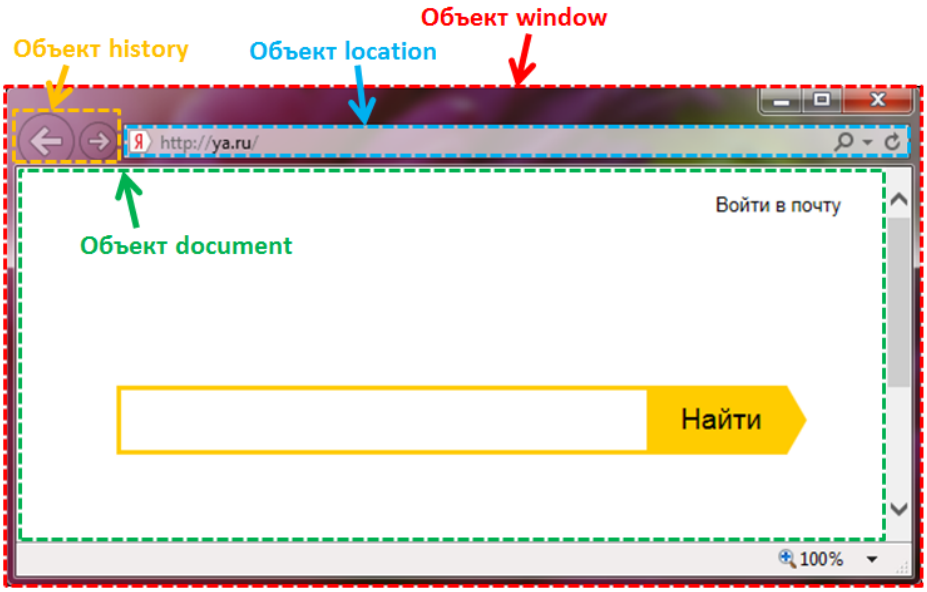


Рисунок 8. Объекты модели BOM

Объект window

window – самый главный объект в браузере, который отвечает за одно из окон (вкладок) браузера. Он является корнем иерархии всех объектов доступных веб-разработчику в сценариях JavaScript. Объект window кроме глобальных объектов (document, screen, location, navigator и др.) имеет собственные свойства и методы, которые предназначены для:

- открытия нового окна (вкладки);
- закрытия окна (вкладки) с помощью метода close();
- распечатывания содержимого окна (вкладки);
- передачи фокуса окну или для его перемещения на задний план (за всеми окнами);
- управления положением и размерами окна, а также для осуществления прокручивания его содержимого;
- изменения содержимого статусной строки браузера;
- взаимодействия с пользователем посредством следующих окон: alert

(для вывода сообщений), confirm (для вывода окна, в котором пользователю необходимо подтвердить или отменить действия), prompt (для получения данных от пользователя);

- выполнения определённых действий через определённые промежутки времени и др.

Пример. Вызвать метод write объекта document, который расположен в текущей вкладке (окне) браузера:

```
window.document.write("Строчка текста");

<!-- Так как данный метод мы вызываем для текущего окна, то "window." можно опустить: -->

document.write("Строчка текста");
```

Размер окна

Чтобы определить размер окна браузера, можно использовать два свойства. Оба свойства возвращают размер в пикселях:

window.innerHeight – внутренняя высота окна браузера (в пикселях)

window.innerWidth – внутренняя ширина окна браузера (в пикселях)

Окно браузера (область просмотра) не включает панель инструментов и полосу прокрутки.

Практическое решение JavaScript (охватывает все браузеры):

Пример.

```
var w = window.innerWidth
|| document.documentElement.clientWidth
|| document.body.clientWidth;

var h = window.innerHeight
|| document.documentElement.clientHeight
|| document.body.clientHeight;
```

Другие методы объекта window:

- **window.open()** - открывает новое окно
- **window.close()** - закрывает текущее окно
- **window.moveTo()** - передвигает текущее окно
- **window.resizeTo()** - изменяет размер текущего окна

Объект navigator

navigator – информационный объект с помощью которого существует возможность получить различные данные, содержащиеся в браузере:

- информацию о самом браузере в виде строки (User Agent);
- внутреннее "кодовое" и официальное имя браузера;
- версию и язык браузера;
- информацию о сетевом соединении и местоположении устройства пользователя;
- информацию об операционной системе и многое другое.

Файлы cookie

Свойство **cookieEnabled** возвращает значение **true**, если **cookie** (специальные файлы-метки) разрешены, в обратном случае возвращается значение **false**:

Пример.

```
<p id="demo"></p>

<script>
        document.getElementById("demo").innerHTML =
        "cookiesEnabled установлено в " +
        navigator.cookieEnabled;
</script>
```

Имя браузера как приложения

Свойство **appName** возвращает имя браузера, как приложения:

Пример.

```
<p id="demo"></p>

<script>
        document.getElementById("demo").innerHTML =
        "navigator.appName - " + navigator.appName;
</script>
```

Кодовое имя браузера

Свойство **appCodeName** возвращает кодовое имя браузера:

Пример.

```
<p id="demo"></p>

<script>
        document.getElementById("demo").innerHTML =
        "navigator.appCodeName - " +
        navigator.appCodeName;
</script>
```

Движок браузера

Свойство **product** возвращает имя движка браузера:

Пример.

```
<p id="demo"></p>

<script>
        document.getElementById("demo").innerHTML =
        "navigator.product - " + navigator.product;
</script>
```

Другие свойства:

- Свойство **appVersion** возвращает информацию о версии браузера;
- Свойство **userAgent** возвращает заголовок пользовательского агента, посланного браузером серверу;
- Свойство **platform** возвращает платформу браузера (операционную систему);
- Свойство **language** возвращает язык браузера;
- Свойство **onLine** возвращает true, если браузер подключен к сети Интернет;
- Метод **javaEnabled()** возвращает true, если обработка Java включена.

Объект history

history – объект, который позволяет получить историю переходов пользователя по ссылкам в пределах одного окна (вкладки) браузера. Данный объект отвечает за кнопки forward (вперёд) и back (назад). С помощью методов объекта history можно имитировать нажатие на эти кнопки, а также переходить на определённое количество ссылок в истории вперёд или назад. Кроме этого, с появлением HTML5 History API веб-разработчику стали доступны методы для добавления и изменения записей в истории, а также событие, с помощью которого возможно обрабатывать нажатие кнопок forward (вперёд) и back (назад).

Метод **history.back()** загружает предыдущий URL в списке посещенных страниц. По сути этот метод действует так же, как кнопка браузера "Назад". В следующем примере на странице создается своя кнопка возврата назад:

Пример:

```
<script>
        function goBack() {
                                window.history.back();
                                }
</script>

<body>
        <input type="button" value="Назад" onclick="goBack()">
</body>
```

Метод **history.forward()** загружает следующий URL в списке посещенных страниц. Он действует аналогично предыдущему методу. По сути этот метод действует так же, как кнопка браузера "Вперед".

Объект location

location – объект, который отвечает за адресную строку браузера. Данный объект содержит свойства и методы, которые позволяют: получить текущий адрес страницы браузера, перейти по указанному URL, перезагрузить страницу и т.п. Существующие методы:

- **window.location.href** возвращает ссылку (URL) текущей страницы;
- **window.location.hostname** возвращает доменное имя веб-хоста;
- **window.location.pathname** возвращает путь и имя файла текущей страницы;
- **window.location.protocol** возвращает использованный веб-протокол (http: или https:);
- **window.location.assign** загружает новый документ.

Объект screen

screen – объект, который предоставляет информацию об экране пользователя: разрешение экрана, максимальную ширину и высоту, которую может иметь окно браузера, глубина цвета и т.д.

Свойства:

- **screen.width** – возвращает ширину экрана;
- **screen.height** – возвращает высоту экрана;
- **screen.availWidth** - возвращает ширину экрана пользователя в пикселях минус интерфейс вроде таскбара Windows;
- **screen.availHeight** - возвращает высоту экрана пользователя в пикселях минус интерфейс вроде таскбара Windows;
- **screen.colorDepth** - возвращает число бит, используемых для отображения цвета;
- **screen.pixelDepth** - возвращает глубину пикселя экрана в битах.

Объект document

document – HTML документ, загруженный в окно (вкладку) браузера. Он

является корневым узлом HTML документа и "владельцем" всех других узлов: элементов, текстовых узлов, атрибутов и комментариев. Объект **document** содержит свойства и методы для доступа ко всем узловым объектам. **document** как и другие объекты, является частью объекта **window** и, следовательно, он может быть доступен как **window.document**.

Объектная модель документа: DOM

В момент открытия веб-страницы в браузере, он получает исходный текст HTML и разбирает (парсит) его. Браузер строит модель структуры документа и использует её, чтобы нарисовать страницу на экране.

Это представление документа и есть одна из моделей, доступных в песочнице JavaScript. Существует возможность модифицировать ее. Это возможно в реальном времени – как только вносятся изменения, страница на экране обновляется, отражая их.

Структура документа

Можно представить HTML как набор вложенных коробок. Теги представляют контейнеры и включают в себя другие теги, которые в свою очередь включают теги, или текст. Далее приведен пример документа:

Пример:

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1> My home page </h1>
    <p>Hello, I am Marijn and this is..</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

Структура данного документа будет иметь вид, представленный на рисунке 9.

