Компьютерные языки разметки (ПОиБМС) Основы информационных технологий (ИСиТ)

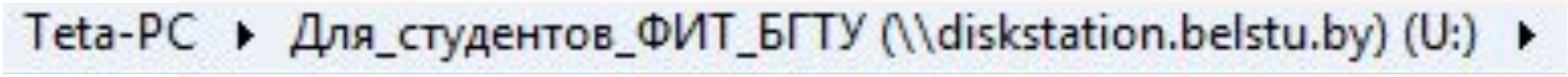
Жиляк Н.А. ИСиТ 311-1

JavaScript

http://shamansir.github.io/JavaScript-Garden/

https://habrahabr.ru/post/151070/

http://forwebdev.ru/javascript/constructors/



ПЛАН ЛЕКЦИИ:

- JS, история развития
- Спецификации JS
- Подключение JS
- Структура кода JS

JavaScript

Разработан компаниями

Sun Microsystems и Netscape

JavaScript

Скриптовый язык общего назначения, соответствующий спецификации ECMAScript.

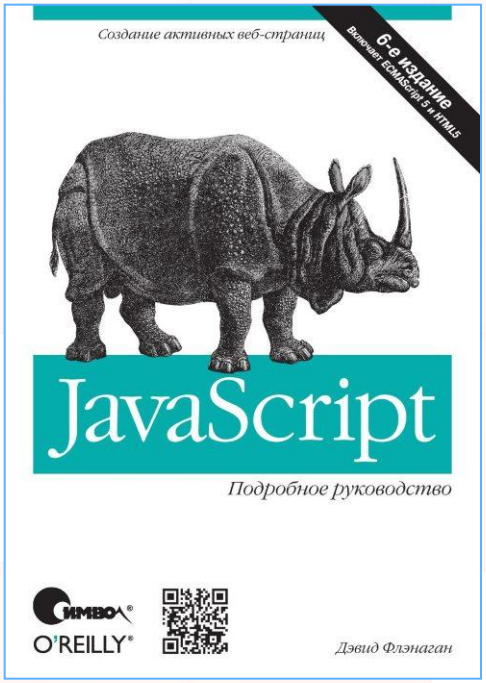
ECMAScript — это спецификация, на которой JavaScript основан.

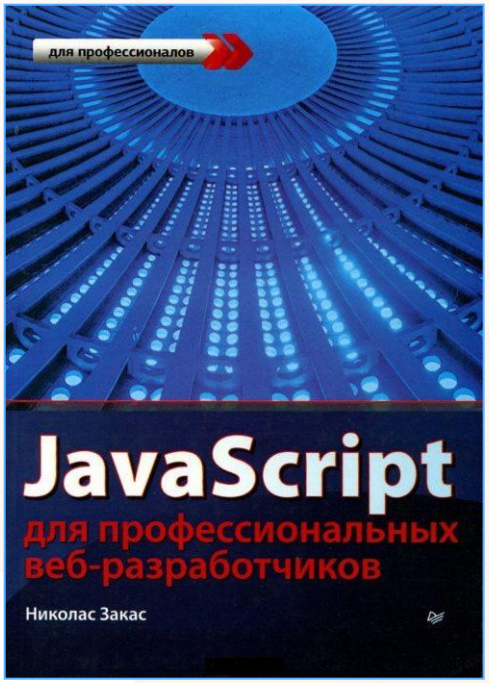
Из спецификации ECMAscript вы узнаете, как создать скриптовый язык, а из документации JavaScript— как использовать скриптовый язык.

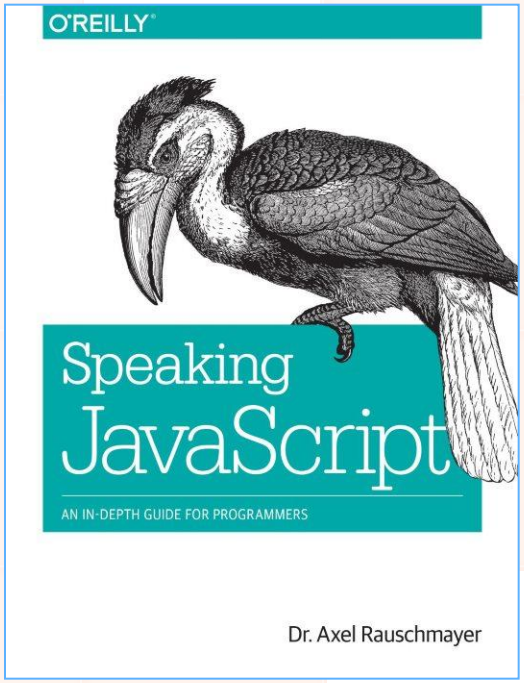
JavaScript- интерпретатор с элементами объектно-ориентированной модели

- JS использует методы и свойства объектов и событий
- Иерархия наследования свойств объектов
- Сложность: JS встраивается в HTML документ и взаимодействует с ним

