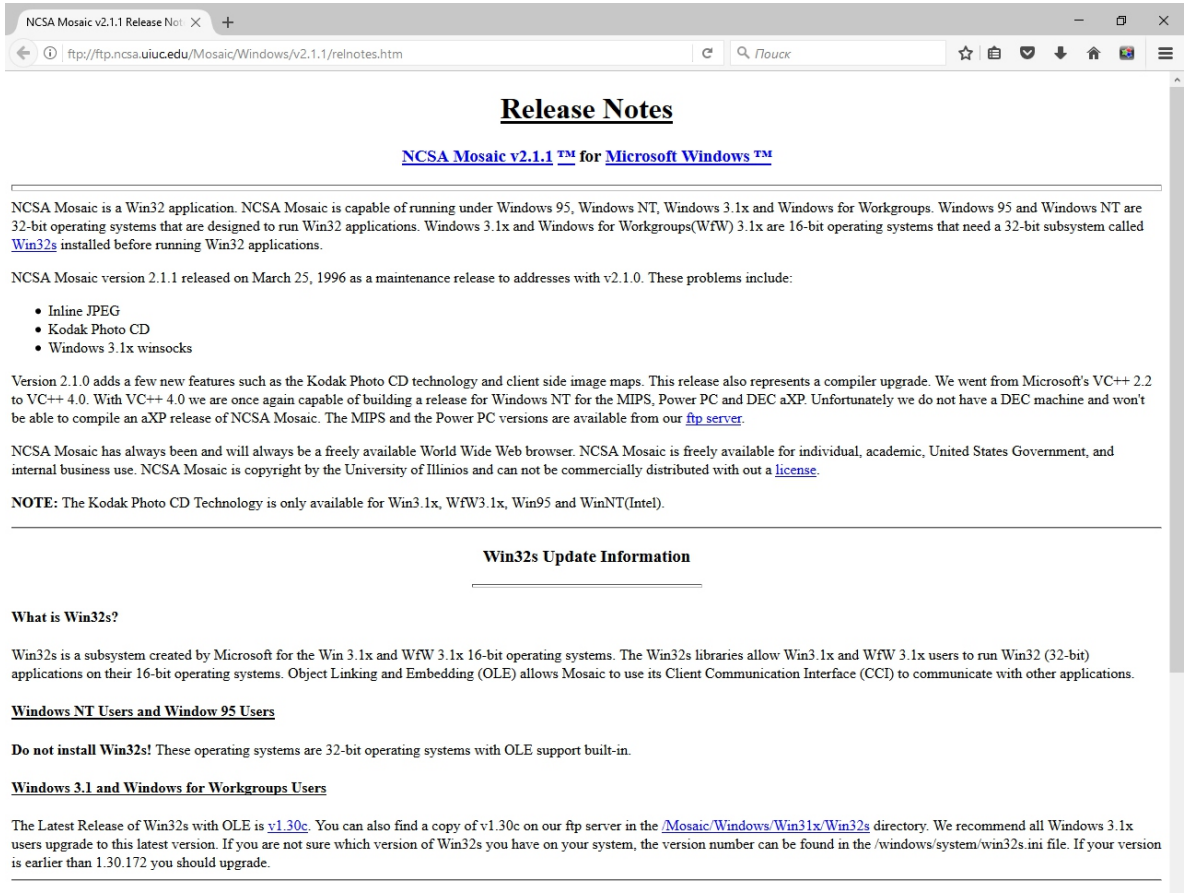
Модуль 1 «Основы компьютерной геометрии» Лекция 3 «Введение в JavaScript (отличия от C)»

к.ф.-м.н., доц. каф. ФН-11, Захаров Андрей Алексеевич,
ауд.:930а(УЛК)
моб.: 8-910-461-70-04,
email: azaharov@bmstu.ru

МГТУ им. Н.Э. Баумана

2 октября 2017 г.

Жизнь до JavaScript



Ещё совсем недавно Всемирная паутина казалась очень скучным местом. Составленные из одного лишь HTML-текста веб-страницы отображали информацию и ничего больше. Можно было лишь перейти по ссылке и ждать загрузки новой веб-страницы — вот что считалось настоящей интерактивностью.

История JavaScript

Проект JavaScript стартовал в компании Netscape в 1995 году. Брендан Эйх, создатель языка JavaScript, заимствовал все лучшее из ряда других языков программирования. Результат представлял собой своеобразный язык наподобие эсперанто, который обманчиво воспринимался людьми, имеющими опыт работы с другими языками программирования, как язык, с которым они уже знакомы.

Язык JavaScript первоначально назывался Mocha. С выпуском первой бета-версии браузера Netscape Navigator он был переименован в LiveScript, но в версию браузера Netscape 2, выпущенную в 1995 г., он был встроен уже как язык JavaScript.

Компания Netscape представила язык JavaScript в организацию Ecma International, занимающуюся стандартизацией информационных и коммуникационных технологий, где он был утвержден в качестве стандарта ECMAScript в 1997 г. Брендан Эйх высказал по поводу названия стандартизированного языка нашумевшее замечание, в котором охарактеризовал ECMAScript как «неудачное название торговой марки, звучащее подобно названию кожного заболевания».

Название JavaScript, которое было дано языку компанией Netscape и используется большинством тех, кто с ним работает, тоже нельзя признать удачным. Несмотря на определённое сходство, в действительности Java и JavaScript — это два совершенно разных языка.

Развитие JavaScript

Сразу же после своего дебюта язык JavaScript быстро приобрёл популярность в качестве средства, позволяющего делать веб-страницы более динамичными. Одним из первых результатов встраивания JavaScript в браузеры стало появление так называемого динамического HTML (DHTML) — языка разметки веб-страниц, обеспечившего реализацию не только всевозможных визуальных эффектов, но и более полезных вещей, например раскрывающихся меню и средств валидации (проверки корректности данных) веб-страниц.

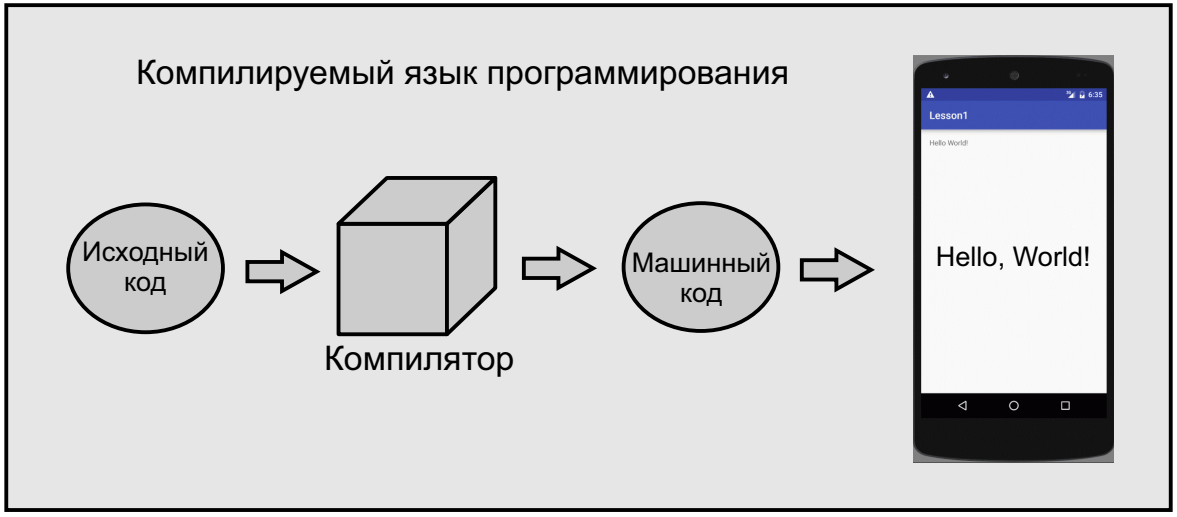
Сейчас JavaScript стал наиболее широко используемым языком программирования в мире, и практически на каждом из эксплуатируемых в настоящее время персональных компьютеров установлен по крайней мере один браузер, способный выполнять JavaScript-код.

Язык JavaScript достаточно гибок для того, чтобы быть пригодным для использования непрограммистами, и одновременно располагает достаточно мощными возможностями, которых профессиональным программистам вполне хватает для того, чтобы поддерживать функциональность практически любого современного сайта в Интернете, начиная с одностраничных сайтов и кончая такими «монстрами», как сайты Google, Amazon, Facebook и многие другие.

Компьютерные программы

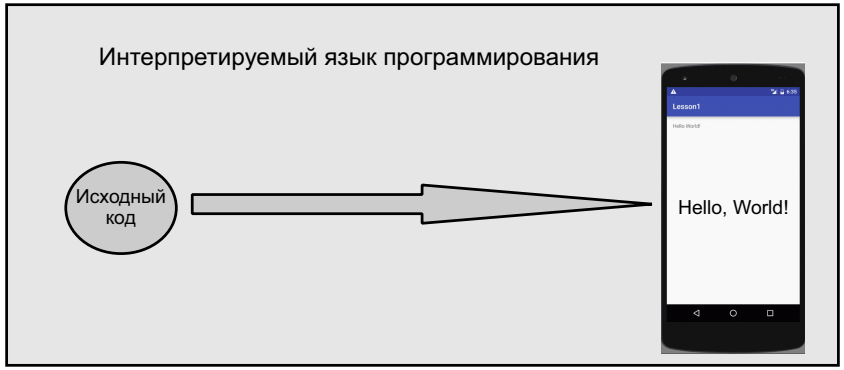
Компьютерные программы — это наборы инструкций, с помощью которых можно заставить компьютер выполнять определённые действия. Каждый компьютерный язык программирования располагает собственным набором инструкций и правилами их записи, которыми должны руководствоваться программисты. Компьютер неспособен понимать эти инструкции в непосредственном виде. Чтобы компьютер мог понимать язык программирования, требуется предварительный процесс преобразования инструкций, в ходе которого они транслируются (переводятся) из формы, удобной для чтения (и записи) человеком, в машинный язык. В зависимости от того, когда именно осуществляется процесс трансляции инструкций, языки программирования делятся на два основных типа: компилируемые и интерпретируемые.

Компилируемые языки программирования



Компилируемые языки программирования — это языки, которые требуют, чтобы написанный программистом код был обработан специальной программой, так называемым *компилятором*, который анализирует данный код, а затем преобразует его в машинный язык. После этого компьютер может выполнить скомпилированную программу.
В качестве примера компилируемых языков программирования можно привести C, C++, Fortran, Java, Objective-C и COBOL.