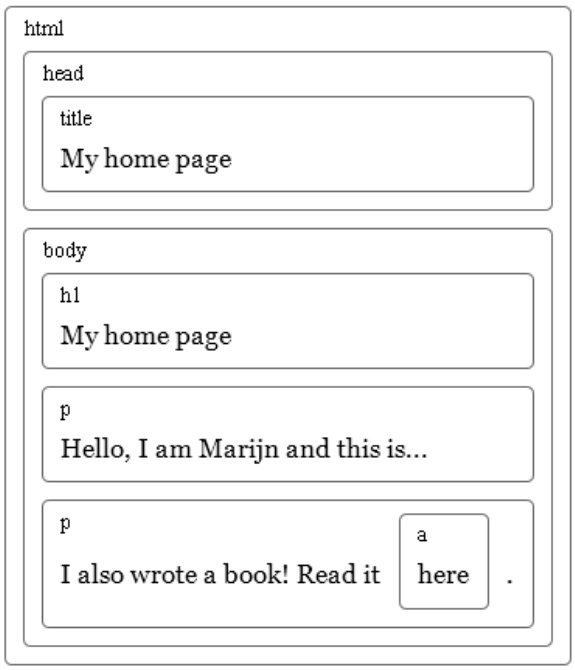


Рисунок 9. Структура документа

Структура данных, использующаяся браузером для представления документа, отражает его форму. Для каждого блока есть объект, с которым можно взаимодействовать и узнавать про него разные данные – какой тег он представляет, какие блоки и текст содержит. Это представление называется **Document Object Model** (объектная модель документа), или сокращённо **DOM**.

Доступ к этим объектам осуществляется через глобальную переменную **document**. Её свойство **documentElement** ссылается на объект, представляющий тег. Он также предоставляет свойства **head** и **body**, в которых содержатся объекты для соответствующих элементов.

Деревья

Структура документа браузера представляется в виде синтаксического дерева. Каждый узел может ссылаться на другие узлы, у каждого из ответвлений может быть своё ответвление. Эта структура – типичный пример вложенных структур, где элементы содержат подэлементы, похожие на них самих.

Структура данных является деревом, когда она разветвляется, не имеет циклов (узел не может содержать сам себя), и имеет единственный ярко выраженный «корень». В случае DOM в качестве корня выступает **document.documentElement**.

То же и у DOM. Узлы для обычных элементов, представляющих теги HTML, определяют структуру документа. У них могут быть дочерние узлы.

Пример такого узла — **document.body**. Некоторые из этих дочерних узлов могут оказаться листьями – например, текст или комментарии (в HTML комментарии записываются между символами).

У каждого узлового объекта DOM есть свойство **nodeType**, содержащее цифровой код, определяющий тип узла. У обычных элементов он равен 1, что также определено в виде свойства-константы **document.ELEMENT_NODE**. У текстовых узлов, представляющих отрывки текста, он равен 3 (**document.TEXT_NODE**). У комментариев — 8 (**document.COMMENT_NODE**).

То есть, существует ещё один способ графически представить дерево документа (рис. 10).

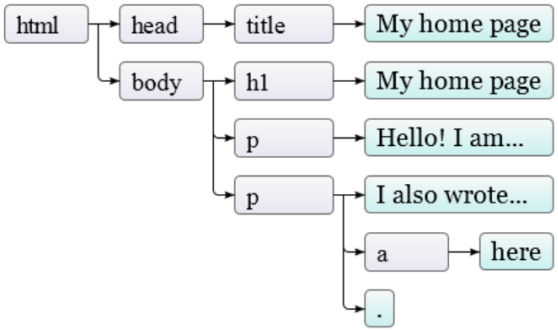


Рисунок 10. Структура документа в виде дерева

Листья – текстовые узлы, а стрелки показывают взаимоотношения отец-ребёнок между узлами.

Обход дерева

Узлы DOM содержат много ссылок. Это показано на рисунке 11.

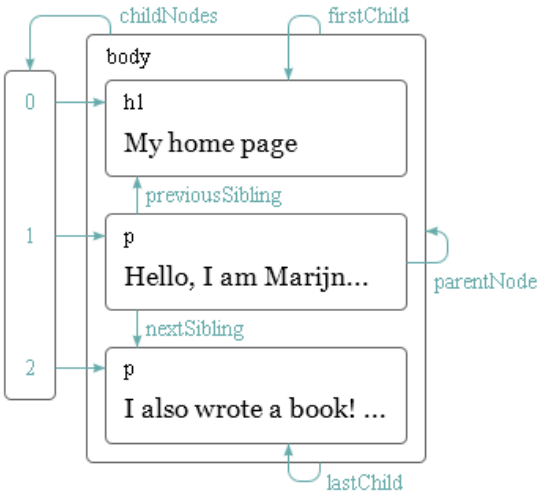


Рисунок 11. Ссылки внутри документа

Хотя тут показано только по одной ссылке каждого типа, у каждого узла есть свойство **parentNode**, указывающего на его родительский узел. Также у каждого узла-элемента (тип 1) есть свойство **childNodes**, указывающее на массивоподобный объект, содержащий его дочерние узлы.

В теории можно пройти в любую часть дерева, используя только эти ссылки. Но JavaScript предоставляет много дополнительных вспомогательных ссылок. Свойства **firstChild** и **lastChild** показывают на первый и последний дочерний элементы, или содержат **null** у тех узлов, у которых нет дочерних. **previousSibling** и **nextSibling** указывают на соседние узлы – узлы того же родителя, что и текущего узла, но находящиеся в списке сразу до или после текущей. У первого узла свойство **previousSibling** будет **null**, а у последнего **nextSibling** будет **null**.

При работе с такими вложенными структурами пригождаются рекурсивные функции. В примере происходит поиск в документе текстовых узлы, содержащих заданную строку, и возвращат **true**, когда выполняется условие поиска. Свойства текстового узла nodeValue содержит строчку текста.

Пример:

```
function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (var i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string))
        return true;
    }
    return false;
  } else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "книг"));
// → true
```

Поиск элементов

Часто бывает полезным ориентироваться по этим ссылкам между родителями, детьми и родственными узлами и проходить по всему документу. Однако если нужен конкретный узел в документе, очень неудобно идти по нему,

начиная с **document.body** и перебирая жёстко заданный в коде путь. Поступая так, вносятся в программу допущения о точной структуре документа – а её мы позже может возникнуть потребность поменять. Другой усложняющий фактор – текстовые узлы создаются даже для пробелов между узлами. В документе из примера у тега **body** не три дочерних (**h1** и два **p**), а целых семь: эти три плюс пробелы до, после и между ними.

Так что если нужен атрибут **href** из ссылки, не следует писать в программе что-то вроде: «второй ребёнок шестого ребёнка **document.body**». Лучше бы, если б мы могли сказать: «первая ссылка в документе». И так можно сделать:

Пример:

```
var link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

У всех узлов-элементов есть метод **getElementsByTagName**, собирающий все элементы с данным тэгом, которые происходят (прямые или не прямые потомки) от этого узла, и возвращает его в виде массивоподобного объекта.

Чтобы найти конкретный узел, можно задать ему атрибут **id** и использовать метод **document.getElementById**.

Пример:

```
<p>Мой университет МИРЭА:</p>
<p><img id="mirea" src="img/mirea.png"></p>

<script>
  var ostrich = document.getElementById("mirea");
  console.log(ostrich.src);
</script>
```

Третий метод – **getElementsByClassName**, который, как и **getElementsByTagName**, ищет в содержимом узла-элемента и возвращает все элементы, содержащие в своём классе заданную строчку.

Изменение документа

Почти всё в структуре DOM можно менять. У узлов-элементов есть набор методов, которые используются для их изменения. Метод **removeChild** удаляет заданную дочерний узел. Для добавления узла можно использовать **appendChild**, который добавляет узел в конец списка, либо **insertBefore**, добавляющий узел, переданную первым аргументом, перед узлом, переданным вторым аргументом.

Пример:

```
<p>Один</p>
<p>Два</p>
<p>Три</p>

<script>
  var paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

Узел может существовать в документе только в одном месте. Поэтому вставляя параграф «Три» перед параграфом «Один» фактически происходит его удаление из конца списка и вставка в начало. В результате получается «Три/Один/Два». Все операции по вставке узла приведут к его исчезновению с текущей позиции (если у него таковая была).

Метод **replaceChild** используется для замены одного дочернего узла другим. Он принимает два узла: новый, и тот, который надо заменить. Заменяемый узел должен быть дочерним узлом того элемента, чей метод мы вызываем. Как **replaceChild**, так и **insertBefore** в качестве первого аргумента ожидают получить новый узел.

Создание узлов

В следующем примере реализуется скрипт, заменяющий все картинки (тег **img**) в документе текстом, содержащимся в их атрибуте “**alt**”, который задаёт альтернативное текстовое представление картинки.

Для этого необходимо не только удалить картинки, но и добавить новые текстовые узлы им на замену. Для этого используется метод **document.createTextNode**.