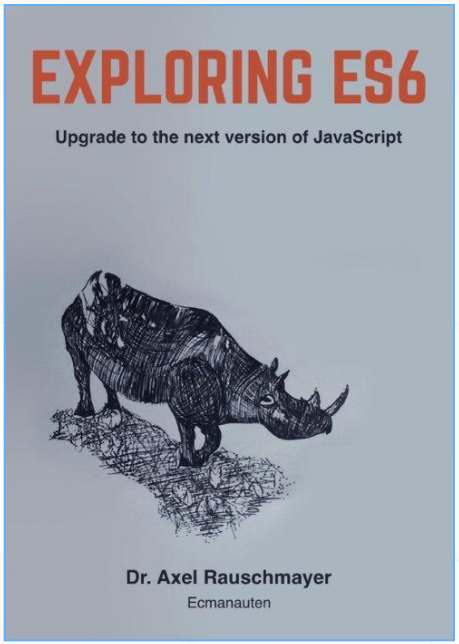
Книги – 1

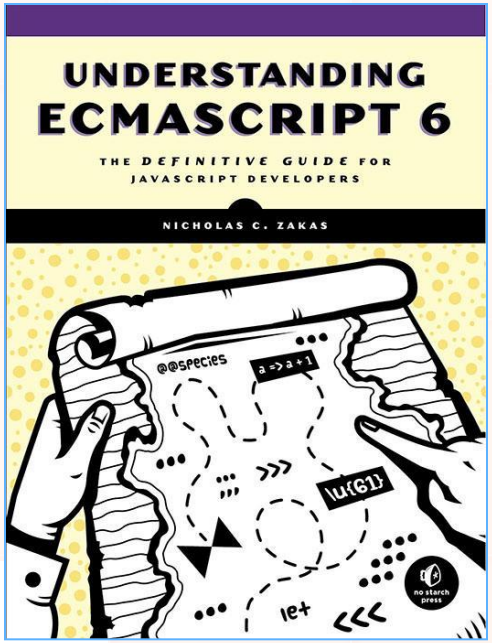

Создание активных веб-страниц
6-е издание
Включает ECMAScript 5 и HTML5
JavaScript
Подробное руководство
O'REILLY
Дэвид Флэнаган


для профессионалов
JavaScript
для профессиональных веб-разработчиков
Николас Закас


O'REILLY
Speaking JavaScript
AN IN-DEPTH GUIDE FOR PROGRAMMERS
Dr. Axel Rauschmayer

Книги – 2


EXPLORING ES6
Upgrade to the next version of JavaScript
Dr. Axel Rauschmayer
Ecmanauten


UNDERSTANDING
ECMASCRIPT 6
THE DEFINITIVE GUIDE FOR
JAVASCRIPT DEVELOPERS
NICHOLAS C. ZAKAS
@@species
a => a+1
\u{61}
let
no starch press


O'REILLY
«Вряд ли вы сможете найти более глубокий или вдумчивый анализ ES6».
Ангус Кролл, автор книги «If Hemingway Wrote JavaScript»
КАЙЛ СИМПСОН
ES6 &
НЕ ТОЛЬКО
ВЫ НЕ ЗНАЕТЕ
JS

Ссылки

1. https://learn.javascript.ru/ – учебник Ильи Кантора
2. https://developer.mozilla.org/ru/docs/Web/JavaScript
3. http://www.ecma-international.org/ecma-262/8.0/ – стандарт ECMAScript 2017
4. http://jslint.com/ – проверка корректности кода
5. http://kangax.github.io/compat-table/es2016plus/
6. https://babeljs.io/

JavaScript: общая характеристика

- *JavaScript* (JS) – прототипно-ориентированный сценарный язык программирования.

- Обычно используется как *встраиваемый* язык для программного доступа к объектам приложений.
- Широкое применение находит в браузерах для придания интерактивности веб-страницам.

История JavaScript

- **1995** – Брендан Айк (Brendan Eich) создаёт встроенный скриптовый языка для браузера Netscape Navigator. Вдохновение: ООП, функциональные языки, синтаксис C, автоматическое управление памятью.

09.1995 – LiveScript (NN 2.0 beta).

12.1995 – JavaScript (NN 2.0B3).

*) Java – «модное» слово (язык Java – май 1995).

История JavaScript

- **07.1996** – Microsoft создаёт **JScript** (Internet Explorer 3.0).

- В конце 1996 года Netscape обращается в ECMA International с просьбой утвердить стандарт JavaScript.
- **06.1997** – спецификация ECMA-262 (первая редакция).

- Язык, описанный в ECMA-262, называют **ECMAScript**.

Редакции ECMA-262

Версия (номер редакции)	Дата выпуска
1	06.1997
2	08.1998
3	12.1999
4	*Не вышла*
5	**12.2009**
2015 (6)	06.2015
2016 (7)	06.2016
2017 (8)	**27 июня 2017**

Движок JavaScript

- *Движок JavaScript* (JavaScript engine) – программа или библиотека, транслирующая и выполняющая JavaScript-код (как правило, в браузере).

- Популярные движки: V8, SpiderMonkey, Chakra, Nitro.

Что можно делать с помощью JavaScript?