Интерпретируемые языки программирования



Интерпретируемые языки программирования — это языки, которые также компилируются компьютером в машинный язык, но процесс компиляции осуществляется в браузере пользователя непосредственно во время выполнения программы.
Плюсы: изменения могут быть внесены в программу в любой момент.
Минусы: компиляция кода во время его выполнения может замедлять выполнение программ. Отчасти именно из-за такого снижения производительности интерпретируемые языки получили репутацию не очень серьезных языков программирования. Однако с появлением улучшенных компиляторов, осуществляющих компиляцию кода на стадии его выполнения (так называемые *JIT-компиляторы* («just-in-time») или *оперативные компиляторы*), и процессоров с повышенным быстродействием такая точка зрения очень быстро устарела.
Примерами интерпретируемых языков программирования могут служить PHP, Perl, Haskell, Ruby, JavaScript.

Динамический язык сценариев

JavaScript часто описывают как *динамический сценарный язык*, т.е. язык для написания сценариев (жарг. *скриптовый язык*).
Чаще всего программы на JavaScript выполняются в браузерах.
Существуют три способа выполнения JavaScript в браузере, каждый из которых рассматривается в последующих разделах:

- поместить код непосредственно в атрибут события HTML-элемента:
  ```
  <button id="bigButton" onclick="alert('Привет,
                    мир!');">Щелкните здесь</button>
  ```
  В данном случае, когда пользователь щелкает на кнопке, созданной этим HTML-элементом, на экране появляется всплывающее окно, в котором отображается текст "Привет, мир!";
- поместить код между открывающим и закрывающим тегами `script`:
  ```
  <script>
      Сюда вставляется код JavaScript
  </script>
  ```
- поместить код в отдельный документ, который включается в HTML:
  ```
  <script src="myScript.js"></script>
  ```

Преимущества использования внешних файлов JavaScript:

- улучшается удобочитаемость HTML-файлов;
- упрощается сопровождение кода, поскольку вносить изменения или исправлять ошибки приходится только в одном месте.

Атрибуты событий элементов HTML

В HTML предусмотрено несколько (около 70) специальных атрибутов, предназначенных для запуска JavaScript при наступлении определённых событий в браузере или при выполнении пользователем определённых действий. Наиболее часто используемые из них приведены в табл.:

Атрибут	**Условие запуска сценария**
`onload`	Окончание загрузки страницы
`onfocus`	Получение элементом фокуса ввода (например, при активизации текстового поля)
`onblur`	Потеря элементом фокуса ввода (например, в результате выполнения пользователем щелчка в другом текстовом поле)
`onchange`	Изменение значения элемента
`onselect`	Выделение текста
`onsubmit`	Отправка формы
`onkeydown`	Нажатие клавиши
`onkeypress`	Нажатие и последующее отпускание клавиши
`onkeyup`	Отпускание клавиши
`onclick`	Щелчок мышью на элементе
`ondrag`	Перетаскивание элемента
`ondrop`	Вставка перетаскиваемого элемента
`onmouseover`	Перемещение указателя мыши над элементом

Объявление переменных

В JavaScript переменные можно создавать одним из двух способов:

С использованием ключевого слова `var` :

```
var myName;
```

Переменной, объявляемой с помощью ключевого слова `var`, можно присвоить начальное значение в момент её создания (это называется инициализацией переменной).

```
var myName = "Chris"
```

Без использования ключевого слова `var` :

```
myName = "Chris";
```

Переменная, созданная без использования ключевого слова `var`, становится глобальной переменной. Использование таких глобальных переменных может приводить к возникновению проблем в работе программы, которые трудно обнаружить и исправить. Поэтому рекомендуется полностью отказаться от практики объявления переменных без использования ключевого слова `var`. Если возникает необходимость в создании глобальной переменной, её также лучше всего объявлять с помощью ключевого слова `var`, рекомендуется всегда придерживаться этого правила.

Зарезервированные слова

Некоторые слова нельзя использовать в качестве имен переменных. Ниже приведен перечень зарезервированных слов, которые не могут выступать в качестве названий переменных, функций, методов, меток циклов и объектов.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Создание констант с помощью ключевого слова const

Константы объявляются с использованием ключевого слова const:

```
const heightOfTheEmpireStateBuilding = 1454;
const speedOfLight = 299792458;
const numberOfProblems = 99;
const meanNumberofBooksReadIn2014 = 12;
```

Типы данных

JavaScript относится к категории так называемых слабо типизированных языков. Это означает, что не требуется сообщать JavaScript или даже знать самому, будет ли создаваемая переменная содержать текст, число или какой-либо другой тип данных. JavaScript автоматически выполняет всю работу по определению типов данных, хранящихся в переменных. JavaScript распознает пять основных или, как говорят, примитивных (элементарных) типов данных. Для сравнения, в языке программирования C++ насчитывается по крайней мере 12 различных типов данных. Перечислим основные типы данных языка JavaScript:

- `undefined` — если значение не определено. При создании переменной JavaScript, если не присвоить ей какое-либо значение, то все равно она всегда будет иметь значение по умолчанию `"undefined"`.
- `number` — целые числа или числа с плавающей точкой (дробные числа);
- `string` — строки;
- `boolean` — логический тип данных. Может содержать значения `true` (истина) или `false` (ложь);
- `function` — ссылка на функцию;
- `object` — массивы, объекты, а также переменная со значением `null` (ссылка на пустой объект). Данный тип не относится к примитивным (элементарным) типам данных.

Числовой тип данных

В JavaScript числа хранятся в виде 64-разрядных значений с плавающей точкой. В переводе на обычный язык это означает, что их значения могут меняться в пределах от $\pm 5 \times 10^{-324}$ до $\pm 1.7976931348623157 \times 10^{308}$. Любое число может быть записано как с десятичной точкой, так и без неё. В отличие от большинства других языков программирования, в JavaScript отсутствуют отдельные типы данных для целых чисел (положительных и отрицательных чисел без дробной части) и чисел с плавающей точкой.

Строковый тип данных

Строковая переменная создаётся путём заключения значения в одинарные или двойные кавычки:

```
var myString = "Привет, я строка.";
```

Не имеет никакого значения, какие кавычки используются — одинарные или двойные, лишь бы открывающая и закрывающая кавычки были одного типа.
Для строк поддерживается оператор конкатенации строк +:

```
var Str = "Строка1" + "Строка2";
// Переменная Str будет содержать значение "Строка1Строка2"
```

Часто необходимо сформировать строку, состоящую из имени переменной и её значения. Если написать

```
var X = "Строка1";
var Z = "Значение равно " + X;
```

то переменная Z будет содержать значение "Значение равно Строка1".

NaN

NaN — это сокращение от Not a Number (не число). Этот результат возникает при попытке выполнить математические операции над строками или в тех случаях, когда вычисление приводит к ошибке или не может быть выполнено. Например, невозможно вычислить квадратный корень отрицательного числа. Результатом такой попытки будет NaN.

Основные сведения о массивах

В любом элементе массива могут храниться данные любого типа, в том числе и другие массивы. Кроме того, элементы массивов могут содержать функции и объекты JavaScript.
Имея возможность хранить в массиве данные любого типа, также возможно хранить разные типы данных в разных элементах одного и того же массива:

```
item[0] = "apple";
item[1] = 4+8;
item[2] = 3;
item[3] = item[2] * item[1];
```

Заполнение массивов значениями

Для создания массива JavaScript используется следующая конструкция:

```
var dogNames =["Shaggy", "Tennesee", "Dr. Spock"];
```

Создание пустого массива с последующим добавлением в него элементов осуществляется следующим образом:

```
var peopleList =[];
peopleList[0] = "Chris Minnick";
peopleList[1] = "Eva Holland";
peopleList[2] = "Abraham Lincoln";
```

Не обязательно строго соблюдать последовательность заполнения массива элементами. В этом примере вполне допустимой была бы следующая инструкция:

```
peopleList[99] = "Tina Turner";
```

Такой способ добавления элемента в массив дополнительно сопровождается созданием пустых элементов для всех индексов между `peopleList[2]` и `peopleList[99]`.

Операторы сравнения

Оператор	Описание	Пример
==	Равно	`3 == "3" //true`
!=	Не равно	`3 != "3" //false`
===	Строго равно	`3 === "3" //false`
!==	Строго не равно	`3 !== "3" //true`
>	Больше	`7 > 1 //true`
>=	Больше или равно	`7 >= 7 //true`
<	Меньше	`7 < 10 //true`
<=	Меньше или равно	`2 <= 2 //true`

В чем отличие оператора == (равно) от оператора === (строго равно)? Дело все в том, что если используется оператор ==, интерпретатор пытается преобразовать разные типы данных к одному и лишь затем сравнивает их. Оператор ===, встретив данные разных типов, сразу возвращает `false` (ложь).

Написание функций

Функция — это фрагмент кода JavaScript, который можно вызвать из любого места программы. Функция описывается с помощью ключевого слова `function` по следующей схеме:

```
function <Имя функции> ([<Параметры>]) {
    <Тело функции>
    [return <Значение>]
}
```

или

```
var myFunction = new Function([<Параметры>]) {
    <Тело функции>
    [return <Значение>]
};
```

В определении функции разрешается задавать до 255 параметров. Функция объявляется и определяется в программе или на веб-странице только один раз. Если же вы определите одну и ту же функцию более одного раза, то JavaScript не сообщит об ошибке. В подобных случаях используется та версия функция, которая была определена последней.

Написание функций

Количество аргументов, задаваемых при вызове функции, не обязано совпадать с количеством параметров, указанных в определении этой функции. Если в определении функции содержатся три параметра, но вы вызываете её, задавая всего лишь два аргумента, то третий параметр создаст в функции переменную, имеющую значение `undefined`.
Если желательно, чтобы значения аргументов по умолчанию были отличными от `undefined`, то установите эти значения, например, следующим образом:

```
function welcome(yourName){
    if (typeof yourName === 'undefined'){
        yourName = "друг";
}
```

В следующей версии JavaScript, которая называется ECMAScript 6, будет предоставлена возможность устанавливать для параметров значения по умолчанию в заголовке функции:

```
function welcome(yourName = "друг") {
    document.write("Привет," + yourName);
}
```

Однако ещё не все браузеры могут поддерживать такую возможность.