Пример:

```
<p>Это <img src="img/cat.png" alt="Кошка"> в
  <img src="img/hat.png" alt="сапожках">.</p>

<p><button onclick="replaceImages()">Заменить</button></p>

<script>
  function replaceImages() {
    var images = document.body.getElementsByTagName("img");
    for (var i = images.length - 1; i >= 0; i--) {
      var image = images[i];
      if (image.alt) {
        var text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Получая строку, **createTextNode** возвращает тип 3 узла **DOM** (текстовый), который можно вставить в документ, чтобы он был показан на экране.

Цикл по картинкам начинается в конце списка узлов. Это сделано потому, что список узлов, возвращаемый методом **getElementsByTagName** (или свойством **childNodes**) постоянно обновляется при изменениях документа. Если бы перебор элементов был начат с начала списка, удаление первой картинки привело бы к потере списком первого элемента, и во время второго прохода цикла, когда i равно 1, он бы остановился, потому что длина списка стала бы также равняться 1.

Если необходимо работать с фиксированным списком узлов вместо «живого», можно преобразовать его в настоящий массив при помощи метода **slice**.

Пример:

```
var arrayish = {0: "один", 1: "два", length: 2};
var real = Array.prototype.slice.call(arrayish, 0);
real.forEach(function(elt) { console.log(elt); });
// → один
//   два
```

Для создания узлов-элементов (тип 1) можно использовать **document.createElement**. Метод принимает имя тега и возвращает новый пустой узел заданного типа. Следующий пример определяет инструмент **elt**, создающий узел-элемент и использующий остальные аргументы в качестве его детей. Эта функция потом используется для добавления дополнительной информации к цитате.

Пример:

```
<blockquote id="quote">
Никакая книга не может быть закончена. Во время работы над ней мы
узнаём достаточно для того, чтобы найти её незрелой сразу же после
того, как мы отвлеклись от неё.
</blockquote>

<script>
  function elt(type) {
    var node = document.createElement(type);
    for (var i = 1; i < arguments.length; i++) {
      var child = arguments[i];
      if (typeof child == "string")
        child = document.createTextNode(child);
      node.appendChild(child);
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Карл Поппер"),
        ", предисловие ко второму изданию ",
        elt("em", "Открытое общество и его враги "),
        ", 1950"));
</script>
```

Атрибуты

К некоторым элементам атрибутов, типа href у ссылок, можно получить доступ через одноимённое свойство объекта. Это возможно для ограниченного числа часто используемых стандартных атрибутов.

Но HTML позволяет назначать узлам любые атрибуты. Это полезно, т.к. позволяет вам хранить дополнительную информацию в документе. Если вы придумаете свои названия атрибутов, их не будет среди свойств узла-элемента. Вместо этого необходимо использовать методы **getAttribute** и **setAttribute** для работы с ними.

Пример:

```
<p data-classified="secret">Код запуска 00000000.</p>
<p data-classified="unclassified">У кошки четыре ноги.</p>

<script>
  var paras = document.body.getElementsByTagName("p");
  Array.prototype.forEach.call(paras, function(para) {
    if (para.getAttribute("data-classified") == "secret")
      para.parentNode.removeChild(para);
  });
</script>
```

Рекомендуется перед именами придуманных атрибутов ставить “**data-**“, чтобы быть уверенным, что они не конфликтуют с любыми другими. В качестве простого примера предлагается реализовать подсветку синтаксиса, который ищет теги **<pre>** с атрибутом **data-language** (язык) и пытается подсветить ключевые слова в языке.

Пример:

```
function highlightCode(node, keywords) {
  var text = node.textContent;
  node.textContent = ""; // Очистим узел

  var match, pos = 0;
  while (match = keywords.exec(text)) {
    var before = text.slice(pos, match.index);
    node.appendChild(document.createTextNode(before));
    var strong = document.createElement("strong");
    strong.appendChild(document.createTextNode(match[0]));
    node.appendChild(strong);
    pos = keywords.lastIndex;
  }
  var after = text.slice(pos);
  node.appendChild(document.createTextNode(after));
}
```

Функция **highlightCode** принимает узел **<pre>** и регулярное выражение (с включённой настройкой **global**), совпадающую с ключевым словом языка программирования, которое содержит элемент.

Свойство **textContent** используется для получения всего текста узла, а затем устанавливается в пустую строку, что приводит к очищению узла. Осуществляется проход в цикле по всем вхождениям выражения **keyword**, добавление между ними текста в виде простых текстовых узлов, а совпавший текст (ключевые слова) добавляются, заключенные в элементы **** (жирный шрифт).

Существует еще один часто используемый атрибут, **class**, имя которого является ключевым словом в JavaScript. По историческим причинам, когда старые реализации JavaScript не умели обращаться с именами свойств, совпадавшими с ключевыми словами, этот атрибут доступен через свойство под названием **className**. К нему также можно получить доступ по его настоящему имени "**class**" через методы **getAttribute** и **setAttribute**.

Расположение элементов (layout)

Можно заметить, что разные типы элементов располагаются по-разному. Некоторые, типа параграфов **<p>** и заголовков **<h1>** растягиваются на всю ширину документа и появляются на отдельных строках. Такие элементы называют блочными. Другие, как ссылки **<a>** или жирный текст **** появляются на одной строчке с окружающим их текстом. Они называются встроенными (inline).

Для любого документа браузеры могут построить расположение элементов, расклад, в котором у каждого будет размер и положение на основе его типа и содержимого. Затем этот расклад используется для создания внешнего вида документа.

Размер и положение элемента можно узнать через JavaScript. Свойства **offsetWidth** и **offsetHeight** возвращают размер в пикселях, занимаемый элементом. Пиксель – основная единица измерений в браузерах, и обычно соответствует размеру минимальной точки экрана. Сходным образом, **clientWidth** и **clientHeight** возвращают размер внутренней части элемента, не считая рамки (свойство border).

Пример:

```
<p style="border: 3px solid red">
  Текст в блоке
</p>

<script>
  var para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

Самый эффективный способ узнать точное расположение элемента на экране – метод **getBoundingClientRect**. Он возвращает объект со свойствами **top, bottom, left, и right** (сверху, снизу, слева и справа), которые содержат положение элемента относительно левого верхнего угла экрана в пикселях. Если необходимо получить эти данные относительно всего документа, следует прибавить текущую позицию прокрутки, которая содержится в глобальных переменных **pageXOffset** и **pageYOffset**.

Разбор документа – крайне сложная задача. В целях быстродействия браузерные движки не перестраивают документ каждый раз после его изменения, а ждут так долго, как это возможно. Когда программа JavaScript, изменившая документ, заканчивает работу, браузеру надо будет просчитать новую раскладку страницы, чтобы вывести изменённый документ на экран. Когда программа запрашивает позицию или размер чего-либо, читая свойства типа **offsetHeight** или вызывая **getBoundingClientRect**, для предоставления корректной информации тоже необходимо рассчитывать раскладку.

Программа, которая периодически считывает раскладку DOM и изменяет DOM, заставляет браузер много раз пересчитывать раскладку, и в связи с этим будет работать медленно. В следующем примере есть две разные программы, которые строят линию из символов X шириной в 2000 пикс, и измеряют время работы.

Пример:

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
 function time(name, action) {
   var start = Date.now(); // Текущее время в миллисекундах
   action();
   console.log(name, "заняло", Date.now() - start, "ms");
 }

 time("тупо", function() {
   var target = document.getElementById("one");
   while (target.offsetWidth < 2000)
     target.appendChild(document.createTextNode("X"));
 });
 // → тупо заняло 32 ms

 time("умно", function() {
   var target = document.getElementById("two");
   target.appendChild(document.createTextNode("XXXXX"));
   var total = Math.ceil(2000 / (target.offsetWidth / 5));
   for (var i = 5; i < total; i++)
     target.appendChild(document.createTextNode("X"));
 });
 // → умно заняло 1 ms
</script>
```

Стили и классы

JavaScript может менять и **CSS-классы**, и свойство **style**.

Классы – всегда предпочтительный вариант по сравнению со style. Следует манипулировать свойством style только в том случае, если классы не могут решить требуемую задачу.

className и classList

Изменение класса является одним из наиболее часто используемых

действий в скриптах. Для классов было введено свойство "className": elem.className соответствует атрибуту "class".

Пример:

```
<body class="main page">
  <script>
    alert(document.body.className); // main page
  </script>
</body>
```

Если происходит присваивание elem.className, то это заменяет всю строку с классами. Но, иногда необходимо работать с отдельным классом. Для этого есть другое свойство: elem.classList – это специальный объект с методами для добавления/удаления одного класса. Например:

Пример:

```
<body class="main page">
  <script>
    // добавление класса
    document.body.classList.add('article');

    alert(document.body.className); // main page article
  </script>
</body>
```

Как можно заметить существует возможность работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList.

*Методы **classList**:*

- **elem.classList.add/remove("class")** – добавить/удалить класс.
- **elem.classList.toggle("class")** – добавить класс, если его нет, иначе удалить.
- **elem.classList.contains("class")** – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of.

Пример:

```
<body class="main page">
  <script>
    for (let name of document.body.classList) {
      alert(name); // main, затем page
    }
  </script>
</body>
```

Element style

Свойство **elem.style** – это объект, который соответствует тому, что написано в атрибуте "**style**". Установка стиля **elem.style.width="100px"** работает так же, как наличие в атрибуте **style** строки **width:100px**.

Для свойства из нескольких слов используется **camelCase**:

Пример:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Пример:

```
document.body.style.backgroundColor = prompt('background
color?', 'green');
```

Сброс стилей

Иногда необходимо добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, мы можем задать elem.style.display = "none".

Также можно удалить свойство **style.display**, чтобы вернуться к первоначальному состоянию. Вместо **delete elem.style.display** следует присвоить ему пустую строку: **elem.style.display** = "".

Пример:

```
// если мы запустим этот код, <body> "мигнёт"
document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000); //
возврат к нормальному состоянию
```

Если установить в **style.display** пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства **style.display** вообще не было.

Полная перезапись style.cssText

Обычно используется **style.*** для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, **div.style="color: red; width: 100px"**, потому что **div.style** – это объект, и он доступен только для чтения. Для задания нескольких стилей в одной строке используется специальное свойство **style.cssText**.

Пример:

```
<div id="div">Button</div>

<script>
  // можем даже устанавливать специальные флаги для
стилей, например, "important"
  div.style.cssText="color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center; ";

  alert(div.style.cssText);
</script>
```

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно ненароком удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда точно известно, что не удалится существующий стиль.

То же самое можно сделать установкой атрибута: **div.setAttribute('style', 'color: red...')**.

Вычисленные стили getComputedStyle

Крайне важной задачей помимо задания стилей является возможность их прочитать. Например, необходимо знать размер, отступы, цвет элемента. Свойство **style** оперирует только значением атрибута "**style**", без учёта CSS-каскада. Поэтому, используя **elem.style**, нельзя прочитать ничего, что приходит из классов CSS. Например, здесь style не может видеть отступы:

Пример:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  Красный текст
  <script>
    alert(document.body.style.color); // пусто
    alert(document.body.style.marginTop); // пусто
  </script>
</body>
```

В случае, если необходимо увеличить отступ на 20px необходимо прочитать исходные значения свойства. Для этого есть метод: getComputedStyle.

getComputedStyle(element, [pseudo])

- **element -** элемент, значения для которого нужно получить;
- **pseudo** – указывается, если нужен стиль псевдоэлемента, например **::before**. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на **elem.style**, но с учётом всех CSS-классов.

Пример:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  <script>
    let computedStyle = getComputedStyle(document.body);

    // сейчас мы можем прочитать отступ и цвет

    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>

</body>
```

Браузерные события и их обработка

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Далее представлен список самых часто используемых DOM-событий. Полный перечень событий представлен в Приложении А.

События мыши:

- **click** – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- **contextmenu** – происходит, когда кликнули на элемент правой кнопкой мыши.
- **mouseover / mouseout** – когда мышь наводится на / покидает элемент.
- **mousedown / mouseup** – когда нажали / отжали кнопку мыши на элементе.
- **mousemove** – при движении мыши.

События на элементах управления:

- **submit** – пользователь отправил форму <form>.
- **focus** – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:

- **keydown и keyup** – когда пользователь нажимает / отпускает клавишу.

События документа:

- **DOMContentLoaded** – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

- **transitionend** – когда CSS-анимация завершена.

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя. Есть несколько способов назначить событию обработчик.

Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется **on<событие>**. Например, чтобы назначить обработчик события **click** на элементе **input**, можно использовать атрибут **onclick**:

Пример:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте **onclick**.

Следует обратить внимание, для содержимого атрибута **onclick** используются одинарные кавычки, так как сам атрибут находится в двойных. Если это не учесть и поставить двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого

количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её.

Следующий пример по клику запускает функцию **countRabbits()**:

Пример:

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента **on<событие>**. К примеру, **elem.onclick**:

Пример:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```

Доступ к элементу через this

Внутри обработчика события **this** ссылается на текущий элемент, то есть на тот, на котором, назначен обработчик. В коде ниже **button** выводит своё содержимое, используя **this.innerHTML**:

Пример:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

Частые ошибки

Функция должна быть присвоена как sayThanks, а не sayThanks().

Пример:

```
// правильно
button.onclick = sayThanks;

// неправильно
button.onclick = sayThanks();
```

Если добавить скобки, то **sayThanks()** – это уже вызов функции, результат которого (равный **undefined**, так как функция ничего не возвращает) будет присвоен **onclick**. Так что это не будет работать. А вот в разметке, в отличие от свойства, скобки нужны:

Пример:

```
<input type="button" id="button" onclick="sayThanks()">
```

Это различие просто объяснить. При создании обработчика браузером из атрибута, он автоматически создаёт функцию с *телом из значения атрибута*: **sayThanks()**.

Так что разметка генерирует такое свойство:

Пример:

```
button.onclick = function() {
  sayThanks(); // содержимое атрибута
};
```

Использовать именно функции, а не строки.

Назначение обработчика строкой **elem.onclick = "alert(1)"** также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не использовать setAttribute для обработчиков.

Такой вызов работать не будет:

Пример:

```
// при нажатии на body будут ошибки,
// атрибуты всегда строки, и функция станет строкой
document.body.setAttribute('onclick', function() { alert(1) });
```

Регистр DOM-свойства имеет значение.

Использовать следует elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика –- невозможность повесить несколько обработчиков на одно событие. Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение. Следовательно, необходимо назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

Пример:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // заменит
предыдущий обработчик
```

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов **addEventListener** и **removeEventListener**. Они свободны от указанного недостатка.

Синтаксис добавления обработчика:

element.addEventListener(event, handler[, options]);

event - имя события, например "click".
handler - ссылка на функцию-обработчик.

options - дополнительный объект со свойствами:

- **once:** если true, тогда обработчик будет автоматически удалён после выполнения.
- **capture:** фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе Всплытие и погружение. Так исторически сложилось, что options может быть false/true, это тоже самое, что {capture: false/true}.
- **passive:** если true, то указывает, что обработчик никогда не вызовет preventDefault(), подробнее об этом будет рассказано в главе Действия браузера по умолчанию.

Для удаления обработчика следует использовать **removeEventListener**

Синтаксис добавления обработчика:

```
element.removeEventListener(event, handler[, options]);
```

Метод **addEventListener** позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

Пример:

```
<input id="elem" type="button" value="Нажми меня"/>

<script>
  function handler1() {
    alert('Спасибо!');
  };

  function handler2() {
    alert('Спасибо ещё раз!');
  }

  elem.onclick = () => alert("Привет");
  elem.addEventListener("click", handler1); // Спасибо!
  elem.addEventListener("click", handler2); // Спасибо ещё
раз!
</script>
```

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство и через **addEventListener**. Однако, во избежание путаницы, рекомендуется выбрать один способ.

Объект события

Чтобы корректно обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

Пример демонстрирует получение координат мыши из объекта события:

Пример:

```
<input type="button" value="Нажми меня" id="elem">

<script>
  elem.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert("Координаты: " + event.clientX + ":" +
event.clientY);
  };
</script>
```

Некоторые свойства объекта event:

- **event.type** - тип события, в данном случае "**click**";
- **event.currentTarget** - элемент, на котором сработал обработчик. Значение – обычно такое же, как и у **this**, но если обработчик является функцией-стрелкой или при помощи **bind** привязан другой объект в качестве **this**, то возмоно получить элемент из **event.currentTarget**.
- **event.clientX / event.clientY - к**оординаты курсора в момент клика относительно окна, для событий мыши.

Объект-обработчик: handleEvent

Существует возможность назначить обработчиком не только функцию, но и объект при помощи **addEventListener**. В этом случае, когда происходит событие, вызывается метод объекта **handleEvent**.

Пример:

```
<button id="elem">Нажми меня</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " на " + event.currentTarget);
    }
  });
</script>
```

Можно заметить, если **addEventListener** получает объект в качестве обработчика, он вызывает **object.handleEvent(event)**, когда происходит событие.

Также можно использовать для этого класс:

Пример:

```
<button id="elem">Нажми меня</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Нажата кнопка мыши";
          break;
        case 'mouseup':
          elem.innerHTML += "...и отжата.";
          break;
      }
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Здесь один и тот же объект обрабатывает оба события. Следует обратить внимание, что необходимо явно назначить оба обработчика через

addEventListener. Тогда объект **menu** будет получать события **mousedown** и **mouseup**, но не другие (не назначенные) типы событий.

Метод **handleEvent** не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые приспособлены под обработку конкретных типов событий, вот так:

Пример:

```
<button id="elem">Нажми меня</button>

<script>
  class Menu {
    handleEvent(event) {
      // mousedown -> onMousedown
      let method = 'on' + event.type[0].toUpperCase() +
event.type.slice(1);
      this[method](event);
    }

    onMousedown() {
      elem.innerHTML = "Кнопка мыши нажата";
    }

    onMouseup() {
      elem.innerHTML += "...и отжата.";
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ JQUERY

jQuery – это библиотека JavaScript (т.е., она написана на JavaScript), предназначенная для абстрагирования, выравнивания, исправления и упрощения скриптинга при работе с узлами HTML-элементов в браузере или для работы в браузере без графического интерфейса.

Ключевые возможности jQuery:

- абстрагируется интерфейс объектной модели документа (он же DOM API);
- выравниваются все различия реализаций DOM, существующие между браузерами;
- исправляются известные браузерные баги, связанные с CSS и DOM.

Все это оборачивается в более простой и менее некорректный API, в отличие от нативного API DOM – основное преимущество библиотеки jQuery.

При помощи jQuery совершенно тривиально решаются задачи «визуально скрыть второй элемент h2 в документе .html». Код jQuery для такой задачи выглядел бы так:

Пример:

```
jQuery ('h2:eq(1)').hide();
```

Сначала вызывается функция jQuery(), ей мы передаем специальный CSS-селектор библиотеки jQuery, выбирающий второй элемент h2 в HTML-документе. Затем вызывается метод jQuery.hide(), скрывающий элемент h2.

Проведя сравнение с нативным DOM-кодом, который потребовалось бы написать, без использования jQuery можно увидеть существенную разницу.