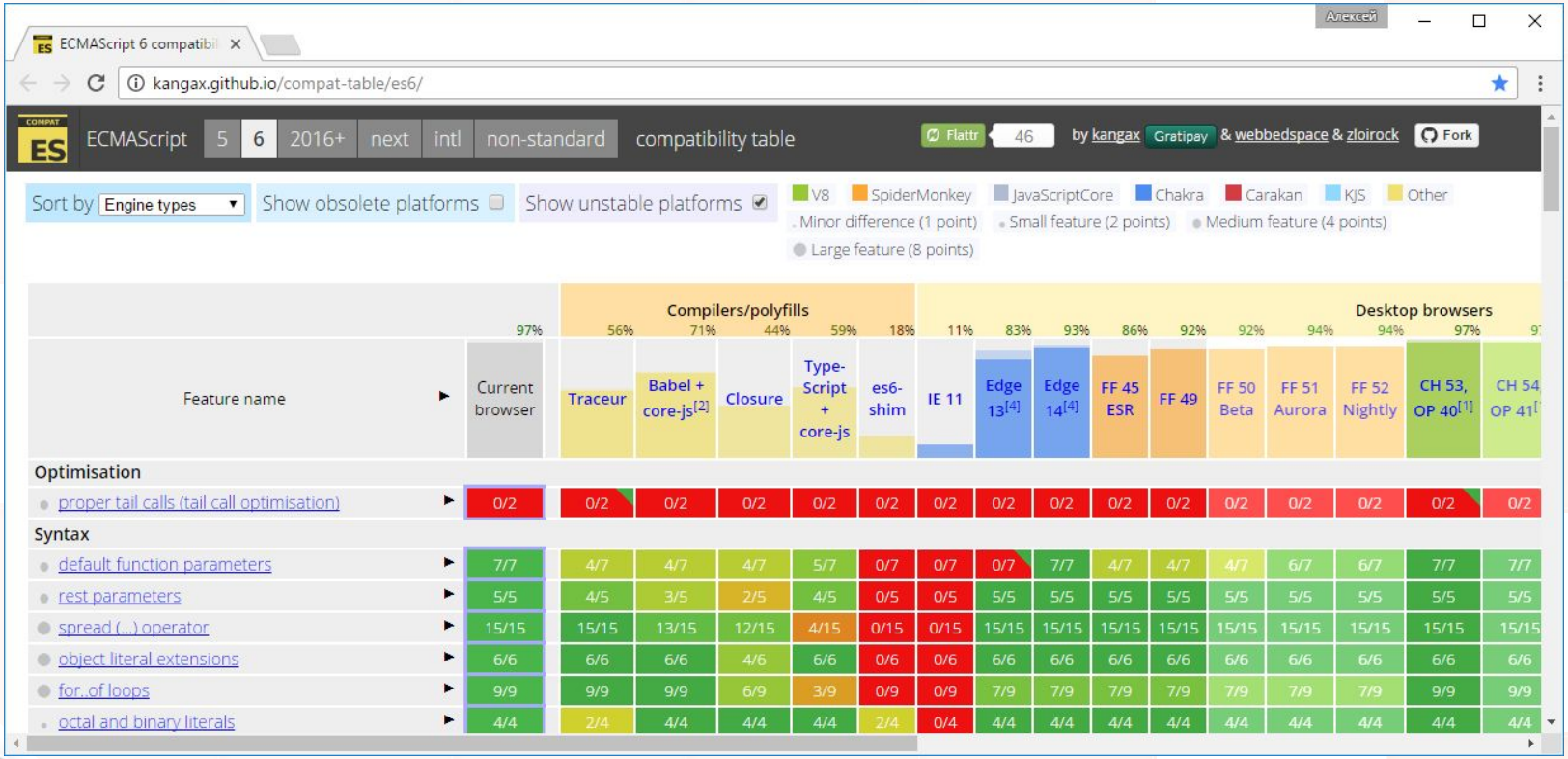
- Динамически изменять содержимое веб-страниц;
- Привязывать к элементам обработчики событий (функции которые выполнят свой код только после того, как совершатся определенные действия);
- Выполнять код через заданные промежутки времени;
- Управлять поведением браузера (*открывать новые окна, загружать указанные документы и т.д.*);
- Создавать и считывать cookies;
- Определять, какой браузер использует пользователь (*также можно определить ОС, разрешение экрана, предыдущие страницы, которые посещал пользователь и т.д.*);
- Проверять данные форм перед отправкой их на сервер и многое другое.

Сводка по движкам JavaScript

Движок	Описание
`V8`	Движок с открытым исходным кодом. Быстрый, с эффективной сборкой мусора. Используется в браузерах на основе Chromium (Chrome, Яндекс.Браузер), в Opera (с 15 версии), Node.js.
`SpiderMonkey`	Исторически первый движок. Открытый код. Содержит интерпретатор в байт-код, JIT-компилятор байт-кода (IonMonkey), сборщик мусора. Используется в Firefox и Adobe Acrobat.
`Chakra`	Выполняет JIT-компиляцию в параллельном потоке. Используется в IE 9+ и Edge (отдельная ветка).
`Nitro (JavaScriptCore)`	Особенности: прямая компиляция в машинный код (ранее – интерпретатор и JIT-компилятор). Используется Apple Safari.