Функции с произвольным количеством аргументов

Класс массива аргументов позволяет получить доступ ко всем аргументам, переданным функции. Массив доступен только внутри тела функции. Получить доступ к аргументу можно, указав его индекс, а свойство `length` позволяет определить количество аргументов, переданных функции.

```
function f_Sum(x, y) {
    return arguments[0]+arguments[1];
}
document.write(f_Sum(5,6)); //11
```

При использовании массива аргументов можно обработать случай, когда функции передаётся больше аргументов, чем первоначально объявлено. Например, можно просуммировать сразу несколько чисел, а не только два:

```
function f_Sum(x, y) {
    var z = 0;
    for (var i=0, c=arguments.length; i<c; i++) {
        z += arguments[i];
    }
    return z;
}
document.write(f_Sum(5, 6, 7, 20)); // 38
```

Написание функций

Ссылку на функцию можно сохранить в какой-либо переменной. Для этого название функции указывается без круглых скобок:

```
function f_Sum(x, y) (
    return x + y;
}
var s;
s = f_Sum; // Присваиваем ссылку на функцию
s(5,6); // Вызываем функцию f_Sum() через переменную s
```

В заголовке функции имя не является обязательной частью, поэтому можно создавать функции, не имеющие имени(анонимные функции):

```
var s = function(x, y) { // Присваиваем ссылку на анонимную функцию
    return x + y;
};
s(5,6); // Вызываем анонимную функцию через переменную s
```

Анонимные функции, присвоенные переменной, существуют и могут вызываться лишь после выполнения операции присваивания. Доступ же к именованным функциям возможен в любом месте программы.
Значение переменной, которой присвоено значение анонимной функции может быть в любой момент изменено, и ей может быть присвоена другая функция.

Документирование функций с помощью JSDoc

Считается хорошей практикой всегда документировать JavaScript-код с помощью какой-либо стандартной системы. Наиболее широко используемой системой документирования JavaScript, которая стала стандартом де-факто, является система JSDoc.
Язык JSDoc — это простой язык разметки, которая может внедряться в файлы JavaScript. Текущая, 3-я версия JSDoc основана на системе JavaDoc, применяющейся для документирования кода, написанного на языке Java.
Аннотировав свои JavaScript-файлы с помощью JSDoc, можно использовать генератор документации, например jsdoc-toolkit, для создания HTML-файлов, документирующих код.
Разметка JSDoc окружается специальными тегами блочных (многострочных) комментариев. Единственное различие между разметкой JSDoc и обычными блочными комментариями JavaScript состоит в том, что в первом случае разметка начинается символами /** и заканчивается символами */, тогда как в последнем случае вслед за начальной косой чертой указывается лишь один символ `звездочка`. Дополнительная `звездочка` не препятствует использованию разметки JSDoc в качестве обычного блочного комментария, но при необходимости делает её частью генерируемой документации.

Документирование функций с помощью JSDoc

Различные части и разделы программы документируются с использованием тегов JSDoc. Вот список наиболее популярных из них.

Тег JSDoc	Описание
`@author`	Имя разработчика
`@constructor`	Маркирует функцию как конструктор
`@deprecated`	Маркирует метод устаревшим и не рекомендуемым
`@exception`	Описывает исключение, генерируемое методом; синоним тега `@throws`
`@exports`	Указывает член, экспортируемый модулем
`@param`	Описывает параметр метода
`@private`	Обозначает, что метод закрытый
`@return`	Описывает возвращаемое значение; синоним `@returns`
`@returns`	Описывает возвращаемое значение; синоним `@return`
`@see`	Описывает связь с другим объектом
`@this`	Задает тип объекта, на который указывает ключевое слово this в функции
`@throws`	Описывает исключение, генерируемое методом
`@version`	Версия библиотеки

Документирование функций с помощью JSDoc

Пример документирования функции с помощью JSDoc:

```
/**
*Сложение элементов массива
*@param {Array.<number>} numbersToAdd
*@return {Number} sum
*/
function addNumbers(numbersToAdd)
{
    var sum = 0;
    for (var i = 0; i<numbersToAdd.length; i++)
        sum += numbersToAdd[i];

    return sum;
}
```

Более подробно о системе JSDoc можно прочитать на сайте `http://usejsdoc.org`.

Классы и объекты

Язык JavaScript предоставляет возможность создания собственных объектов. Однако в JavaScript нет полноценной поддержки объектно-ориентированного программирования, такой как в языках C++ или Java.
Класс — это тип объекта, включающий в себя переменные и функции для управления этими переменными. Переменные называют *свойствами*, а функции — *методами*.
В действительности все, что находится внутри объекта, является свойством. Просто свойство, значением которого является функция, называется методом.
В JavaScript объекты можно создавать двумя способами:

- посредством объектного литерала;
- с помощью метода, называемого конструктором.

Определение объектов с помощью объектных литералов

Создание объекта с помощью объектного литерала начинается с определения обычной переменной посредством ключевого слова `var`, за которым следует оператор присваивания. В правой части этой инструкции записывается пара фигурных скобок, в которые заключаются пары «имя-значение», разделённые запятыми:

```
var car = {
    model: "ВАЗ-2109", // Сохранили строку
    year: 2007, // Сохранили число
    getModel: function() { // Сохранили ссылку на функцию
        return this.model;
    }
};
window.alert(car.model); // "ВАЗ-2109"
window.alert(car.getModel()); // "ВАЗ-2109"
```

Если в момент создания объекта ещё не известно, какие свойства он будет иметь, или впоследствии понадобится, чтобы он имел дополнительные свойства, то можно создать объект с каким-то начальным набором свойств или вообще без таковых, а нужные свойства добавить позднее.

```
var car = {};
car.model = "ВАЗ-2109";
car.year = 2007;
```

Определение объектов с помощью конструктора Object()

Второй способ создания объектов связан с использованием конструктора `Object()`. При этом сначала используется выражение `new Object()`, а затем определяются и инициализируются свойства полученного объекта:

```
var car = new Object ();
car.model = "ВАЗ-2109"; // Сохранили строку
car.year = 2007; // Сохранили число
car.getModel = function () { // Сохранили ссылку на функцию
    return this.model;
};
window.alert(car.model); // "ВАЗ-2109"
window.alert(car.getModel()); // "ВАЗ-2109"
```

Создание объектов с помощью конструктора `Object()` вполне приемлемо, однако считается, что такой подход менее предпочтителен по сравнению с созданием объектов с помощью объектных литералов. Основанием для этого служат следующие соображения:

- использование конструктора требует ввода большего количества символов по сравнению с объектными литералами;
- использование конструктора несколько снижает производительность кода в браузере;
- результирующий код читать труднее, чем в случае использования объектных литералов.

Получение и установка свойств объектов

После создания объекта и определения его свойств необходимо иметь возможность получать и устанавливать значения этих свойств. Возможны два способа доступа к свойствам объектов: с использованием точечной или скобочной нотации.

Точечная нотация

В соответствии с точечной нотацией после имени объекта ставится точка, а за ней имя свойства, значение которого нужно получить или установить. Например, чтобы создать новое свойство `color` объекта `car` или изменить значение уже существующего свойства, можно использовать инструкцию наподобие следующей:

```
car.color = "вишневый";
```

Если до этого свойство `color` отсутствовало в объекте, то данная инструкция создаст его. Если же это свойство уже существует, то оно будет обновлено новым значением.
Как правило, для установки и получения значений свойств используют точечную нотацию, поскольку она требует меньшего объёма ввода и легче читается.

Скобочная нотация

Чтобы установить с помощью скобочной нотации значение свойства, следует записать имя свойства, заключенное в кавычки, в квадратных скобках:

```
window.alert(car["model"]); // "ВАЗ-2109"
window.alert(car["year"]); // 2007
window.alert(car["getModel"]()); // "ВАЗ-2109"
```

Скобочная нотация предлагает возможности, которые точечная нотация не в состоянии обеспечить. Что наиболее важно, в квадратных скобках можно использовать переменные в тех случаях, когда при написании программы имя свойства не известно заранее:

```
var carProperty = "model";
car[carProperty] = "ВАЗ-2109";
```

Удаление свойств

Для удаления свойств из объектов используется оператор `delete`:

```
var myObject = {
    var1 : "the value",
    var2 : "another value",
    var3 : "yet another"
};
// удалить свойство var2 из объекта myObject
delete myObject.var2;

// попытка записи значения var2
document.write(myObject.var2); // ошибка
```

Использование ключевого слова this

Ключевое слово this — это краткая запись ссылки на объект, содержащий данный метод:

```
var Cars = {
    model:"",
    year:0,
    makeCar: function (m, y) {
        this.model = m;
        this.year = y;
}}
```

Указатель this полезен тогда, когда есть функция, которая может применяться множеством различных объектов. В таком случае ключевое слово this не связывается с каким-то одним объектом и будет указывать на тот объект, который вызывает данный метод.

Создание объектов с помощью конструкторов

С помощью функции конструктора можно создать новый тип объекта. Конструкторы объектов создаются так же, как и любая другая функция JavaScript, но для присвоения свойств новому объекту используется ключевое слово `this`. Новый объект наследует свойства объектного типа.

```
function Cars(m, y) { // Конструктор объекта
    this.model = m;
    this.year = y;
    this.getModel = function() {
        return this.model;
    }
}
```

Теперь для создания нового объекта достаточно назначить функцию новой переменной с помощью оператора `new`, например:

```
// Создание экземпляра
var car = new Cars("ВАЗ-2109", 2007);
```

Объект `car` наследует свойства объектного типа `Cars`. И хотя никаких действий для создания свойств объекта `car` не предпринималось, он содержит все свойства объекта `Cars`.

```
window.alert(car.model); // "ВАЗ-2109"
window.alert(car.getModel()); // "ВАЗ-2109"
```

Видоизменение объектного типа

Пусть, у нас есть объектный тип Cars, который служит прототипом для нескольких объектов. В один прекрасный момент мы осознаем, что объект Cars, а также все другие объекты, которые являются его потомками, нуждаются в нескольких дополнительных свойствах.
Для изменения объектного прототипа используют свойство prototype, которое наследуется каждым объектом от объекта Object:

```
function Cars(m, y) { // Конструктор объекта
    this.model = m;
    this.year = y;
    this.getModel = function() {
        return this.model;
    }
}
var car1 = new Cars("Москвич-412", 1978);
var car2 = new Cars("ВАЗ-2109", 2002);
// Объект Cars нуждается в дополнительных свойствах!
Cars.prototype.vehicleType = "passenger";
Cars.prototype.countAxles = 2;
// Проверка наличия новых свойств у существующих объектов
document.write (car1.countAxles); // 2
```

Прототипы

Предположим, необходимо составить массив, описывающий тысячу автомобилей. Свойства `model` и `year` в этом случае будут содержать разные значения, а вот метод `getModel()` во всех этих объектах один и тот же.
Использование прототипов позволяет определить метод вне конструктора. Таким образом, метод `getModel()` будет определен один раз, но будет наследоваться всеми экземплярами объекта.

```
function Cars(m, y) { // Конструктор объекта
    this.model = m;
    this.year = y;
}
Cars.prototype.getModel = function() {
    return this.model;
}
var car1 = new Cars("Москвич-412", 1978);
window.alert(car1.getModel()); // "Москвич-412"
var car2 = new Cars("ВАЗ-2109", 2002);
window.alert(car2.getModel()); // "ВАЗ-2109"
```