Пример:

```
document.querySelectorAll('h2')[1].style.setProperty('display','none');
```

Вышеприведенный нативный DOM-код предполагает, что все браузеры поддерживают использованные здесь методы DOM. Однако некоторые старые браузеры не поддерживают querySelectorAll() или setProperty(). Поэтому вышеприведенный код jQuery вполне нормально работал бы в IE8, а нативный

код DOM вызвал бы ошибку JavaScript.

Используя jQuery, можно не беспокоиться о том, какой браузер что поддерживает, либо какой DOM API в каком браузере может некорректно работать. Работая с jQuery, появляется возможность работать быстрее решать задачи на более простом коде, и при этом не беспокоиться, так как jQuery абстрагирует многие проблемы.

Концепцию jQuery можно сформулировать в виде следующего утверждения: первоочередная задача jQuery заключается в организации выбора (то есть, нахождении) или в создании HTML-элементов для решения практических задач. Без jQuery для этого потребовалось бы больше кода и больше сноровки в обращении с DOM. Достаточно вспомнить рассмотренный выше пример с сокрытием элемента h2.

Следует отметить, что круг возможностей jQuery этим не ограничивается. Она не просто абстрагирует нативные DOM-взаимодействия, но и абстрагирует асинхронные HTTP-запросы (т.н. AJAX) при помощи объекта XMLHttpRequest. Кроме того, в ней есть еще ряд вспомогательных решений на JavaScript и мелких инструментов. Но основная польза jQuery заключается именно в упрощении HTML-скриптинга.

Важный аспект работы с jQuery – не в успешном устранении браузерных багов. «Краеугольные камни» нисколько не связаны с этими возможностями jQuery. В долгосрочном отношении самая сильная сторона jQuery заключается в том, что ее API абстрагирует DOM.

Базовые принципы работы с jQuery

Для начала работы понадобится сам фреймворк, его можно скачать с домашней страницы проекта (https://jquery.com), а затем подключить в коде:

Пример:

```
<head>
<script type="text/javascript" src="jquery.js"></script>
</head>
```

Схематично работа с библиотекой jQuery представлена на рисунке 12.

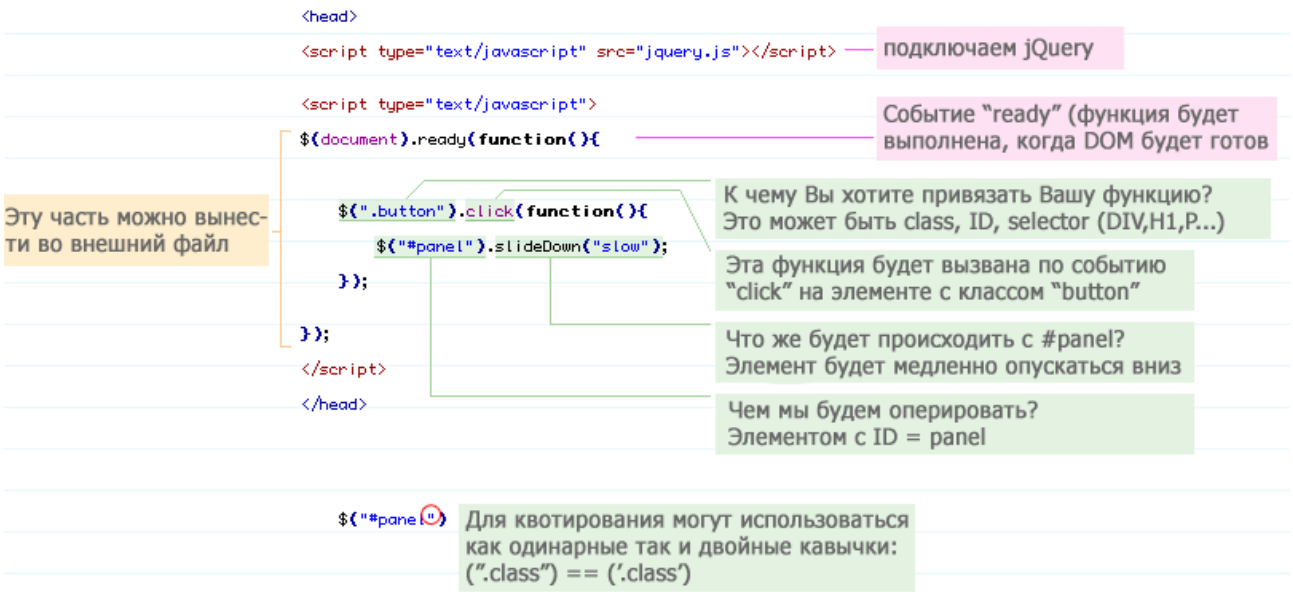


Рисунок 12. Базовые принципы работы с библиотекой jQuery

Порядок работы с библиотекой jQuery:

1. Подключить библиотеку в любом месте кода ДО ЕЕ ИСПОЛЬЗОВАНИЯ;
2. Внутри скрипта JS используя конструкцию $('элемент') – найти нужные элементы;
3. Далее поставить . и указать функции, которые необходимо применить к найденному элементу.

Поиск элементов

Для поиска элементов удобно использовать таблицу 1.

Что необходимо найти	Схема поиска	Пример HTML	Пример jQuery
По имени тега	$('имя тега')	<***p***> </p>	$('p')
По классу	$('.имя класса')	<p ***class='vs'***> </p>	$('.vs')
По id тега	$('#id')	<p ***id='nm'***> </p>	$('#id')
По вложенности	$('#id тег')	<p ***id='z'><a***> </p>	$('#z a')
По параметру	$('тег[параметр]')	<img ***alt***='Карта'>	$('img[alt]')

По конкретному значению параметра	$(‘тег[парам=знач]’)	<input ***type=’button’***>	$(‘input[type=button]’)
По началу значения параметра	$(‘тег[парам^=знач]’)	 	$(‘a[href^=mailto]’)
По окончанию значения параметра	$(‘тег[парам$=знач]’)		$(‘a[href$=com]’)
По части значения параметра	$(‘тег[парам*=знач]’)	 	$(‘a[href*=yandex]’)

Таблица 1. Поиск элементов с использованием jQuery

Фильтрация найденных элементов

Метод .find() возвращает потомков каждого элемента в текущем наборе совпавших элементов, отфильтрованных селектором, объектом jQuery, или элементом. Метод .find() может пройти несколько уровней вложенности, чтобы выбрать всех потомков элемента. Метод .find(), как и большинство методов для фильтрации не возвращает текстовые узлы, чтобы получить все дочерние элементы, включая текстовые узлы и узлы комментариев можно воспользоваться методом .contents().

Синтаксис:

```
// возвращает потомков, если они соответствуют элементу DOM,
или объекту jQuery
$( selector ).find( element )
```

Пример поиска элементов с использованием метода find() представлен далее.

Пример:

```
<!DOCTYPE html>
<html>
	<head>
		<title>Отличие метода .find() от .children()</title>
		<script src =
"https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js"></sc
ript>
		<script>
	$( document ).ready(function(){
	  $( ".children" ).click(function(){ // запускаем функцию при
нажатии на элемент с классом children
	    $( "*" ).css( "background", "#fff" ); // устанавливаем всем
элементам цвет заднего фона белый
	    $( "div" ).children( "span" ).css( "background", "red" ); //
выбираем все дочерние элементы <span> элементов <div> в
документе
	  })
	  $( ".find" ).click(function(){ // запускаем функцию при
нажатии на элемент с классом find
	    $( "*" ).css( "background", "#fff" ); // устанавливаем всем
элементам цвет заднего фона белый
	    $( "div" ).find( "span" ).css( "background", "green"); //
выбираем всех потомки <span> элементов <div>
	  })
	});
		</script>
	</head>
	<body>
		<div>
			Текст
<span>&lt;span&gt;внутри&lt;/span&gt;</span> блока
			<h2>Заголовок
<span>&lt;span&gt;внутри&lt;/span&gt;</span> блока<h2>

	<p><span>&lt;span&gt;Первый&lt;/span&gt;</span>
параграф внутри блока</p>

	<p><span>&lt;span&gt;Второй&lt;/span&gt;</span> параграф
внутри блока</p>
		</div>
	</body>
</html>
```

В представленном примере с использованием jQuery метода **.click()** при нажатии на элемент **<button>** (кнопка) с классом **children** вызывается функция, которая с помощью метода **.css()** устанавливает всем элементам цвет заднего фона белый. Кроме того, с использованием jQuery метода **.children()** выбираются все дочерние элементы **** элементов ** ** в документе, и с помощью метода **.css()** устанавливается им красный цвет заднего фона. В результате чего выбран будет только один элемент, так как только один элемент **** является дочерним элементом.

Также с использованием jQuery метода **.click()** мы при нажатии на элемент **<button>** (кнопка) с классом **find** вызывается функция, которая с помощью метода .**css()** устанавливает всем элементам цвет заднего фона белый. С использованием jQuery метода **find()** выбирает все потомки **** элементов ** ** в документе, и с помощью метода **.css()** устанавливает им зеленый цвет заднего фона. Выбраны будут все вложенные внутри ** ** элементы **.**

События jQuery

Для работы с событиями у jQuery существует собственная событийная модель.

Событие в JavaScript (и jQuery) генерируется, когда что-то происходит с элементом на странице. В список общих событий входят:

- click - генерируется, когда пользователь нажимает на элементе кнопку мыши;
- dblclick - генерируется, когда пользователь делает двойной щелчок кнопкой мыши на элементе;
- mouseover - генерируется, когда пользователь перемещает указатель мыши на элемент;
- load - генерируется после того, как элемент, например, изображение, полностью загружен;
- submit - генерируется, когда происходит отправка формы (данное событие генерируется только для элементов form).