Движки и возможности языка

- http://kangax.github.io/compat-table/es2016plus/



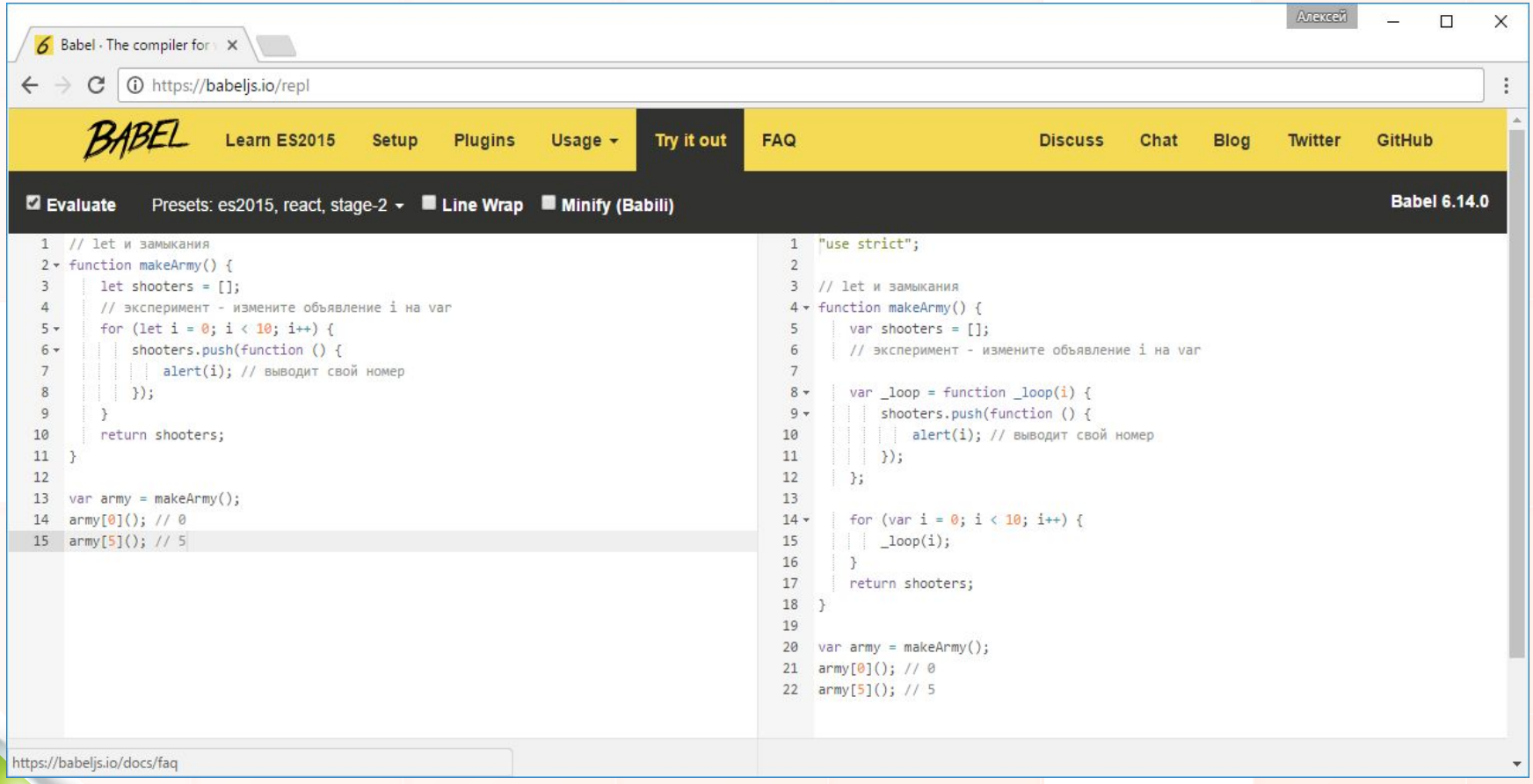
Как использовать ES2017

- Как использовать везде:
  a. При помощи *транспайлеров* (синтаксис).
  b. При помощи *полифиллов* (новые объекты и функции).

- **Babel** (или Babel.js) – популярный транспайлер для ES2015-ES2017 (предлагает для использования и полифилл).
- Сайт: https://babeljs.io/

Babel – вариант использования 1

- REPL-трансляция кода: https://babeljs.io/repl/



Babel – вариант использования 2

1. При автоматической сборке проекта задействуется транспайлер (сборка при помощи `node.js/gulp`).
2. В браузере подключаем результат работы транспайлера и добавляем полифилл.

IDE для JavaScript

1. Notepad++ (https://notepad-plus-plus.org/)
2. Sublime Text (http://www.sublimetext.com/)
3. Visual Studio 2017
4. WebStorm (https://www.jetbrains.com/webstorm/)
5. Online IDE (Cloud9, jsbin.com, jsfiddle.net)

- Скрипты могут находиться в любом месте HTML-документа
- Однако теги HTML нельзя помещать внутри JS-программы
- JS программа помещается между тегами

<script> ... </script>

- Исключение составляют обработчики событий

Главная часть → контейнер

<head>... </head>

Скрипт → HTML – документа лучше перед контейнером →

<body>... </body>

Синтаксис тега:

<script language="JavaScript">
[текст программы] </script>

Подключение JavaScript – 1

index.html

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <h1>Hello</h1>
    <script>
        var x = 10;
        alert(x);
    </script>
  </body>
</html>
```

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

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

Здесь `/path/to/script.js` – это относительный путь к файлу, содержащему скрипт (из корня сайта). Браузер сам скачает скрипт и выполнит. Можно указать и полный URL, например:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
></script>
```

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

Вы также можете использовать путь относительно текущей страницы. Например, `src="lodash.js«` - файл из текущей директории.

Чтобы подключить несколько скриптов, используйте несколько тегов:

```
<script
src="/js/script1.js"></script>
<script
src="/js/script2.js"></script> ...
```

Подключение JavaScript – 2

index.html

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <h1>Hello</h1>
    <script
src="scripts/example.js">
    </script>
  </body>
</html>
```

scripts\example.js

```
var x = 10;
alert(x);
```

На заметку:

- Как правило, в HTML пишут только самые простые скрипты, а сложные выносят в отдельный файл.
- Браузер скачает его только первый раз и в дальнейшем, при правильной настройке сервера, будет брать из своего кеша.
- Благодаря этому один и тот же большой скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера.

Выражения языка JavaScript

- Выражение - это сочетание переменных, операторов и методов, возвращающее определенное значение.
- Условные выражения используются для сравнения одних переменных с другими, а также с константами или значениями, возвращаемыми различными выражениями.

Если указан атрибут src, то содержимое тега игнорируется.

В одном теге SCRIPT нельзя одновременно подключить внешний скрипт и указать код.

Вот так не сработает:

```
<script src="file.js"> alert(1); // так как указан src, то внутренняя часть тега игнорируется </script>
```

Нужно выбрать: либо SCRIPT идёт с src, либо содержит код. Тег выше следует разбить на два: один – с src, другой – с кодом, вот так:

```
<script src="file.js"></script> <script>
alert( 1 ); </script>
```

Атрибут async

Поддерживается всеми браузерами, кроме IE9-. Скрипт выполняется полностью асинхронно. То есть, при обнаружении

```
<script async src="...">
```

браузер не останавливает обработку страницы, а спокойно работает дальше.

Когда скрипт будет загружен – он выполнится.

Атрибут `defer`

Поддерживается всеми браузерами, включая самые старые IE. Скрипт также выполняется асинхронно, не заставляет ждать страницу, но есть два отличия от `async`.

Первое – браузер гарантирует, что относительный порядок скриптов с `defer` будет сохранён.

То есть, в таком коде (с async) первым сработает тот скрипт, который раньше загрузится:

```
<script src="1.js" async></script>
<script src="2.js" async></script>
```

А в таком коде (с defer) первым сработает всегда 1.js, а скрипт 2.js, даже если загрузился раньше, будет его ждать.