Прототипы

Любой созданный объект автоматически наследует свойства класса `Object`. Например, при попытке вывести значение экземпляра объекта в диалоговом окне вызывается метод `toString()`, который должен возвращать значение в виде строки. Для примера выведем текущее значение:

```
var car = new Cars("Москвич-412", 1978);
window.alert(car);
```

В результате в диалоговом окне получим следующий результат:

```
[object Object]
```

С помощью прототипов можно переопределить этот метод таким образом, чтобы выводилось нужное нам значение:

```
Cars.prototype.toString = function() {
    return "Модель: " + this.model + " Год выпуска: " + this.year;
}
var car = new Cars("Москвич-412", 1978);
window.alert(car);
```

В результате в диалоговом окне получим следующий результат:

```
Модель: Москвич-412 Год выпуска: 1978
```

Прототипы

Практически все встроенные объекты JavaScript (например, `String`, `Array`) имеют свойство `prototype`. С его помощью можно расширить возможности встроенных классов, например, добавить новый метод. В качестве примера добавим метод `inArray()` в класс Array. Этот метод будет производить поиск значения в массиве и возвращать индекс первого вхождения. Если вхождение не найдено,то метод вернёт значение `-1`:

```
Array.prototype.inArray = function(elem) {
    for (var i=0, len=this.length; i<len; i++) {
        if (this[i]===elem) return i;
    }
    return -1;
}
var arr = [ 1, 2, 3, 4, 5, 1 ];
var pos = arr.inArray(5);
if (pos != -1) window.alert("Индекс элемента " + pos);
else window.alert("Не найдено");
// Выведет: "Индекс элемента 4"
```

Однако расширять возможности встроенных классов не рекомендуется, т.к. другие программисты могут прийти в недоумение, увидев новый метод.

Создание объекта без экземпляра

Все рассмотренные варианты позволяли создавать свойства и методы экземпляра объекта. Тем не менее можно также создать свойства и методы, связанные с самим объектом, а не с его экземпляром:

```
function Cars() { }
Cars.model = "ВАЗ-2109";
Cars.year = 2007;
Cars.getModel = function() {
    return Cars.model;
};
```

Получить значения свойств и вызвать метод можно без создания экземпляра:

```
window.alert(Cars.model); // "ВАЗ-2109"
window.alert(Cars.year); // 2007
window.alert(Cars.getModel()); // "ВАЗ-2109"
```

Создание объектов с помощью метода Object.create()

Ещё один способ создания одних объектов на основе других заключается в использовании метода Object.create(). Преимуществом такого подхода является то, что он не требует написания функции-конструктора и лишь копирует свойства заданного объекта в другой объект. Если один объект наследуется от другого, то объект, от которого наследуются свойства, называется *прототипом*:

```
// создать обобщённый тип Cars
var Cars = {
    model: "ВАЗ-2109",
    year: 2007,
    getModel: function() {
        return this.model;
}
// создать объект car на основе Cars
var car = Object.create(Cars);
// проверка наличия унаследованных свойств
document.write (car.year); // выводит 2007
```

Встроенные классы JavaScript

Элементарные типы, такие как числа, строки и булевы значения, а также массивы и функции JavaScript также могут использоваться как объекты. Т.е. все они имеют свойства, и этим свойствам можно присваивать значения, как и в случае любого другого объекта.

Класс Number. Работа с числами

Класс Number используется для хранения числовых величин, а также для доступа к константам. Экземпляр класса создаётся по следующей схеме:

```
<Экземпляр класса> = new Number (<Начальное значение>);
```

Свойства класса Number можно использовать без создания экземпляра класса:

- MAX_VALUE — максимально допустимое в JavaScript число:

  ```
  var x = Number.MAX_VALUE; // 1.7976931348623157e+308
  ```

- MIN_VALUE — минимально допустимое в JavaScript число:

  ```
  var x = Number.MIN_VALUE; // 5e-324
  ```

- NaN — значение NaN:

  ```
  var x = Number.NaN; // NaN
  ```

- NEGATIVE_INFINITY — значение «минус бесконечность»:

  ```
  var x = Number.NEGATIVE_INFINITY; // -Infinity
  ```

- POSITIVE_INFINITY — значение «плюс бесконечность»:

  ```
  var x = Number.POSITIVE_INFINITY; // Infinity
  ```

Класс Array

Класс Array позволяет создавать массивы как объекты и предоставляет доступ к множеству методов для обработки массивов. Экземпляр класса можно создать следующими способами:

```
<Экземпляр класса> = new Array (<Количество элементов массива>);
<Экземпляр класса> = new Array (<Элементы массива через запятую>);
```

Если в круглых скобках нет никаких параметров, то создаётся массив пулевой длины, т. е. массив, не содержащий элементов. Если указано одно число, то это число задаёт количество элементов массива. Если указано несколько элементов через запятую или единственное значение не является числом, то указанные значения записываются в создаваемый массив.
Пример объявления массива:

```
var catNames = new Array("Larry", "Fuzzball", "Mr. Furly");
```

Свойства массивов

О массивах можно получать некоторую информацию, используя свойства массивов. Чаще всего используется свойство `length`. Это свойство позволяет узнать количество элементов в массиве, включая и те, которым значения ещё не присвоены.

```
var myArray = [];
myArray[2000] = 10;
myArray.length; // возвращает 2001
```

Методы для работы с массивами

push(<Список элементов>) — добавляет в массив элементы, указанные в списке элементов. Элементы добавляются в конец массива. Метод возвращает новую длину массива:

```
var Mass = [ "Один", "Два", "Три" ] ;
document.write(Mass.push ("Четвертый", "Пятый")); // 5
document.write(Mass.join (", ")); // "Один, Два, Три, Четвертый, Пятый"
```

unshift(<Список элементов>) — добавляет в массив элементы, указанные в списке элементов. Элементы добавляются в начало массива:

```
var Mass = [ "Один", "Два", "Три" ];
Mass.unshift("Четвертый", "Пятый");
document.write(Mass.join(", ")); // "Четвертый, Пятый, Один, Два, Три"
```

concat(<Список элементов>) — возвращает массив, полученный в результате объединения текущего массива и списка элементов. При этом в текущий массив элементы из списка не добавляются:

```
var Mass = [ "Один", "Два", "Три" ] ;
var Mass2 = []; // Пустой массив
Mass2 = Mass.concat("Четвертый", "Пятый");
document.write(Mass.join (", ")); // "Один, Два, Три"
document.write(Mass2.join (", "));
// "Один, Два, Три, Четвертый, Пятый"
```

Методы для работы с массивами

join(<Разделитель>) — возвращает строку, полученную в результате объединения всех элементов массива через разделитель:

```
var Mass = [ "Один", "Два", "Три" ];
var Str = Mass.join(" - ") ;
document.write(Str); // "Один - Два - Три"
```

shift() — удаляет первый элемент массива и возвращает его:

```
var Mass = [ "Один", "Два", "Три" ];
document.write(Mass.shift()); // "Один"
document.write(Mass.join(", ")); // "Два, Три"
```

pop() — удаляет последний элемент массива и возвращает его:

```
var Mass = [ "Один", "Два", "Три" ] ;
document.write (Mass.pop()); // "Три"
document.write (Mass.join(", ")); // "Один, Два"
```

reverse — переворачивает массив. Элементы будут следовать в обратном порядке относительно исходного массива:

```
var Mass = [ "Один", "Два", "Три" ];
Mass.reverse();
document.write (Mass.join(", ")); // "Три, Два, Один"
```

Методы для работы с массивами

`slice(<начало>, [<конец>])` — возвращает срез массива, начиная от индекса <начало> и заканчивая индексом <конец>, но не включает элемент с этим индексом. Если второй параметр не указан, то возвращаются все элементы до конца массива:

```
var Mass1 = [ 1, 2, 3, 4, 5 ];
var Mass2 = Mass1 .slice(1, 4);
window.alert (Mass2.join (", ")) ; // "2, 3, 4"
var Mass3 = Mass1.slice(2) ;
window.alert (Mass3.join (", ")) ; // "3, 4, 5"
```

`sort([Функция сортировки])` — выполняет сортировку массива. Если функция не указана, будет выполнена обычная сортировка (числа сортируются по возрастанию, а символы — по алфавиту):

```
var Mass = [ "Один", "Два'', "Три" ];
Mass.sort();
document.write(Mass.join(", ")); // "Два, Один, Три"
```

Методы для работы с массивами

Если нужно изменить стандартный порядок сортировки, это можно сделать с помощью функции сортировки. Функция принимает две переменные и должна возвращать:

- 1 — если первый больше второго;
- -1 — если второй больше первого;
- 0 — если элементы равны.

Например, изменим стандартную сортировку на свою сортировку без учёта регистра символов

```
function f_sort(Str1, Str2) { // Сортировка без учёта регистра
    var Str1_1 = Str1.toLowerCase(); // Преобразуем к нижнему регистру
    var Str2_1 = Str2.toLowerCase(); // Преобразуем к нижнему регистру
    if (Str1_1>Str2_1) return 1;
    if (Str1_l<Str2_1) return -1;
    return 0;
}
var Mass = [ "единица1", "Единый", "Единица2" ];
Mass.sort(f_sort); // Имя функции указывается без скобок
document.write(Mass.join(", ")); // "единица1, Единица2, Единый"
```

Для этого две переменные приводим к одному регистру, а затем производим стандартное сравнение. При этом регистр самих элементов массива не изменяется, т.к. создаются их копии.