Полный список событий представлен в приложении В.

Для работы с событиями в jQuery нужно создать функцию, называемую обработчиком события, которая будет работать с событием, когда оно произойдёт. Затем вызывается метод jQuery bind, который привязывает функцию обработчик события к определённому событию для выбранного элемента (или

элементов).

Существует много методов jQuery для привязки событий к обработчикам, но метод bind() является основным. Он принимает имя события и имя функции как аргументы и привязывает обработчик к событию для выбранного элемента (или элементов):

Синтаксис:

```
$(selector).bind( eventName, functionName );
```

Затем, когда происходит событие, функция-обработчик автоматически запускается и событие обрабатывается так, как требуется.

Пример:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

$( init );

function init() {
  $('#myButton').bind( 'click', sayHello );
}

function sayHello() {
  alert( "Всем - привет!" );
}
</script>
</head>
<body>
<div>
<input type="button" id="myButton" value="Нажми меня" />
</div>
</body>
</html>
```

Доступ к элементам из обработчика событий

Когда событие вызывает функцию-обработчик, существует возможность получить доступ к элементу как к объекту DOM из функции-обработчика с помощью ключевого слова **this**. Это означает, что можно получить больше информации об элементе, для которого сгенерировано событие, можно манипулировать данным элементом и так далее.

Следующий пример создаёт пульсации кнопки (плавно затухает и плавно высвечивается снова) при нажатии на неё. Чтобы выполнить поставленную задачу, обработчик события получает доступ к объекту нажатой кнопки с помощью **this**, оборачивает его объектом jQuery, а затем вызывает методы jQuery **fadeOut()** и **fadeIn()** для организации пульсирования кнопки:

Пример:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

$( init );

function init() {
  $('#myButton').bind( 'click', pulsate );
}

function pulsate() {
  $(this).fadeOut();
  $(this).fadeIn();
}
</script>
</head>
<body>
<div>
<input type="button" id="myButton" value="Нажми меня" />
</div>
</body>
</html>
```

Получение дополнительной информации о событии

Часто обработчик не нуждается в дополнительных сведениях о событии, которое его запустило. Но если нужно больше деталей о элементе, который сгенерировал событие, можно использовать ключевое слово **this** как уже было описано ранее. Однако, jQuery может предоставить большее количество информации о событии, если в этом есть необходимость.

Чтобы получить больше сведений о событии, обработчик должен принимать объект jQuery event в качестве аргумента. В таком случае обработчик события может получить доступ к дополнительной информации с помощью различных методов и свойств данного объекта.

Следующий пример не только выводит сообщение, когда нажата кнопка, но и указывает точную дату и время, когда произошло событие, а также выводит координаты X и Y указателя мыши в это время:

Пример:

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

$( init );

function init() {
  $('#myButton').bind( 'click', displayInfo );
}

function displayInfo( clickEvent ) {
  var clickTime = new Date( clickEvent.timeStamp );
  var clickTimeString = clickTime.toDateString() + " в " +
clickTime.toTimeString();
  var mouseCoords = "(" + clickEvent.pageX + "," + clickEvent.pageY
+ ")";
  alert( "Всем - привет! Вы нажали кнопку " + clickTimeString +
  ".\n\nКоординаты указателя мыши были : " + mouseCoords + "." );
}
</script>
</head>
<body>
<div>
<input type="button" id="myButton" value="Нажми меня" />
</div>
</body>
```

Вышеприведённый скрипт использует три свойства 3 объекта **event**:

- timeStamp - время, когда произошло событие;
- pageX - координата X указателя мыши в момент нажатия на кнопку, по отношению к левому верхнему углу документа;
- pageY - координата Y указателя мыши в момент нажатия на кнопку, по отношению к левому верхнему углу документа.

Остановка действий по умолчанию и распространение событий

Многие события имеют действия по умолчанию. Например, если пользователь нажмет на ссылку, то по умолчанию событие **click** для ссылки открывает ссылку.

Распространение событий. Также, известно как событие происходит тогда, когда происходит привязка обработчика к одному и тому же событию для элемента и его родителя. Например, привязка обработчика к событию **click** для ссылки, и привязка другого обработчика к событию **click** для параграфа, который содержит ссылку. Когда пользователь нажимает ссылку, обработчик события **click** ссылки запускается первым, а затем событие "всплывает" к родительскому параграфу, вызывая срабатывание его обработчика события **click**.

Часто нужно предотвращать запуск действий по умолчанию и/или распространение событий. Например, если установлен обработчик события **click** для ссылки, который нужен для проверки правильности заполнения формы перед тем, как покинуть страницу, то нужно предотвратить переход по ссылке, который выполняется по умолчанию.

Или если для родительского элемента и его потомков установлены одинаковые обработчики события **click** может понадобиться предотвратить запуск всех функций в одно и тоже время.

Объект jQuery **event** даёт пару методов для того, чтобы остановить такое поведение программы:

1. preventDefault(). Останавливает выполнение действий по умолчаниюstop
2. Propagation(). Останавливает «всплытие» события к родительскому элементу

Следующий пример открывает ссылку в новом окне, когда на неё нажимают, и использует метод **preventDefault()** чтобы остановить открытие ссылки в текущем окне.

Пример:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

$( init );

function init() {
  $('#myLink').bind( 'click', openInWindow );
}

function openInWindow( clickEvent ) {
  window.open( this.href, '', 'width=500,height=400' );
  clickEvent.preventDefault();
}
</script>
</head>
<body>
<div>
<a id="myLink" href="http://www.example.com/">Нажми меня</a>
</div>
</body>
</html>
```

Сброс обработчика события

Если есть необходимость удалить обработчик события из элемента, следует вызвать метод **unbind()**.

Синтаксис:

```
$(selector).unbind();
```

Для удаления обработчика заданного события:

Синтаксис:

```
$(selector).unbind( ИмяСобытия );
```

Принудительная генерация события

Иногда нужно запускать события для элементов прямо из кода. Например, нужно отправить форму автоматически, когда пользователь нажмёт на ссылку.

Метод jQuery **trigger()** генерирует определённое событие для выбранного элемента (или элементов). Имя события нужно передать **trigger()** в триггер в качестве аргумента.

В примере происходит отправка формы, когда пользователь нажимает на ссылку на странице:

Пример:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">

$( init );

function init() {
  $('#myLink').bind( 'click', sendForm );
}

function sendForm() {
  $('#myForm').trigger('submit');
  return false;
}
</script>
</head>
<body>
<form id="myForm" method="post" action="handler.php">
<div>
<label for="email">Адрес email:</label>
<input type="text" name="email" />
<a id="myLink" href="#">Отправить</a>
</div>
</form>
</body>
</html>
```

Установка событий через метод ready()

Альтернативный подход для работы с событиями определяется следующим алгоритмом:

1. На найденный элемент устанавливается метод обработки;
2. Код п.1 необходимо внести в анонимную функцию – функцию, не имеющую имени и использующуюся для представления нескольких действий как одного целого при помещении внутрь другого метода;
3. Анонимную функцию п.2 необходимо внести внутрь события
4. Код п.3. необходимо внести в анонимную функцию для внесения в конструкцию $(document).ready() – чтобы браузер ожидал событие и выполнял код после загрузки страницы.

Синтаксис:

```
$( document ).ready(function()
   {
          $( "что вызвало событие" ).событие(function()
      {
            $( "для чего применить" ).действие();
      });
   });
```

ПРИМЕР РАБОТЫ С ФОРМОЙ СРЕДСТВАМИ JQUERY

Данный материал иллюстрирует процесс получение значения из формы на jQuery, вводимые пользователем. Пример иллюстрирует работу простого калькулятора. У него будет два текстовых поля для ввода чисел, поле Submit для запуска функции сложения и параграф для вывода результата.

1) HTML разметка

Все поля формы снабжены идентификаторами, которые привязаны к меткам. Чтобы изобразить кнопку, необязательно использовать тег **button**. Поле **input type="submit"** работает аналогичным образом.

```
<form id="summa">
        <label for="numberOne">Число 1</label><br>
        <input type="text" id="numberOne" autofocus>
        <br>
        <label for="numberTwo">Число 2</label><br>
        <input type="text" id="numberTwo">
        <br><br>
        <input type="submit" value="Сумма">
        <p><b>Сумма: <span id="summaResult"
         class="summaResult"></span></b></p>
</form>
```

2) Подключение библиотеки jQuery

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

3) Код скрипта калькулятора на jQuery

Необходимо внести скрипт в анонимную функцию, отслеживающую готовность документа для работы. Когда все HTML теги построятся, то скрипт начнет выполняться.

```
$(document).ready(function(){

...

});
```

Далее следует отловить событие отправки формы (клик по кнопке). Для этого необходимо указать id формы и событие "submit".

```
$('#summa').on('submit', function(event){
    event.preventDefault();
});
```

После клика по кнопке (во время отправки формы) браузер перезагружается. Это стандартное поведение для данного элемента. Перезагрузку браузера нужно отменить. Для этого необходимо передать в функцию параметр, принимающий в себя **event. Event** - это событие, которое произошло – в данном случае было событие "**submit**".

Внутри функции, следует обратиться к **event** и далее через точку указать его метод **preventDefault()**, который отменяет стандартное поведение формы в браузере.

Далее необходимо получить значения полей, вычислить их сумму и ниже вывести результат суммы. Запись происходит в переменные идентификаторы полей для ввода чисел. Метод val() вернет значения полей, которые введет пользователь.