```
<script src="1.js" defer></script>
<script src="2.js" defer></script>
```

Поэтому атрибут defer используют в тех случаях, когда второй скрипт 2.js зависит от первого 1.js, к примеру – использует что-то, описанное первым скриптом.

Атрибуты async/defer – только для внешних скриптов

Атрибуты async/defer работают только в том случае, если назначены на внешние скрипты, т.е. имеющие src.

При попытке назначить их на обычные скрипты <script>...</script>, они будут проигнорированы.

Итого

- Скрипты вставляются на страницу как текст в теге `<script>`, либо как внешний файл через `<script src="путь"></script>`
- Специальные атрибуты `async` и `defer` используются для того, чтобы пока грузится внешний скрипт – браузер показал остальную (следующую за ним) часть страницы. Без них этого не происходит.

Разница между async и defer:
атрибут defer сохраняет относительную последовательность скриптов, а async – нет.
Кроме того, defer всегда ждёт, пока весь HTML-документ будет готов, а async – нет.

Создание переменных

Переменные создаются либо при помощи **оператора var**,

либо при непосредственном присвоении значений с помощью **оператора присваивания (=).**

var variablename [= value | expression];

Оператор var создает новую переменную с именем **variablename**. Область действия этой переменной будет либо **локальной**, либо **глобальной** в зависимости от того, где создана переменная.

Переменная, созданная внутри функции будет недоступна за пределами функции, то есть переменная будет **локальной**.

Локальные переменные - это переменные, объявленные внутри функции JavaScript.

Они доступны только в пределах той функции, внутри которой они объявлены. При выходе из этой функции переменные уничтожаются.

Можно объявлять внутри разных функций переменные с одинаковым именем - они никак не будут пересекаться, поскольку используются только внутри функции, в которой они созданы.

Глобальные переменные объявляются вне функций и к ним могут обращаться все функции и скрипты на странице. Уничтожаются такие переменные при закрытии страницы.

Если переменную объявить без использования ключевого слова "var", то она автоматически объявляется глобальной, даже если объявление произведено внутри функции.

Например выражения x = 5; или surName = "Ivanov"; объявят переменные x и surName как глобальные, если их еще не существует.

Идентификаторы в JavaScript

- Идентификаторы состоят из **букв**, **цифр**, символов _ (подчеркивание) и $ (доллар). При этом первый символ не должен быть цифрой.
- Длина идентификатора стандартом не оговаривается.

- *) в идентификатор можно включить символ Unicode с любым шестнадцатеричным кодом *XXXX*, используя последовательность \uXXXX.

Для именования переменных JavaScript существует набор правил:

Имена переменных чувствительны к регистру (y и Y это две разных переменных)

Имена переменных должны начинаться с буквы или символов "$" и "_"

Имя переменной может состоять из любых цифр и букв латинского алфавита, а также символов "$" и "_"

В качестве имени переменной нельзя использовать зарезервированные и ключевые слова

Ключевые слова JavaScript :

break, delete, function, return, typeof, case, do, if, switch, var, catch, else, in, this, void, continue, false, instanceof, throw, while, debugger, finally, new, true, with, default, for, null, try

Проще всего понять переменную, если представить её как «коробку» для данных, с уникальным именем.

Например, переменная `message` – это коробка, в которой хранится значение `"Hello!"`:

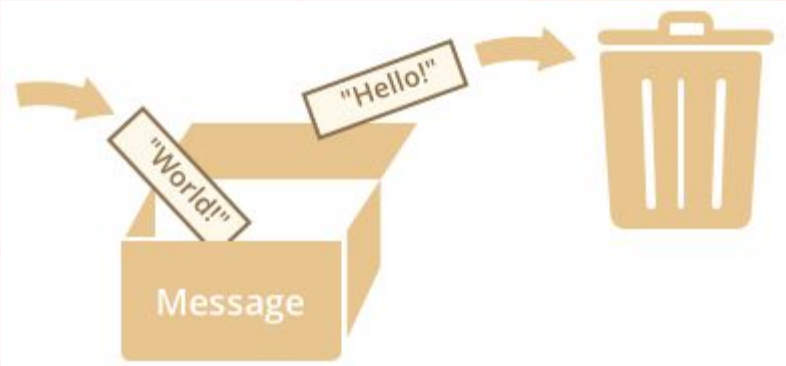
В коробку можно положить любое значение, а позже – поменять его. Значение в переменной можно изменять сколько угодно раз:

```
var message; message = 'Hello!'; message = 'World!'; // заменили значение alert( message );
```

При изменении значения старое содержимое переменной удаляется.

Можно объявить две переменные и копировать данные из одной в другую:

```
var hello = 'Hello world!';
var message; // скопировали значение
message = hello; alert( hello );
// Hello world! alert( message );
// Hello world!
```



Имена переменных

На имя переменной в JavaScript наложены всего два ограничения.

1. Имя может состоять из: букв, цифр, символов $ и _
2. Первый символ не должен быть цифрой.

Для именования переменных JavaScript существует набор правил:

Имена переменных чувствительны к регистру (y и Y это две разных переменных)

Имена переменных должны начинаться с буквы или символов "$" и "_"

Имя переменной может состоять из любых цифр и букв латинского алфавита, а также символов "$" и "_"

В качестве имени переменной нельзя использовать зарезервированные и ключевые слова

Ключевые слова JavaScript :

break, delete, function, return, typeof, case, do, if, switch, var, catch, else, in, this, void, continue, false, instanceof, throw, while, debugger, finally, new, true, with, default, for, null, try

Имена переменных

доллар '$' и знак подчеркивания '_' являются такими же обычными символами, как буквы:

```
var $ = 1; // объявили переменную с именем '$'
var _ = 2; // переменная с именем '_'
alert( $ + _ ); // 3
```

А такие переменные были бы неправильными:

```
var 1a; // начало не может быть цифрой
var my-name; // дефис '-' не является разрешенным символом
```

Регистр букв имеет значение

Переменные `apple` и `AppLE` – две разные переменные.

Русские буквы допустимы, но не рекомендуются

В названии переменных можно использовать и русские буквы, например:

```
var имя = "Вася"; alert( имя ); // "Вася«
```

Технически, ошибки здесь нет, но на практике сложилась традиция использовать в именах только английские буквы.

- Вместе с объявлением можно сразу присвоить значение: `var x = 10`.

Правила именования

- **1. Никакого транслита. Только английский.**

Неприемлемы:

```
var moiTovari; var cena; var ssilka;
```

Подойдут:

```
var myGoods; var price; var link;
```

- **2. Использовать короткие имена только для переменных «местного значения».**

Называть переменные именами, не несущими смысловой нагрузки, например `a`, `e`, `p`, `mg` – можно только в том случае, если они используются в небольшом фрагменте кода и их применение очевидно.
Вообще же, название переменной должно быть понятным. Иногда для этого нужно использовать несколько слов.

Правила именования

3. Переменные из нескольких слов пишутся вместеВотТак

```
var borderLeftWidth;
```

```
var border_left_width;
```

4. Имя переменной должно максимально чётко соответствовать хранимым в ней данным

"Лишняя" переменная – добро, а не зло!!!

Зарезервированные слова

- Они строятся по правилам записи идентификаторов, но не могут применяться в качестве пользовательских имён.

- Категории зарезервированных слов:
  - *ключевые слова*,
  - зарезервированы для использования в будущем,
  - литералы для типов `boolean` и `null`.

Зарезервированные слова ES2017

await	break	case	catch	class	Ключевые слова
const	continue	debugger	default	delete	
do	else	enum	export	extends	Зарезервированы для использования в будущем
finally	for	function	if	implements	
import	in	instanceof	interface	let	
new	package	private	protected	public	Зарезервированы для использования в будущем в строгом режиме
return	static	super	switch	this	
throw	try	typeof	var	void	
while	with	yield			

Объектно-ориентированное программирование в JavaScript

- Объекты могут иметь **свойства** и **методы**.
- **Свойства** являются значениями, которые связаны с объектами.
- **Методы** являются действиями, которые могут быть совершены над объектами.

- При обращении к методу объекта необходимо отделить его точкой от названия объекта и добавить после него круглые скобки, например

объект.метод().

- Если название метода состоит из двух (или более слов) необходимо удалить пробел между ними и начать второе слово с заглавной буквы, или заменить этот пробел на _ (знак нижнего подчеркивания),
- например объект.переключитьСкорость() или объект.переключить_скорость().

Объект - это конструкция, которая может иметь свойства и методы. Объекты могут быть созданы двумя способами:

```
//Создадим объект obj
var obj=new Object();
//Добавим объекту obj свойство name со значением 'Дмитрий'
obj.name='Дмитрий';
//Добавим объекту obj свойство age со значением 26
obj['age']=26;
//Обратимся к свойствам объекта для вывода их значений
document.write(obj.name+'<br />');
document.write(obj.age);
```

Второй способ

```
//Создадим объект obj со свойствами name и age, которые содержат
//значения 'Дмитрий' и 26
var obj={
   name:'Дмитрий',
   age:26
}
//Обратимся к свойствам объекта для вывода их значений
document.write(obj.name+'<br />');
document.write(obj.age);
```

В JavaScript Вы можете обращаться к свойствам объектов двумя способами:

1. Используя точку ('.') после имени объекта:

```
//Присваиваем свойству объекта произвольное значение
имя_объекта.свойство=значение
//Обращаемся к значению свойства объекта
имя_объекта.свойство
```

2. Заключая название свойства в квадратные скобки ([]) после имени объекта:

```
//Присваиваем свойству произвольное значение
имя_объекта['свойство']=значение
//Обращаемся к значению свойства
имя_объекта['свойство']
```

Свойства - это значения, связанные с объектами.

Например, объект человек может иметь следующие свойства: имя, возраст, профессия.

Пример

```
//Создадим объект human со свойствами name, age и job
var human={
   name:'Дмитрий',
   job:'Дизайнер',
   age:26
}
//Выведем значения свойств на страницу
document.write(human.name+'<br />');
document.write(human.job+'<br />');
document.write(human.age);
```

```
  var flag;
//Создадим объект human со свойствами name, age и job
var human={
   name:'Дмитрий',
   job:'Дизайнер',
   age:26
}
//Добавим объекту метод go()
human.go = function () {
   document.write(this.name+' идет.<br />');
   var a=true;
   return a;
}
//Добавим объекту метод sit()
human.sit = function () {
   document.write(this.name+' сел.<br />');
   flag=true;
}
//Добавим объекту метод think()
human.think = function () {
   if (flag==true)
      document.write(this.name+' начал мыслительную деятельность.<br />');
   else
     document.write(this.name+' не может начать мыслительную деятельность на
     ходу.<br />');
}
human.about= function () {
  document.write('Объекта зовут <b>'+this.name+'</b> он работает <b>'+this.job+'ом</b>
  и ему <b>'+this.age+'</b> лет.');
}
//Обратимся к методам объекта human
human.go();
human.think();
human.sit();
human.think();
human.about();
```

Конструкторы объектов

- В JavaScript существует еще один способ создания объектов - с помощью **конструктора объектов**.
- Конструктор объектов - это шаблон, на основе которого создаются объекты. Написанный один раз конструктор, позволяет создать неограниченное количество объектов любой сложности написав лишь одну строчку кода.
- Преимущества данного способа заключается в том, что он позволяет разделить логику создания объекта от его использования. Например, один человек может специализироваться на создании сложных конструкторов объектов, а другой на написании на основе этих объектов конечных программ под заказ.