Методы для работы с массивами

splice(<Начало>, <Количество>, [<Список значений>]) — позволяет удалить, заменить или вставить элементы массива. Возвращает массив, состоящий из удаленных элементов:

```
var Mass1 = [ 1, 2, 3, 4, 5 ];
var Mass2 = Mass1.splice(2, 2);
window.alert(Mass1.join(", ")); // "1, 2, 5"
window.alert(Mass2.join(", ")); // "3, 4"
var Mass3 = Mass1.splice (1, 1, 7, 8, 9);
window.alert(Mass1.join(", ")); // "1, 7, 8, 9, 5"
window.alert(Mass3.join(", ")); // "2"
var Mass4 = Mass1.splice(1, 0, 2, 3, 4);
window.alert(Mass1.join(", ")); // "1, 2, 3, 4, 7, 8, 9, 5"
window.alert(Mass4.join(", ")); // Пустой массив
```

toString() и valueOf() — преобразуют массив в строку. Элементы указываются через запятую без пробела:

```
var Mass = [ "Один", "Два", "Три" ];
document.write(Mass.toString()); // "Один, Два, Три"
```

Методы для работы с массивами

Метод	Возвращаемое значение
`every()`	True, если каждый элемент массива удовлетворяет условию
`filter()`	Массив, который содержит все элементы текущего массива удовлетворяющие условию
`forEach()`	Выполняет заданную функцию для каждого элемента массива
`indexOf()`	Индекс первого вхождения заданного значения или -1
`lastIndexOf()`	Индекс последнего вхождения заданного значения или -1
`map()`	Массив, полученный путём преобразования каждого элемента
`reduce()`	Сводит два значения массива в одно, применяя к обоим заданную функцию (слева направо)
`reduceRight()`	Сводит два значения массива в одно, применяя к обоим заданную функцию (справа налево)
`some()`	True, если один или несколько элементов массива удовлетворяет условию

Многомерные массивы

В любом элементе массива могут храниться данные любого типа, в том числе и другие массивы,например:

```
Mass1[0] = [1, 2, 3, 4];
```

В этом случае получить значение массива можно, указав два индекса:

```
Str = Mass1[0][2]; // Переменной Str будет присвоено значение 3
```

Так как массив является объектом, то операция присваивания сохраняет в переменной ссылку на массив, а не все его значения. Например, если попробовать сделать так:

```
var Mass1, Mass2;
Mass1 = [1, 2, 3, 4];
Mass2 = Mass1; // Присваивается ссылка на массив
Mass2[0] = "Новое значение";
document.write(Mass1.join(", ") + "<br>");
document.write(Mass2.join(", "));
```

то изменение Mass2 затронет Mass1, и мы получим следующий результат:

```
Новое значение, 2, 3, 4
Новое значение, 2, 3, 4
```

Многомерные массивы

Многомерные массивы также можно создать с помощью объекта Array:

```
var Mass = new Array(new Array("Один", "Два", "Три"),
                     new Array("Четыре", "Пять", "Шесть"));
document.write(Mass[0][1]); // "Два"
```

или поэлементно:

```
var Mass = new Array();
Mass[0] = new Array();
Mass[1] = new Array();
Mass[0][0] = "Один";
Mass[0][1] = "Два";
Mass[0][2] = "Три";
Mass[1][0] = "Четыре";
Mass[1][1] = "Пять";
Mass[1][2] = "Шесть";
document.write(Mass[1][2]); // "Шесть"
```

Многомерные массивы

При использовании многомерных массивов метод `slice()` создаёт «поверхностную» копию, а не полную:

```
var Mass1, Mass2;
Mass1 = [[0, 1], 2, 3, 4];
Mass2 = Mass1.slice(0);
Mass2[0][0] = "Новое значение1";
Mass2[1] = "Новое значение2";
```

В результате массивы будут выглядеть так:

```
Mass1 = [["Новое значение1", 1], 2, 3, 4];
Mass2 = [["Новое значение1", 1], "Новое значение2", 3, 4];
```

Как видно из примера, изменение вложенного массива в `Mass2` привело к одновременному изменению значения в `Mass1`. Иными словами, оба массива содержат ссылку на один и тот же вложенный массив.

Класс Math

Класс Math содержит математические константы и функции. Его использование не требует создания экземпляра класса.
Свойства:

- E — e, основание натурального логарифма;
- LN2 — натуральный логарифм 2;
- LN10 — натуральный логарифм 10;
- LOG2E — логарифм по основанию 2 от e;
- LOG10E — десятичный логарифм от e;
- PI — число Пи:

```
document.write(Math.PI); // 3.141592653589793
```

- SQRTI_2 — квадратный корень из 0.5;
- SQRT2 — квадратный корень из 2.

Класс Math

Методы:

- `abs()` — абсолютное значение;
- `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Значение указывается в радианах;
- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах;
- `exp()` — экспонента;
- `log()` — натуральный логарифм;
- `pow(<Число>, <Степень>)` — возведение <Числа> в <Степень>:

```
var x = 5;
document.write(Math.pow(x, 2)); // 25 (5 в квадрате)
```

- `sqrt()` — квадратный корень:

```
var x = 25;
document.write(Math.sqrt(x)); // 5 (квадратный корень из 25)
```

- `max(<Список чисел через запятую>)` — максимальное значение из списка:

```
document.write(Math.max(3, 10, 6)); // 10
```

- `min(<Список чисел через запятую>)` — минимальное значение из списка:

```
document.write(Math.min(3, 10, 6)); // 3
```

Класс Math

- `round()` — значение, округленное до ближайшего целого. Если первое число после запятой от 0 до 4, то округление производится к меньшему по модулю целому, а в противном случае — к большему:

```
var x = 2.499;
var y = 2.5;
document.write(Math.round(x)); // округлено до 2
document.write(Math.round(y)); // округлено до 3
```

- `ceil()` — значение, округленное до ближайшего большего целого:

```
var x = 2.499;
var y = 2.5;
document.write(Math.ceil(x)); // округлено до 3
document.write(Math.ceil(y)); // округлено до 3
```

- `floor()` — значение, округленное до ближайшего меньшего целого:

```
var x = 2.499;
var y = 2.5;
document.write(Math.floor(x)); // округлено до 2
document.write(Math.floor(y)); // округлено до 2
```

- `random()` — случайное число от 0 до 1:

```
document.write(Math.random()); // например, 0.9778613566886634
```

Класс Date

Даты хранятся как количество миллисекунд, прошедших с полуночи 1 января 1970 г. согласно универсальному временному коду (Universal Time Code, UTC). Благодаря такому формату с помощью объектов Date можно точно представлять даты, отстоящие от 1 января 1970 г. на 285 616 лет. В ECMAScript 5 добавлен метод Date.now(), который возвращает дату и время его выполнения в миллисекундах. Это позволяет использовать объекты Date для профилирования кода:

```
// получение времени начала
var start = Date.now();

//вызов функции
doSomething();

//получение времени окончания
var stop = Date.now(),
    result = stop - start;
```

Метод Date.now() реализован в Internet Explorer 9+, Firefox 3+, Safari 3+, Opera 10.5 и Chrome.

Примитивные и ссылочные значения

JavaScript переменные могут содержать значения двух видов: примитивные и ссылочные. *Примитивные значения* (primitive values) — это просто атомарные элементы данных, в то время как *ссылочные значения* (reference values) — это объекты, которые могут состоять из нескольких значений. Примитивные значения относятся к одному из пяти примитивных типов данных: неопределённому (`undefined`), нулевому (`null`), логическому (`boolean`), числовому (`number`) и строковому (`string`). Доступ к переменным этих типов осуществляется по значению (by value), то есть вы работаете с фактическим значением, хранящимся в переменной. Примитивные значения имеют фиксированный размер и хранятся в памяти в стеке.

Ссылочные значения — это объекты, хранящиеся в памяти в куче. При выполнении каких-либо действий над объектом вы на самом деле работаете не с самим объектом, а со *ссылкой* (reference) на него. Говорят, что доступ к таким значениям осуществляется *по ссылке*.

Копирование значений

Когда одна переменная с примитивным значением присваивается другой, создаётся копия значения, хранящегося в объекте переменных, а затем она записывается по адресу новой переменной, например:

```
var num1 = 5;
var num2 = num1;
```

Здесь num1 содержит значение 5. Когда переменная num2 инициализируется значением num1, она также получает значение 5. Она никак не связана с num1, потому что содержит копию значения. Затем эти переменные можно использовать по отдельности.
Когда ссылочное значение одной переменной присваивается другой, то после копирования обе переменные указывают на один объект, поэтому изменения одной из них отражаются на другой:

```
var obj1 = new Object();
var obj2 = obj1;
obj1.name = "Nicholas";
alert(obj2.name);   // "Nicholas"
```

В этом примере переменной obj1 присваивается новый объект, после чего значение переменной копируется в obj2. Теперь обе переменные указывают на один объект, и когда для obj1 задаётся свойство name, оно становится доступным через obj2.

Передача аргументов

Все аргументы функций в JavaScript передаются по значению. Это означает, что значения извне функции копируются в аргументы внутри функции так же, как копируются значения переменных: примитивные — по одному сценарию, ссылочные — по другому.
Когда аргумент передаётся по значению, его значение просто копируется в локальную переменную.

```
function addTen(num) {
    num +=10;
    return num;
}
var count = 20;
var result = addTen(count);
alert(count); // 20 - без изменений
alert(result); // 30
```

При вызове функции `addTen()` ей передаётся переменная `count` со значением 20, которое копируется в аргумент `num` для использования в функции `addTen()`. В функции к значению `num` прибавляется 10, но это не изменяет значение `count` вне функции. Аргумент `num` и переменная `count` не знают друг о друге, они просто имеют одинаковые значения. Если бы значение `num` было передано по ссылке, переменная `count` также стала бы равной 30 согласно изменению внутри функции.

Передача аргументов

При передаче аргумента по ссылке в локальной переменной сохраняется расположение его значения в памяти. Это означает, что изменения локальной переменной отражаются вне функции.

```
function setName(obj) {
    obj.name = "Nicholas";
}
var person = new Object();
setName(person);
alert(person.name); // "Nicholas"
```

Внутри функции переменные obj и person указывают на один и тот же объект. Поэтому при задании свойства name в функции это изменение отражается вне функции. Тем не менее, объекты все равно передаются по значению:

```
function setName(obj) {
    obj.name = "Nicholas";
    obj = new Object();
    obj.name = "Greg";
}
var person = new Object();
setName(person);
alert(person.name); // "Nicholas"
```

Объектная модель браузера

Объектная модель браузера — это совокупность объектов, обеспечивающих доступ к содержимому Web-страницы и ряду функций Web-браузера.
Объектная модель представлена в виде иерархии объектов. То есть имеется объект верхнего уровня и подчинённые ему объекты. В свою очередь подчинённые объекты имеют свои подчинённые объекты.
Часто объект верхнего уровня (и даже подчинённый объект) можно не указывать. Рассмотрим выражение для вызова диалогового окна с сообщением:

```
window.alert("Сообщение");
```

Здесь `window` — это объект самого верхнего уровня, представляющий сам Web-браузер, а `alert()` — это метод объекта `window`. В этом случае указывать объект не обязательно, т. к. объект `window` подразумевается по умолчанию:

```
alert("Сообщение");
```

При печати сообщения в окне Web-браузера часто используется команда:

```
document.write("Сообщение");
```

Поскольку объект `document` является подчинённым объекту `window`, то правильно было бы написать так:

```
window.document.write("Сообщение");
```

Объектная модель браузера

Помимо уже упомянутого объекта самого высокого уровня — `window` в объектной модели имеются следующие основные объекты:

- `event` — предоставляет информацию, связанную с событиями.
- `frame` — служит для работы с фреймами (коллекция `frames`);
- `history` — предоставляет доступ к списку истории Web-браузера;
- `navigator` — содержит информацию о Web-браузере;
- `location` — содержит URL-адрес текущей Web-страницы;
- `screen` — служит для доступа к характеристикам экрана компьютера пользователя;
- `document` — служит для доступа к структуре, содержанию и стилю документа.