```
var numberOne = $('#numberOne').val();
  numberTwo = $('#numberTwo').val();
```

По умолчанию, все вводимые числа пользователями, являются строками. А это значит, что произойдет не математическое действие сложения, а конкатенация строк. Метод parseInt преобразует содержимое переменных в целые числа.

```
numberOne = parseInt(numberOne);
numberTwo = parseInt(numberTwo);
```

Далее необходимо записать результат сложения двух переменных в переменную **summa**.

```
summa = numberOne + numberTwo;
```

Для вывода результата суммы в теге **span c id="summaResult"** следует использовать метод **text()**, передав в параметре результат сложения.

```
$('#summaResult').text(summa);
```

Однако, если не все поля будут заполнены, то сумма не посчитается и выведется NaN. Чтобы такого не произошло, следует проверить, все ли поля заполнены. Необходимо создать условие, если первое поле оказалось без числа (isNAN), то передать ему значение 0. Такую же проверку сделать и для второго поля.

```
if (isNaN(numberOne)){
  numberOne = 0;
};
if (isNaN(numberTwo)){
  numberTwo = 0;
};
```

ПРИЛОЖЕНИЕ

Приложение А. Браузерные события JavaScript

Событие	Описание
onclick	реагирует на клик мыши;
ondblclick	реагирует на двойной клик;
oncontextmenu	реагирует на клик правой кнопкой мыши;
onmouseover	реагирует на наведение мыши (на дочерние элементы тоже);
onmouseenter	реагирует на наведение мыши;
onmouseout	реагирует на отведение мыши (на дочерние элементы тоже);
onmouseleave	реагирует на отведение мыши;
onmousedown	реагирует на зажатие кнопки мыши;
onmouseup	реагирует на отпускание кнопки мыши;
onmousemove	реагирует при движении указателя мыши над элементов;
onwheel	реагирует при движении колёсика мыши над элементом;
altKey	реагирует на нажатие кнопки мыши при нажатой клавиши ALT;
ctrlKey	реагирует на нажатие кнопки мыши при нажатой клавиши CTRL;
shiftKey	реагирует на нажатие кнопки мыши при нажатой клавиши SHIFT;
metaKey	реагирует на нажатие кнопки мыши при нажатой клавиши META(◆,⌘);
button	возвращает номер нажатой клавиши мыши (0,1,2);
buttons	возвращает номер нажатой клавиши мыши (1,2,4,8,16);
which	возвращает номер нажатой клавиши мыши (1,2,3);
clientX	возвращает координату указателя мыши по оси X (относительно окна);
clientY	возвращает координату указателя мыши по оси Y (относительно окна);
detail	возвращает количество кликов по объекту;
relatedTarget	возвращает родственный элемент;

screenX	возвращает координату указателя мыши по оси X (относительно экрана);
screenY	возвращает координату указателя мыши по оси Y (относительно экрана);
deltaX	возвращает количество скроллов по оси X;
deltaY	возвращает количество скроллов по оси Y;
deltaZ	возвращает количество скроллов по оси Z;
deltaMode	возвращает единицу измерения длины скролла;

Таблица 2. События мыши

Событие	Описание
onkeydown	реагирует на нажатие клавиши;
onkeypress	реагирует на нажатие клавиши;
onkeyup	реагирует на отпускание клавиши;
altKey	реагирует на нажатие клавиши ALT;
ctrlKey	реагирует на нажатие клавиши CTRL;
shiftKey	реагирует на нажатие клавиши SHIFT;
metaKey	реагирует на нажатие клавиши META(◆,⌘);
key	возвращает значение нажатой клавиши;
keyCode	возвращает Unicode нажатой клавиши;
which	возвращает Unicode нажатой клавиши;
charCode	возвращает Unicode нажатой клавиши;
location	возвращает код группы клавиш (цифры, буквы, ...) (0,1,2,3)

Таблица 3. События клавиатуры

Событие	Описание
onabort	срабатывает когда загрузка ресурса была отклонена;
onbeforeunload	срабатывает во время загрузки страницы;

onerror	срабатывает в случае ошибки загрузки ресурса;
onhashchange	срабатывает при изменении якоря на странице;
onload	срабатывает после загрузки объекта;
onpageshow	срабатывает после каждой загрузки страницы;
onpagehide	срабатывает когда пользователь уходит со страницы;
onresize	срабатывает когда размер страницы был изменен;
onscroll	срабатывает когда скроллбар элемента был передвинут;
onunload	срабатывает однажды после выгрузки страницы.

Таблица 4. События объектов и ресурсов

Событие	Описание
onblur	срабатывает когда элемент теряет фокус;
onfocus	срабатывает когда элемент получает фокус;
onfocusin	тоже, что и *onfocus*, но и для дочерних элементов;
onfocusout	тоже, что и *onblur*, но и для дочерних элементов;
onchange	срабатывает когда содержимое элемента было изменено;
oninput	срабатывает когда *input* получает ввод;
oninvalid	срабатывает когда содержимое элемента неверно;
onreset	срабатывает после сброса форма;
onsearch	срабатывает при вводе в поле с типом *search;*
onselect	срабатывает после выделения текста в элементе;
onsubmit	срабатывает после отправки формы.

Таблица 5. События форм и их элементов

Событие	Описание
ondrag	срабатывает при переносе элемента;

ondragend	срабатывает после переноса элемента;
ondragenter	срабатывает когда элемент находится в целевой drop зоне;
ondragleave	срабатывает когда элемент покидает drop зону;
ondragover	срабатывает когда элемент находится над drop зоной;
ondragstart	срабатывает при начале перетаскивания;
ondrop	срабатывает после отпускания элемента в drop зону.

Таблица 6. События Grag and Grop

Событие	Описание
oncopy	срабатывает при копировании содержимого элемента;
oncut	срабатывает при вырезании содержимого элемента;
onpaste	срабатывает при вставке данных в элемент.

Таблица 7. События буфера обмена

Событие	Описание
onafterprint	срабатывает после начала печати страницы;
onbeforeprint	срабатывает перед началом печати страницы.

Таблица 8. События печати

Событие	Описание
onabort	срабатывает при ошибке загрузки медиа;
oncanplay	срабатывает после полной буферизации медиа;
oncanplaythrough	срабатывает если браузер готов к воспроизведению медиа;
ondurationchange	срабатывает если продолжительность медиа была изменена;
onemptied	срабатывает если медиа не доступно;
onended	срабатывает когда произведение медиа закончилось;
onerror	срабатывает при ошибке загрузки медиа;

onloadeddata	срабатывает после загрузки данных медиа;
onloadedmetadata	срабатывает после загрузки мета-данных медиа;
onloadstart	срабатывает после начала загрузки медиа;
onpause	срабатывает при включении паузы;
onplay	срабатывает когда медиа начало воспроизводится;
onplaying	срабатывает в момент воспроизведения медиа;
onprogress	срабатывает в момент скачивания медиа;
onratechange	срабатывает при изменении скорости воспроизведения;
onseeked	срабатывает при изменении положения начала воспроизведения;
onseeking	срабатывает в момент изменения положения начала воспроизведения;
onstalled	срабатывает при неполучении данных о медиа;
onsuspend	срабатывает при нарочном неполучении данных о медиа;
ontimeupdate	срабатывает при изменении положения начала воспроизведения;
onvolumechange	срабатывает при изменении громкости медиа;
onwaiting	срабатывает при включении паузы для буфферизации.

Таблица 9. Медиа события

Событие	Описание
animationstart	срабатывает когда анимация началась;
animationend	срабатывает когда анимация закончилась;
animationiteration	срабатывает когда анимация повторяется;
animationName	возвращает имя анимации;
elapsedTime	возвращает время воспроизведения анимации;
propertyName	возвращает имя анимированого CSS свойства;
elapsedTime	возвращает время воспроизведения перехода (transition);

Таблица 10. События анимации

Событие	Описание
transitionend	срабатывает когда плавный переход закончился.

Таблица 11. События плавного перехода

Событие	Описание
onerror	срабатывает когда возникает ошибка в источнике события;
onmessage	срабатывает когда сообщение получено через источник события;
onopen	срабатывает когда соединение с источником события открыто.

Таблица 12. События сервера

Событие	Описание
ononline	срабатывает когда браузер находится online;
onoffline	срабатывает когда браузер находится offline;
onstorage	срабатывает когда Web Storage был обновлен;
onshow	срабатывает когда <*menu*> отражено как контекстное меню;
ontoggle	срабатывает при нажатии на элемент <*details*>;
onpopstate	срабатывает при изменении истории окна.

Таблица 13. События браузера

Событие	Описание
ontouchstart	срабатывает при касании экрана;
ontouchmove	срабатывает при перемещения касания;
ontouchend	срабатывает при прекращении касания
ontouchcancel	срабатывает когда касание прервано.

Таблица 14. События сенсорных экранов

Событие	Описание
bubbles	показывает является ли событие - *bubbles* событием;
cancelable	показывает является ли событие - *cancelable* событием;
currentTarget	возвращает элемент, событие которого было вызвано;

defaultPrevented	показывает был вызван метод *preventDefault()* для события;
eventPhase	возвращает текущую фазу потока события;
isTrusted	показывает является ли событие - *trusted* событием;
target	возвращает элемент, который вызвал событие;
timeStamp	возвращает время с момента срабатывания события;
type	возвращает имя события элемента;
view	возвращает ссылку объекту Window, где произошло событие;
preventDefault()	предотвращает реагирование объекта на событие;
stopImmediatePropagation()	предотвращает реагирование на прослушивание объекта
stopPropagation()	предотвращает реагирование объекта на дальнейшие события.

Таблица 15. События объектов