```
//Создадим конструктор, который будет создавать объекты со свойствами name, age и
//job и методом who
function Human(name,age,job) {
   //Свойства шаблона
   this.name=name;
   this.age=age;
   this.job=job;
   //Методы шаблона
   this.who=function who() {
      document.write('Я<b> ' + this.name + '</b> мне <b>'+ this.age + ' </b>лет.');
      document.write(' Я работаю <b>' + this.job + 'ом</b>.<br />' );
   }
}
//Теперь мы можем создавать объекты следующим образом
human1=new Human('Дмитрий', 26, 'Дизайнер');
//Обратимся к методу who созданного объекта
human1.who();
human2=new Human('Станислав', 29, 'Программист');
human2.who();
human3=new Human('Сергей', 35, 'Менеджер');
human3.who();
```

- **Обратите внимание:** ключевое слово **this** используется для обозначения "текущего" объекта. К примеру при создании объекта human1, this в конструкторе Human подменяется на human1, а при создании объекта human2 подменяется на human2.
- **Обратите внимание:** оператор **new** обязательно должен присутствовать при создании экземпляра объекта.

Структура кода

команды: `alert('Привет, мир!')` выводит сообщение.
Для того, чтобы добавить в код ещё одну команду – можно поставить её после точки с запятой.
Например, вместо одного вызова `alert` сделаем два:

```
alert('Привет'); alert('Мир');
```

Как правило, каждая команда пишется на отдельной строке – так код лучше читается:

```
alert('Привет'); alert('Мир');
```

в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт, которому

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

```
alert('Привет') alert('Мир')
```

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

Операторы комментариев и примечаний

```
// Текст комментариев
/* Текст
комментариев
*/
```

Первый комментарий может иметь только одну строку, второй несколько.

Комментарии нужны для пояснений или для временного исключения некоторых фрагментов программы во время отладки.

Шесть типов данных, typeof

Число «number»

```
var n = 123; n = 12.345;
```

Единый тип *число* используется как для целых, так и для дробных чисел. Существуют специальные числовые значения **Infinity (бесконечность) и NaN (ошибка вычислений).**

Шесть типов данных, typeof

Например, бесконечность `Infinity` получается при делении на ноль:

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

Ошибка вычислений `NaN` будет результатом некорректной математической операции, например:

```
alert( "нечисло" * 2 ); // NaN, ошибка
```

Эти значения формально принадлежат типу «число», хотя, конечно, числами в их обычном понимании не являются.

Строка «string»

```
var str = "Мама мыла раму"; str =
'Одинарные кавычки тоже подойдут';
```

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

Тип *символ* не существует, есть только *строка*. В некоторых языках программирования есть специальный тип данных для одного символа. Например, в языке C это `char`. В JavaScript есть только тип «строка» `string`.

Булевый (логический) тип «boolean»

У него всего два значения: `true` (истина) и `false` (ложь).

Как правило, такой тип используется для хранения значения типа да/нет, например:

```
var checked = true; // поле формы помечено галочкой
checked = false; // поле формы не содержит галочки
```

Специальное значение «null»

Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

```
var age = null;
```

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

Специальное значение «undefined»

Значение `undefined`, как и `null`, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено». Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть `undefined`:

```
var x; alert( x ); // выведет "undefined“
```

Специальное значение «undefined»

Можно присвоить undefined и в явном виде, хотя это делается редко:

```
var x = 123;
x = undefined;
alert( x ); // "undefined“
```

В явном виде undefined обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого» или «неизвестного» значения используется null.

Объекты «object»

Первые 5 типов называют *«примитивными»*. Особняком стоит шестой тип: *«объекты»*.

Он используется для коллекций данных и для объявления более сложных сущностей. Объявляются объекты при помощи фигурных скобок {...}, например:

```
var user = { name: "Вася" };
```

Объекты как ассоциативные массивы.

Оператор typeof

Оператор `typeof` возвращает тип аргумента.

У него есть два синтаксиса: со скобками и без:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функции: `typeof(x)`.

Результатом typeof является строка, содержащая тип:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof true // "boolean"
typeof "foo" // "string"
typeof {} // "object" typeof null //
"object" (1)
typeof function(){} // "function" (2)
```

Последние две строки помечены, потому что typeof ведет себя в них по-особому.

Результат `typeof null == "object"` – это официально признанная ошибка в языке, которая сохраняется для совместимости. На самом деле `null` – это не объект, а отдельный тип данных.

Функции мы пройдём чуть позже. Пока лишь заметим, что функции не являются отдельным базовым типом в JavaScript, а подвидом объектов. Но typeof выделяет функции отдельно, возвращая для них `"function"`. На практике это весьма удобно, так как позволяет легко определить функцию.

Операции присваивания

=	Прямое присваивание значения левому операнду
+=	Складывает значения левого и правого операндов и присваивает результат левому операнду
+	Складывает значения левого и правого операндов и присваивает результат левому операнду
++	Увеличивает значение левого операнда (правый может отсутствовать)
-=	Вычитает значения левого и правого операндов и присваивает результат левому операнду
-	Вычитает значения левого и правого операндов и присваивает результат левому операнду
--	Уменьшает значение левого операнда (правый может отсутствовать)
*	Умножает значения левого и правого операндов и присваивает результат левому операнду
*=	Умножает значения левого и правого операндов и присваивает результат левому операнду
/	Делит значения левого на правого операндов и присваивает результат левому операнду
/=	Делит значения левого на правого операндов и присваивает результат левому операнду

nval *=10;
вместо:
nval = nval * 10;

Операции сравнения

==	Равенство (равно)
!=	Не равно
!	Логическое отрицание
>=	Больше или равно
<=	Меньше или равно
>	Больше
<	Меньше (по возможности желательно воздержаться от применения этого типа)

if mvar <h . . . . . .bgcolor- может интерпретироваться как начало заголовка HTML

!Теги HTML в JS программах недопустимы!