Объект window

Основная область браузера называется *окном* (window). Это та область, в которую загружаются HTML-документы (и связанные с ними ресурсы). Каждая вкладка браузера представляется в JavaScript экземпляром объекта `window`. Объект `window` — это объект самого верхнего уровня. Объект `window` подразумевается по умолчанию, поэтому указывать ссылку на объект не обязательно. Методы объекта `window` перечислены в табл.:

Метод	Использование
`alert()`	Отображает окно оповещения, которое содержит текст сообщения и кнопку OK
`clearInterval()`	Отменяет повторное выполнение кода, заданного методом `setInterval()`
`close()`	Закрывает текущее окно
`confirm()`	Отображает окно подтверждения, которое содержит сообщение и две кнопки — OK и Cancel (Отмена)
`createPopup()`	Создает всплывающее окно
`focus()`	Делает текущее окно активным
`showModalDialog()`	Открывает модальное диалоговое окно
`promt()`	Отображает окно запроса, ожидающее пользовательского
`setInterval()`	Периодически вызывает функцию через определенные промежутки времени (заданные в миллисекундах))

Таймеры

Таймеры позволяют однократно или многократно выполнять указанную функцию через определённый интервал времени. Для управления таймерами используются следующие методы объекта `window`:

- `setTimeout()` — создаёт таймер, однократно выполняющий указанную функцию или выражение спустя заданный интервал времени:

  `<Идентификатор> = setTimeout(<Функция или выражение>, <Интервал>);`

- `clearTimeout(<Идентификатор>)` — останавливает таймер, установленный методом `setTimeout()`.
- `setInterval()` — создаёт таймер, многократно выполняющий указанную функцию или выражение через заданный интервал времени.

  `<Идентификатор> = setInterval(<Функция или выражение>, <Интервал>);`

- `clearInterval(<Идентификатор>)` — останавливает таймер, установленный методом `setInterval()`.

Здесь `<Интервал>` — это промежуток времени, по истечении которого выполняется `<Функция или выражение>`. Значение указывается в миллисекундах.

Использование свойств и методов объекта document

DOM (Document Object Model — объектная модель документа) — это интерфейс, посредством которого JavaScript работает с HTML-документами в окнах браузера.
Объект document предоставляет доступ ко все элементам Web-страницы.
Некоторые методы объекта document:

Метод	Использование
addEventListener()	Назначает обработчик событий в документе
createAttribute()	Создаёт узел атрибута
createComment()	Создает узел комментария
createElement()	Создает узел элемента
createDocumentFragment()	Создает пустой фрагмент документа
getElementById()	Получает элемент по заданному атрибуту id
getElementsByName()	Получает все элементы с указанным именем
getElementsByTagName()	Получает все элементы с указанным именем тега
removeEventListener()	Удаляет из документа обработчик событий, добавленный ранее addEventListener()
write()	Записывает текст, переданный как параметр, в текущее место документа

Метод getElementById()

Метод getElementById(<Идентификатор>), до сих пор чаще всего используемый для выбора элементов, занимает важное место в современной веб-разработке. Это весьма удобное средство, позволяющее находить и обрабатывать любой элемент по его идентификатору — уникальному атрибуту id.
Используя этот метод, можно работать с любым элементом в документе, где бы он ни находился, если известен его идентификатор.
Метод возвращает ссылку только на один элемент (даже если элементов с одинаковым идентификатором несколько), т.к. по определению идентификатор должен быть уникальным.
Пример:

```
var Str = document.getElementByld("checkl").value;
```

Использование свойств и методов объекта `Element`

Объект `Element` предоставляет свойства и методы для работы с HTML-элементами в документе.

Некоторые свойства объекта `Element`:

Свойство	Использование
`innerHtml`	Получает или устанавливает содержимое элемента
`height`	Высота элемента
`width`	Ширина элемента
`clientHeight`	Высота ширина элемента без учёта рамок, границ, полос прокрутки и т.п.
`clientWidth`	Ширина элемента без учёта рамок, границ, полос прокрутки и т.п.
`clientLeft`	Смещение левого края элемента относительно левого края элемента-родителя без учёта рамок, границ, полос прокрутки и т.п.
`clientTop`	Смещение верхнего края элемента относительно верхнего края элемента-родителя без учёта рамок, границ, полос прокрутки и т.п.
`offsetHeight`	Высота элемента относительно элемента-родителя
`offsetWidth`	Ширина элемента относительно элемента-родителя;
`offsetLeft`	Смещение левого края элемента относительно левого края элемента-родителя

Свойство innerHTML

Самым важным свойством элемента, которое можно изменить посредством DOM, является свойство innerHTML:

```
...
<p id="nearFar"> The near and far values are displayed here. </p>
...
// Получить ссылку на элемент с атрибутом id = nearFar
var nf = document.getElementById('nearFar');
...
// Вывести текущие значения near и far
nf.innerHTML = 'near: ' + g_near + ', far: ' + g_far;
```

В текст сообщения, записываемого с помощью свойства innerHTML можно также вставлять теги HTML. Например, если свойству innerHTML присвоить строку "Good Morning, Marisuke-san!"(«Доброе утро, Марисуки-сан!»), слово «Marisuke» будет выделено жирным.

Методы объекта `Element`:

Метод	Использование
`addEventListener()`	Назначает обработчик событий для элемента
`removeEventListener()`	Удаляет указанный слушатель событий
`click()`	Имитирует щелчок мышью на элементе
`blur()`	Лишает элемент фокуса ввода
`focus()`	Переводит фокус ввода на данный элемент
`getAttribute()`	Получает значение указанного атрибута элемента
`setAttribute()`	Устанавливает указанное значение для атрибута
`removeAttribute()`	Удаляет заданный атрибут из элемента
`hasAttribute()`	Возвращает значение `true`, если элемент содержит указанный атрибут, иначе — `false`
`appendChild()`	Вставляет новый дочерний узел в элемент (в качестве последнего дочернего узла)
`insertBefore()`	Вставляет новый дочерний узел перед заданным существующим узлом
`replaceChild()`	Заменяет указанный дочерний узел другим
`hasChildNodes()`	Возвращает значение `true`, если элемент содержит дочерние узлы, иначе — `false`
`removeChild()`	Удаляет заданный дочерний узел
`toString()`	Преобразует элемент в строку

Контекст выполнения и область видимости

Концепция контекста выполнения, или просто *контекста* (context), очень важна в JavaScript. От контекста выполнения переменной или функции зависит, какие другие данные ей доступны и как она должна работать. С каждым контекстом выполнения связан *объект переменных* (variable object), содержащий все переменные и функции контекста. Он недоступен в коде, но используется за кулисами для обработки данных.
Наиболее общим является глобальный контекст выполнения. В веб-браузерах глобальным контекстом считается контекст объекта `window`, так что все глобальные переменные и функции создаются как свойства и методы объекта `window`. Когда весь код в контексте выполнен, он уничтожается вместе со всеми определёнными в нём переменными и функциями (глобальный контекст существует вплоть до закрытия веб-страницы или веб-браузера).
У каждого вызова функции имеется свой локальный контекст выполнения. При передаче управления в функцию её контекст помещается в стек контекста, а при выходе из функции он извлекается из стека, при этом управление возвращается в прежний контекст. Так осуществляется контроль над последовательностью операций в JavaScript-программе.

Контекст выполнения и область видимости

При выполнении кода в контексте создаётся *цепочка областей видимости* (scope chain) объектов переменных, которая обеспечивает упорядоченный доступ ко всем переменным и функциям, доступным в контексте выполнения. Первым звеном цепочки областей видимости всегда является объект переменных контекста, код которого выполняется. Каждый последующий объект переменных в цепочке относится к все более внешнему контексту, пока не достигается глобальный контекст. Объект переменных глобального контекста всегда последний в цепочке областей видимости.

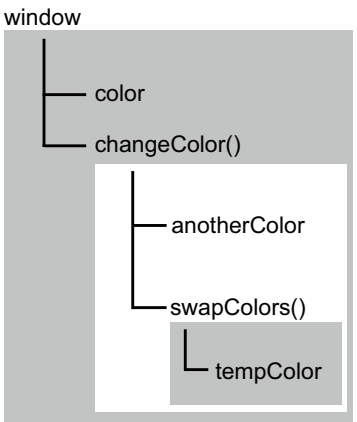
При разрешении идентификатора его имя ищется в цепочке областей видимости. Поиск всегда осуществляется в направлении от первого звена к последнему, пока не обнаруживается идентификатор (если найти его не удаётся, обычно возникает ошибка).

Если переменная инициализируется без использования ключевого слова `var`, то в этот момент она создаётся в глобальном контексте.

Аргументы функций считаются переменными и подчиняются тем же правилам доступа, что и любые другие переменные в контексте выполнения.