Событие:	**Описание:**
newURL	возвращает URL страницы после изменения хэша;
oldURL	возвращает URL страницы до изменения хэша;

Таблица 16. События хэша

Событие:	**Описание**
persisted	реагирует на кэширование страницы

Таблица 17. События кэша

Приложение В. Браузерные события jQuery

Метод	Описание
.keydown()	**Привязывает JavaScript обработчик событий "**keydown**" (нажатие на любую клавишу клавиатуры), или запускает это событие на выбранный элемент.**
.keypress()	**Привязывает JavaScript обработчик событий "**keypress**" (нажатие на любую клавишу клавиатуры кроме специальных - Alt, Ctrl, Shift, Esc, PrScr и тому подобные), или запускает это событие на выбранный элемент.**
.keyup()	**Привязывает JavaScript обработчик событий "**keyup**" (нажатая клавиша была отпущена), или запускает это событие на выбранный элемент.**

Таблица 18. События клавиатуры

Метод	Описание
.click()	Привязывает *JavaScript* обработчик событий "**click**" (клик левой кнопкой мыши), или запускает это событие на выбранный элемент.
.contextmenu()	Привязывает *JavaScript* обработчик событий "**contextmenu**" (вызов контекстного меню на элементе - клик правой кнопкой мыши), или запускает это событие на выбранный элемент.
.dblclick()	Привязывает *JavaScript* обработчик событий "**dblclick**" (двойной клик левой кнопкой мыши), или запускает это событие на выбранный элемент.
.hover()	Связывает один или два обработчика событий для элемента, которые будут выполнены, когда указатель мыши находится на элементе и при его отведении.
.mousedown()	Привязывает *JavaScript* обработчик событий "**mousedown**" (нажатие кнопки мыши на элементе), или запускает это событие на выбранный элемент.
.mouseenter()	Привязывает *JavaScript* обработчик событий "**mouseenter**" (срабатывает, когда указатель мыши заходит на элемент), или запускает это событие на выбранный элемент.

.mouseleave()	Привязывает *JavaScript* обработчик событий "**mouseleave**" (срабатывает, когда указатель мыши выходит из элемента), или запускает это событие на выбранный элемент.
.mousemove()	Привязывает *JavaScript* обработчик событий "**mousemove**" (срабатывает, когда указатель мыши перемещается внутри элемента), или запускает это событие на выбранный элемент.
.mouseout()	Привязывает *JavaScript* обработчик событий "**mouseout**" (срабатывает, когда указатель мыши покидает элемент), или запускает это событие на выбранный элемент.
.mouseover()	Привязывает *JavaScript* обработчик событий "**mouseover**" (срабатывает, когда указатель мыши входит в элемент), или запускает это событие на выбранный элемент.
.mouseup()	Привязывает *JavaScript* обработчик событий "**mouseup**" (срабатывает, когда указатель мыши находится над элементом и кнопка мыши отпущена), или запускает это событие на выбранный элемент.

Таблица 19. События мыши

Метод	Описание
.resize()	Привязывает *JavaScript* обработчик событий "**resize**" (срабатывает при изменении размеров окна браузера), или запускает это событие на выбранный элемент.
.scroll()	Привязывает *JavaScript* обработчик событий "**scroll**" (срабатывает при прокрутке элементов), или запускает это событие на выбранный элемент.

Таблица 20. События браузера

Метод	Описание
.blur()	Привязывает *JavaScript* обработчик событий "**blur**" (потеря фокуса элементом), или запускает это событие на выбранный элемент.
.change()	Привязывает *JavaScript* обработчик событий "**change**" (изменение элемента), или запускает это событие на выбранный элемент. Метод используется с элементами HTML формы.

.focus()	Привязывает *JavaScript* обработчик событий "**focus**" (получение фокуса элементом), или запускает это событие на выбранный элемент.
.focusin()	Привязывает *JavaScript* обработчик событий "**focusin**" (получение фокуса элементом, или любым вложенным элементом), или запускает это событие на выбранный элемент. Метод поддерживает всплывающие события (*event bubbling*).
.focusout()	Привязывает *JavaScript* обработчик событий "**focusout**" (потеря фокуса элементом, или любым вложенным элементом), или запускает это событие на выбранный элемент. Метод поддерживает всплывающие события (*event bubbling*).
.select()	Привязывает *JavaScript* обработчик событий "**select**" (срабатывает при выделении текста), или запускает это событие на выбранный элемент. Метод используется с элементом **<input>** (с текстовым типом **type = "text"**) и элементом **<textarea>**.
.submit()	Привязывает *JavaScript* обработчик событий "**submit**" (обработчик отправки формы), или запускает это событие на выбранный элемент.

Таблица 21. События формы

Метод	Описание
$.proxy()	Принимает функцию и возвращает новую, которая всегда будет иметь определенный контекст выполнения.
.off()	Позволяет удалить обработчик, или обработчики событий, присоединенные методом **.on()**.
.on()	Присоединяет для выбранных элементов функцию обработчика для одного или нескольких событий.
.one()	Присоединяет для выбранных элементов функцию обработчика, которая будет выполнена не более одного раза к каждому элементу по каждому типу событий.
.trigger()	Позволяет выполнить все функции обработчики событий, присоединенные у выбранного элемента для данного типа события.

.triggerHandler()	Позволяет вызвать все функции обработчики событий, присоединенные у выбранного элемента по указанному типу событий без вызова самого события.

Таблица 22. Присоединяемые события

Свойство / Метод	Описание
event.currentTarget	Определяет текущий элемент DOM, в котором в настоящий момент обрабатывается событие.
event.data	Необязательный объект данных, передаваемый методу события, когда текущий обработчик события привязывается.
event.delegateTarget	Возвращает элемент, к которому был прикреплен вызванный в данный момент обработчик события. Это свойство будет отличаться от свойства **event.currentTarget** только в том случае, если обработчик события делегирован, а не на прямую привязан к элементу.
event.isDefaultPrevented()	Метод возвращает логическое значение **true**, если для этого объекта событий вызывался метод **event.preventDefault()** и **false** в обратном случае.
event.isImmediatePropagationStopped()	Метод возвращает логическое значение **true**, если для этого объекта событий вызывался метод **event.stopImmediatePropagation()** и **false** в обратном случае.
event.isPropagationStopped()	Метод возвращает логическое значение **true**, если для этого объекта событий вызывался метод **event.stopPropagation()** и **false** в обратном случае.
event.metaKey	Содержит логическое значение, которое указывает на то, была ли нажата, или нет мета клавиша **Cmd** (Mac) / **Windows** (Windows), когда событие сработало.

event.namespace	Соответствует пользовательскому пространству имён, определенному при срабатывании события (строковое значение).
event.pageX	Позиция курсора мыши относительно левого края документа, или элемента.
event.pageY	Позиция курсора мыши относительно верхнего края документа, или элемента.
event.preventDefault()	Если этот метод вызывается, то действие события по умолчанию не будет срабатывать (отменяет действие события по умолчанию). Событие продолжает распространяться как обычно, только с тем исключением, что событие ничего не делает.
event.relatedTarget	Соответствует другому элементу DOM, который участвует в событии, если таковой имеется.
event.result	Содержит последнее значение, возвращаемое обработчиком события, которое было вызвано этим событием (если значение не было равно **undefined**).
event.stopImmediatePropagation()	Прекращает дальнейшую передачу текущего события (предотвращает всплытие по дереву DOM) и останавливает цепочку вызова событий для текущего элемента.
event.stopPropagation()	Прекращает дальнейшую передачу текущего события (предотвращает всплытие по дереву DOM).
event.target	DOM элемент, который инициировал событие.
event.timeStamp	Разница в миллисекундах между тем моментом, когда браузер создал событие и полуночи *1 января 1970 года* (**Unix-время**).
event.type	Описывает тип события, которое было вызвано.
event.which	В зависимости от типа события свойство указывает на определенную клавишу клавиатуры

	или кнопку мыши, которая была нажата пользователем.

Таблица 23. Свойства и методы объекта Even

ЛИТЕРАТУРА

1. Никсон. Р. Создаем динамические веб-сайты с помощью PHP, mySOL, JavaScript, CSS и HTML 4-е изд. – Питер, - 2016.
2. Браун Э. Изучаем JavaScript. – Диалектика, - 2019.
3. Флэнган Д. JavaScript. – Вильямс, - 2018.
4. Пьюривал Сэмми. Основы разработки веб-приложений. – Питер, -1019.
5. Вуд К. Расширение библиотеки jQuery. – ДМК, - 2017.
6. Закас Н. Оптимизация производительности. – Символ, - 2019.
7. Бибо Б., Кац И. jQuery. – Символ, - 2018.

СВЕДЕНИЯ ОБ АВТОРАХ

Кашкин Евгений Владимирович, к.т.н., доцент кафедры «Интеллектуальные системы информационной безопасности» Института комплексной безопасности и специального приборостроения Российского технологического университета (МИРЭА).

Антонова Ирина Игоревна, к.т.н., доцент, доцент кафедры «Интеллектуальные системы информационной безопасности» Института комплексной безопасности и специального приборостроения Российского технологического университета (МИРЭА).