Контекст выполнения и область видимости

```
var color = "blue";
function changeColor(){
    var anotherColor = "red";
    function swapColors(){
        var tempColor = anotherColor;
        anotherColor = color;
        color = tempColor;
        // здесь доступны переменные color, anotherColor и tempColor
    }
    // здесь доступны переменные color и anotherColor, но не tempColor
    swapColors();
}
// здесь доступна только переменная color
changeColor();
```



Отсутствие блочных областей видимости

Блочных областей видимости в JavaScript нет. Например, следующий JavaScript-код работает не так, как его аналоги во многих других языках:

```
if (true) {
    var color = "blue";
}
alert(color); // "blue"
```

Здесь в инструкции `if` определяется переменная `color`. В языках вроде C, C++ и Java она была бы уничтожена после выполнения `if`, однако в JavaScript объявляемая переменная добавляется в текущий контекст выполнения (глобальный в данном случае). Об этом важно помнить при создании циклов `for`, которые обычно выглядят так:

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


alert(i); // 10
```

В JavaScript переменная `i`, созданная в инструкции `for`, продолжает существовать вне цикла после его выполнения.

Управление памятью

JavaScript — это язык со сборкой мусора, то есть за управление памятью при работе сценариев отвечает среда выполнения. В языках вроде C и C++ слежение за использованием памяти относится к важнейшим задачам и является источником многих проблем. JavaScript освобождает разработчиков от забот по управлению памятью, автоматически выделяя сценариям нужную память и возвращая в среду память, которая больше не используется. Сборщик мусора запускается периодически с заданной частотой или в предопределенные моменты выполнения кода.
Как правило, объём памяти, доступной в веб-браузерах, гораздо меньше, чем в обычных приложениях. Он ограничивается в основном ради безопасности, чтобы сценарии JavaScript в веб-страницах не могли вызвать сбой операционной системы, израсходовав всю системную память. Ограничения памяти влияют не только на выделение памяти для переменных, но и на стек вызовов и количество инструкций, выполняемых в одном потоке.

Управление памятью

Если данные больше не нужны, то для оптимизации сборки мусора, лучше всего присвоить соответствующей переменной значение `null`. Этот совет относится преимущественно к глобальным переменным и свойствам глобальных объектов. В случае локальных переменных ссылки на данные удаляются автоматически, когда переменные покидают контекст:

```
function createPerson(name){
    var localPerson = new ObjectQ;
    localPerson.name = name;
    return localPerson;
}
var globalPerson = createPerson("Nicholas");
// какие-то действия с globalPerson
globalPerson = null;
```

В этом коде переменная `localPerson` покидает контекст при завершении функции `createPerson()`, так что разрывать её связь со значением не требуется. Что касается `globalPerson`, то это глобальная переменная, поэтому когда она больше не требуется, ей присваивается значение `null`. Разрыв связи переменной со значением не приводит к автоматическому освобождению занятой им памяти. Значение просто выводится из контекста, что делает возможным возвращение памяти при следующей сборке мусора.

Использование событий в JavaScript

События — это все то, что происходит в браузере (например, загрузка веб-страницы), а также то, что делает пользователь (например, щелчки мышью, нажатия клавиш, перемещения мыши, изменение размеров окна, и т.п.). В браузере постоянно происходят какие-то события.
HTML DOM позволяет JavaScript распознавать события, происходящие в браузере, и реагировать на них. События можно разделить на отдельные группы в соответствии с тем, с какими HTML-элементами или объектами браузера они связаны.

События, поддерживаемые всеми HTML-элементами

Событие	Условие возникновения
`click`	Выполнения щелчка мышью на элементе
`dbclick`	Выполнение двойного щелчка мышью на элементе
`keydown`	Нажатие клавиши. Наступает постоянно до отпускания
`keypress`	Аналогично событию `keydown`, но возвращает значение кода символа в кодировке Unicode.
`keyup`	Отпускание клавиши после того, как она была нажата
`mousedown`	Нажатие кнопки мыши над элементом
`mouseenter`	Перемещение указателя мыши в область элемента, к которому подключен слушатель событий
`mouseleave`	Перемещение указателя мыши за пределы элемента, к которому подключен слушатель событий
`mousemove`	Перемещение указателя мыши над элементом
`mouseout`	Перемещение указателя мыши за пределы элемента или одного из его дочерних элементов, к которому подключен слушатель событий
`mouseover`	Перемещение указателя мыши над элементом или одним из его дочерних элементов, к которому подключен слушатель событий
`mouseup`	Отпускание кнопки мыши над элементом
`mousewheel`	Вращение колесика мыши

События,поддерживаемые всеми элементами кроме <body>,<frameset>

Событие	Условие возникновения
blur	Потеря элементом фокуса ввода
error	Ошибка при загрузке файла
focus	Приобретение элементом фокуса ввода
load	После загрузки Web-страницы
resize	Изменение размера окна
scroll	Прокрутка документа или элемента
afterprint	Начало вывода документа на печать
beforeprint	Открытие окна предварительного просмотра выводимого на печать документа или завершение подготовки документа к печати
beforeunload	Окно, документ и подключенные к нему файлы подготовлены к выгрузке
pagehide	Браузер покидает страницу в истории просмотра
pageshow	Браузер переходит на страницу в истории сеанса
popstate	Изменение элемента истории просмотра в активном сеансе
unload	Выгрузка документа или включенного в него файла. Наступает после события beforeunload;

Последовательность событий

События возникают последовательно, например, последовательность событий при нажатии кнопкой мыши на элементе страницы будет такой:

```
mousedown
mouseup
click
```

При двойном нажатии последовательность будет такой:

```
mousedown
mouseup
click
dblclick
```

Это значит, что событие `dblclick` возникает после события `click`.
При нажатии клавиши на клавиатуре последовательность будет такой:

```
keydown
keypress
keyup
```

Последовательность возникновения событий также может зависеть от используемого браузера.

Обработка событий

Обработчик события — это асинхронная функция обратного вызова, обрабатывающая события ввода от пользователя, такие как щелчки мышью или нажатия клавиш на клавиатуре. Возможность определения обработчиков позволяет создавать динамические веб-страницы и изменять их содержимое в соответствии с вводом пользователя.
Первая из систем обработки событий была введена одновременно с выпуском первых версий JavaScript. Она основана на использовании специальных атрибутов обработчиков событий.
Встроенные атрибуты обработчиков событий образуются путём добавления префикса `on` к имени события. Их использование сводится к добавлению атрибута события в HTML-элемент. Когда происходит указанное событие, выполняется JavaScript-код, записанный в виде значения атрибута:

```
<a href="home.html" onclick="alert('Перейти на главную страницу!')";
Щёлкните здесь для перехода на главную страницу</a>
```

При щёлкнете на ссылке на экране отобразится окно сообщения с текстом «Перейти на главную страницу!». Когда вы закроете это окно, выполнится заданный по умолчанию обработчик событий для этого элемента, который осуществит переход по ссылке, указанной в атрибуте `href`.

Обработка событий с использованием свойств элементов

Одним из наибольших недостатков, связанных с использованием устаревшей техники встраивания обработчиков событий в элементы, является то, что при этом нарушается главный принцип наилучшей практики программирования, а именно отделение кода представления (ответственного за внешний вид объектов) от функционального кода (ответственного за выполнение полезных действий). Смешивание обработчиков событий с тегами HTML затрудняет сопровождение веб-страниц и их отладку, одновременно затрудняя понимание кода.
В версии 3 браузера Netscape была введена новая модель событий, позволяющая программистам подключать события к элементам в виде свойств:

```
document.getElementById("incrementButton").onclick = increaseCount;

function increaseCount(){
    ...
}
```

Обратите внимание, что за именами функций, назначаемых обработчику событий, не следуют круглые скобки. Такой синтаксис не означает вызова функции и лишь указывает на то, что эта функция должна выполняться всякий раз, когда происходит данное событие.

Обработка событий с использованием метода addEventListener()

Назначить обработчик события в модели DOM Level 2 позволяет метод `addEventListener()`. Метод `addEventListener()` прослушивает события на любом DOM-узле и запускает выполнение заданных действий в зависимости от наступившего события. Когда запускается функция, указанная для обработчика события в качестве выполняемого действия, она автоматически получает единственный аргумент — объект `Event`. В соответствии с общепринятым соглашением для этого аргумента используется имя e.
По сравнению с использованием атрибутов DOM-событий метод `addEventListener()` обладает следующими преимуществами:

- одному элементу может быть назначено несколько обработчиков событий;
- работает на любом узле DOM, а не только на элементах;
- предоставляет большую степень контроля над обработкой события.

Удалить обработчик события можно с помощью метода `removeEventListener()`.

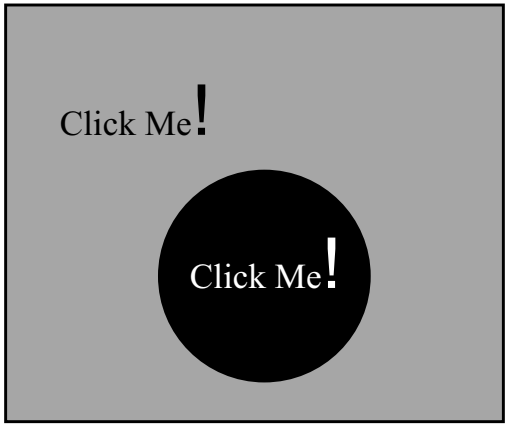
Обработка событий с использованием метода addEventListener()

Пример использования метода addEventListener():

```
document.getElementByld("incrementButton").
                    addEventListener('click',increaseCount,false);
function increaseCount(e){ // e - ссылка на объект event
    window.alert("Элемент " + this.getAttribute("id"));
}
```

Метод addEventListener() реализован с использованием трёх аргументов.
Первый аргумент — это тип события. В отличие от двух других способов обработки событий, метод addEventListener() требует указания лишь имени события без префикса on, например, click вместо onclick.
Второй аргумент — это функция, которая должна вызываться при наступлении события. В эту функцию в качестве аргумента передаётся ссылка на объект event, а внутри функции через ключевое слово this доступна ссылка на текущий элемент.
Третий аргумент — это булево значение (true или false), которое определяет очередность выполнения обработчиков событий в тех случаях, когда у элемента, с которым связано событие, имеется родительский элемент, также связанный с событием.

Обработка событий с использованием метода addEventListener()



В случае вложенных элементов важно знать, какое из двух событий произойдет первым. Например, если щёлкнуть на внутреннем круге, то какое событие должно произойти первым: связанное с квадратом или с кругом?

Наиболее распространённой стратегией обработки подобных событий является модель *всплытия событий*. События, относящиеся к наиболее глубоко вложенным элементам, происходят первыми, а затем «всплывают» в направлении наружных элементов. Чтобы использовать эту стратегию, следует установить для последнего аргумента метода `addEventListener()` значение `false`, которое является также значением по умолчанию и используется в большинстве случаев.

Другой способ обработки подобных сценариев называется *захватом событий*. В этом случае первыми происходят внешние события, а последними — внутренние.

Объект event

Объект event позволяет получить детальную информацию о произошедшем событии и выполнить необходимые действия. Объект event доступен только в обработчиках событий. При наступлении следующего события все предыдущие значения свойств сбрасываются. Объект event имеет следующие свойства:

- clientX и clientY — координаты события (по осям X и Y) в клиентских координатах;
- screenX и screenY — координаты события (по осям X и Y) относительно окна;
- offsetX и offsetY — координаты события (по осям X и Y) относительно контейнера;
- x и y — координаты события по осям X и Y. В модели DOM Level 2 этих свойств нет;
- button — число, указывающее нажатую кнопку мыши. Может принимать следующие значения:
  - 0 — кнопки не были нажаты;
  - 1 — нажата левая кнопка мыши;
  - 2 — нажата правая кнопка мыши;
  - 3 — левая и правая кнопки мыши были нажаты одновременно;
  - 4 — нажата средняя кнопка.

  В модели DOM Level 2 значения другие:
  - 0 — нажата левая кнопка мыши;
  - 1 — нажата средняя кнопка;
  - 2 — нажата правая кнопка мыши;

Свойства объекта event

- `keyCode` — код нажатой клавиши клавиатуры.
- `altKey` — `true`, если в момент события была нажата клавиша <Alt>;
- `ctrlKey` — `true`,если была нажата клавиша <Ctrl>;
- `shiftKey` — `true`,если была нажата клавиша <Shift>;
- `target` — содержит объект, породивший событие.

Действия по умолчанию и их отмена

Во многих случаях может оказаться желательным, чтобы действие по умолчанию, связанное с данным элементом, не было выполнено. Например, чтобы закрытие окна сообщения не сопровождалось никакими другими действиями.
Для этого в JavaScript разработано несколько приемов. Один из них состоит в возврате значения `false`. Другой вариант — сделать действие по умолчанию нерезультативным. Например, замена значения атрибута `href` на # приведет к тому, что ссылка будет указывать на саму себя:

```
<a href="#" onclick=
"alert('Перейти на главную страницу!'); return false;">
Щёлкните здесь</a>
```

Кроме того для отмены действий по умолчанию в модели DOM Level 2 можно воспользоваться методом `preventDefault()` объекта `event` или присвоить свойству `returnValue` значение `false`:

```
<a href="file.html" onclick="f_event(event);">
function f_event(e) {
    e = e||window.event;
    if (e.preventDefault) e.preventDefault();
    else e.returnValue = false;
    window.alert("Перехода по ссылке не будет!");
}
```

Прерывание всплывания событий

Помимо стратегий всплытия и захвата событий, вложенные события можно обрабатывать третьим способом: обработать единственное событие и на этом остановиться. Для этого свойству `cancelBubble` объекта `event` следует присвоить значение `true`. Кроме того, в модели DOM Level 2 для прерывания всплывания событий можно воспользоваться методом `stopPropagation()` объекта `event`:

```
<body onclick="f_print(event);">

function f_print(e) {
    e = e||window.event;
    if (e.stopPropagation) e.stopPropagation();
    else e.cancelBubbie = true;
}
```

Объект AJAX

AJAX (Asynchronous JavaScript and XML, асинхронный JavaScript и XML) — это технология программной загрузки произвольных данных. Программа инициирует загрузку файла с данными, а потом считывает его содержимое и обрабатывает — и все это без перезагрузки самой страницы. С помощью этой технологии можно загружать обычные текстовые файлы. Технология AJAX позволяет загружать данные исключительно с Web-сервера. Загрузка из локальных файлов во всех Web-обозревателях заблокирована в целях безопасности.

Объект AJAX. Подготовка к загрузке данных

Прежде чем загрузить какой-либо файл с применением технологии AJAX, следует получить объект, который, собственно, и выполнит его загрузку. В Firefox, Chrome, Opera, Safari и Internet Explorer, начиная с версии 7, загрузкой данных «заведует» класс `XMLHttpRequest`. Этот класс поддерживается самим Web-обозревателем и определен в стандарте DOM, поэтому способ загрузки данных с его помощью носит название стандартного.
Нам нужно лишь создать объект упомянутого класса оператором `new`. Никакие параметры при этом не указываются:

```
var oAJAX = new XMLHttpRequest();
```

Класс `XMLHttpRequest` также доступен через одноимённое свойство объекта `window`.

Объект AJAX. Отправка запроса

Получив объект, реализующий технологию AJAX, мы можем отправить Web-серверу запрос на загрузку файла с данными.
Существуют две разновидности запросов на получение данных AJAX, которые мы можем отправить:

- *синхронный запрос*, при котором Web-обозреватель приостанавливает выполнение программы и ждёт, пока файл с данными не будет получен;
- *асинхронный запрос*, при котором Web-обозреватель продолжает выполнять программу, не дожидаясь получения запрошенного файла, а когда он, наконец, будет получен, генерирует особое событие.

На практике чаще встречаются асинхронные запросы, т.к. они позволяют получить данные по ходу выполнения прочего JavaScript-кода и, соответственно, не приводят к «зависанию» всей страницы.

Объект AJAX. Задание параметров запроса

Сначала нам следует указать параметры отправляемого запроса: метод отсылки данных (GET или POST), интернет-адрес файла с данными или программы, которая сгенерирует эти данные, и вид запроса (синхронный или асинхронный). Все это выполняет метод open класса XMLHttpRequest:

```
<объект класса XMLHttpRequest>.open( <метод отправки данных>,
<интернет-адрес>, true|false)
```

Метод отправки данных указывается в виде строки "GET" или "POST". Интернет-адрес, с которого запрашиваются данные, также указывается как строка. Если третьим параметром передано значение true, будет выполнен асинхронный запрос, если false — синхронный. Метод open не возвращает результат. Например:

```
oAJAX.open ( "GET", filename, true);
```

Задаём параметры синхронного запроса на получение файла filename, в котором хранится фрагмент HTML-кода для вывода на страницу.

Объект AJAX. Собственно отправка запроса

Собственно отправка запроса выполняется вызовом не возвращающего результат метода `send` класса `XMLHttpRequest`. Пример:

```
oAJAX.send();
```

Если данные, которые мы собираемся подгрузить, генерируются выполняющейся на стороне сервера программой, эта программа для работы может требовать какую-либо входную информацию. И отправить её можно методом GET ИЛИ POST:

- Если входные данные отправляются по методу GET, они просто добавляются к интернет-адресу, указанному вторым параметром метода `open`. Тогда метод `send` вызывается без указания параметров.
- Если входные данные отправляются по методу POST, эти данные указываются в качестве единственного параметра метода `send`. К запрашиваемому интернет-адресу они в этом случае не добавляются.

Объект AJAX. Получение данных

Получив AJAX-запрос на подгрузку данных, Web-сервер либо отправит запрошенный файл, либо вызовет серверную программу, которая получит отправленную входную информацию и сгенерирует результирующие данные, которые, опять же, будут отправлены.
Для получения данных следует требуется написать обработчик. Его можно оформить двумя способами, первый из которых является универсальным, а второй применим лишь в случае синхронного запроса.

Объект AJAX. Оформление обработчика данных

- Обработчик можно оформить в виде функции (обычной или анонимной), которую нужно присвоить свойству onreadystatechange класса XMLHttpRequest. Такой обработчик будет универсальным, подходящим и для синхронного, и для асинхронного запроса. Присвоение функции-обработчика свойству onreadystatechange следует выполнять перед вызовом метода send. Существует вероятность того, что запрошенные данные загрузятся ранее, чем для них будет указан обработчик, и такую ситуацию лучше исключить:

```
function getData() {
    // Здесь получаем и обрабатываем данные
}

oAJAX.open("GET", fileName, true);
oAJAX.onreadystatechange = getData;
oAJAX.send();
```

- Если был выполнен синхронный запрос, код обработчика можно просто поместить после вызова метода send:

```
oAJAX.open("GET", fileName, false);
oAJAX.send();
// Здесь получаем и обрабатываем данные
```

Объект AJAX. Определение успешного получения данных

Если мы используем первый способ обработки полученных данных, то следует помнить, что событие onreadystatechange будет возникать всякий раз, когда изменяется состояние ожидания этих данных (когда запрос собственно отправляется, когда данные получены, но ещё не обработаны, и др.). И ещё следует иметь в виду, что Web-сервер или серверная программа может вернуть не только запрашиваемые данные, но и сообщение об ошибке.

Поэтому, в рассматриваемом случае, понадобятся следующие свойства класса XMLHttpRequest:

- readystate — возвращает число, обозначающее состояние ожидания данных:
  - 0 — соединение с Web-сервером ещё не установлено;
  - 1 — соединение с Web-сервером установлено;
  - 2 — данные загружены, но ещё не обработаны;
  - 3 — идет обработка загруженных данных;
  - 4 — данные обработаны и могут быть извлечены;
- status — возвращает код ответа Web-сервера в виде числа: 200 (данные успешно получены) или 404 (возникла ошибка).

Пример:

```
function getData() {
    if ((oAJAX.readystate == 4) && (oAJAX.status == 200)) {
        // Данные получены. Выполняем их обработку
    }
```

Объект AJAX. Собственно получение данных

Загруженные данные можно извлечь из свойства responseText класса XMLHttpRequest. Эти данные представляют собой строку данных.

```
// Читает файл
function readOBJFile(fileName, gl) {
    var request = new XMLHttpRequest();

    request.onreadystatechange = function() {

        if (request.readyState === 4 && request.status !== 404) {

            onReadOBJFile(request.responseText, fileName, gl);

        }
    }
    request.open('GET', fileName, true); // Создать запрос
    request.send(); // Послать запрос
}
```

Объект Image

Для загрузки содержимого графического файла в Java-Script-сценарий, удобнее всего использовать объекта класса Image, который представляет графическое изображение. Для этого с помощью оператора new создаём объекта класса Image. Далее присваиваем свойству src этого объекта интернет-адрес загружаемого графического файла в виде строки:

```
var image = new Image(); // Создать объект изображения
// Зарегистрировать обработчик, вызываемый послезагрузки изображения
image.onload = function(){loadTexture(gl,n,texture,u_Sampler,image);};
// Заставить браузер загрузить изображение
image.src = '../resources/sky.jpg';
```

Ошибки в программе

В некоторых языках (например, в PHP) ошибки времени выполнения возникают из-за деления на ноль или обращения к несуществующему элементу массива. В языке JavaScript в этих случаях программа прервана не будет.
При попытке деления на ноль возвращается значение `Infinity`:

```
window.alert(5/0); // Infinity
```

При обращении к несуществующему элементу массива возвращается значение `undefined`:

```
var arr = [ 1, 2];
window.alert(arr[20]); // undefined
```

Обработка ошибок

В некоторых случаях требуется указать программе, что возникла неисправимая ошибка, и прервать выполнение всей программы. Для этого предназначен оператор `throw`:

```
if (d < 0)
    throw new Error("Переменная не может быть меньше нуля");
```

Сообщение будет выведено в окне браузера.

Объект console

Браузеры с поддержкой консоли предоставляют объект `console`, у которого есть несколько методов для вывода данных в отладочном режиме. Спецификация этого объекта описана в Console API.
Метод `console.log()` выводит данные в консоль из сценария. Он может выводить в консоль сразу несколько значений, разделенных запятыми:

```
console.log('Высота', height);
```

Чтобы разделять разные типы сообщений, которые записываются в консоль, их можно выводить с помощью трёх разных методов. Каждый из них отличается собственным цветом и значком:

1. `console.info()` используется для вывода общей информации.
2. `console.warm()` используется для вывода предупреждений.
3. `console.error()` используется для вывода ошибок.

С помощью метода `console.assert()` можно проверить, истинно ли условие, и выполнить запись в консоль, только если выражение вернуло значение `false`:

```
console.assert(this.value > 10, 'Пользователь ввел число меньше 10');
```