Простейшие операции

Теперь мы можем перейти к изучению операций.
Операции в JavaScript условно можно разделить на несколько видов:

- Арифметические операторы
- Операторы присваивания
- Операторы сравнения
- Логические операторы

Начнем с самых простых - **Арифметические операторы**.

Все результаты примеров будем рассматривать при исходных данных **y = 5**

Сложение "+"

выражение x = y + 2 даст результат x = 7; y = 5;

Вычитание "-"

выражение x = y - 2 даст результат x = 3; y = 5;

Умножение "*"

выражение x = y * 2 даст результат x = 10; y = 5;

Деление "/"

выражение x = y / 2 даст результат x = 2.5; y = 5;

Более сложные **Арифметические операторы,** начальное значение **y = 5:**
Остаток от деления (он же - деление по модулю) "%"
выражение x = y % 2 даст результат x = 1; y = 5;

Инкремент "++"
Эта операция производит увеличение аргумента на единицу, т.е. выражение x++ будет эквивалентно выражению x = x + 1.
Порядок применения инкремента в javascript имеет значение, например:

выражение x = ++y даст результат x = 6, y = 6, так как **вначале** увеличивается значение переменной y на единицу и затем переменной x присваивается значение переменной y;

а выражение x = y++ даст результат x = 5, y = 6, поскольку в данном случае операция инкрементирования (увеличения) происходит ПОСЛЕ того как переменной x присвоили значение из переменной y.

Декремент "--"

А эта операция производит уменьшение значения переменной на единицу, т.е. x-- эквивалентно x = x - 1. Порядок применения декремента в JavaScript также имеет значение, например:

- выражение x = --y даст результат x = 4, y = 4 - тут мы вначале уменьшаем на единицу переменную y, а затем присваиваем переменной x значение переменной y;
  а выражение x = y-- даст результат x = 5,
- y = 4 - так как операция декремент была выполнена ПОСЛЕ того как переменной x присвоили значение из переменной y.

Операторы присваивания бывают такие:

В примерах будем рассматривать x = 10; y = 5;

Оператор = это обычный оператор присваивания.
Выполнение x = y приведет к x = 5.

Некоторые арифметические операторы можно использовать вместе с оператором присваивания:

Оператор += Это присваивание со сложением
Выполнение x += y приведет к x = 15, эквивалентно x = x + y

Оператор -= Это присваивание с вычитанием
Выполнение x -= y приведет к x = 5, эквивалентно x = x - y

Оператор *= Присваивание с умножением
Выполнение x *= y приведет к x = 50, эквивалентно x = x * y

Оператор /= Присваивание и деление.
Выполнение x /= y приведет к x = 2, эквивалентно x = x / y

Оператор %= Присваивание с операцией "остаток от деления"

Выполнение x %= y приведет к x = 0, эквивалентно x = x % y

Логические операции

И **&&**

ИЛИ **||**

Эти операции применимы только к булевым значениям

Например:

bvar1 = true;

bvar2 = false;

bvar1 || bvar2

bvar1 && bvar2 false

```
if ((bvar1 && bvar2) || bvar3) {
  function1();
}
    else {
  function2();
}
```

"Активизировать функцию function1(), если обе переменные bvar1 и bvar2 содержат значения true, или хотя бы bvar3 содержит true, иначе вызвать функцию function2 "

Побитовые операторы

интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Они производят операции, используя двоичное представление числа, и возвращают новую последовательность из 32 бит (число) в качестве результата.

Оператор	Использование	Описание
Побитовое И (AND)	a & b	Ставит 1 на бит результата, для которого соответствующие биты операндов равны 1.
Побитовое ИЛИ (OR)	a | b	Ставит 1 на бит результата, для которого хотя бы один из соответствующих битов операндов равен 1.
Побитовое исключающее ИЛИ (XOR)	a ^ b	Ставит 1 на бит результата, для которого только один из соответствующих битов операндов равен 1 (но не оба).
Побитовое НЕ (NOT)	~a	Заменяет каждый бит операнда на противоположный.
Левый сдвиг	`a << b`	Сдвигает двоичное представление a на b битов влево, добавляя справа нули.
Правый сдвиг, переносящий знак	`a >> b`	Сдвигает двоичное представление a на b битов вправо, отбрасывая сдвигаемые биты.
Правый сдвиг с заполнением нулями	`a >>> b`	Сдвигает двоичное представление a на b битов вправо, отбрасывая сдвигаемые биты и добавляя нули слева.

Итого

- Бинарные побитовые операторы: `&` `|` `^` `<<` `>>` `>>>`.
- Унарный побитовый оператор один: `~`.

  Как правило, битовое представление числа используется для:
- Округления числа: `(12.34^0) = 12`.
- Проверки на равенство `-1`: `if (~n) { n не -1 }`.
- Упаковки нескольких битововых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором &.
- Других ситуаций, когда нужны битовые маски.

Литералы

- *Литерал* – последовательность символов в исходном коде, которая представляет фиксированное значение некоторого типа данных.

- Литерал – это константа, непосредственно включённая в текст скрипта.

Литералы в JavaScript

- Вот так в JavaScript выглядит обычное объявление и инициализация переменной:
- `var x = "This is a string variable";`

- Единственный способ определить тип `x` – по литералу справа (в данном примере тип `x` – это `string`).

Целые десятичные числа

- Для записи целых **десятичных** чисел используются цифры `0`…`9`. Перед числом допустимы знаки `+` или `-`:
- `var x = -123;`

Целые шестнадцатеричные числа

- **Шестнадцатеричное** число начинается с `0x` или `0X`. Используются шестнадцатеричные цифры `0…9` `a…f` `A…F`. Перед числом допустимы знаки `+` или `–`:
- `var x = 0x123abc;`
- `var y = -0XFFF;`

Новые литералы чисел в ES2015

- Целые **двоичные литералы**: начинаем с `0b` или `0B` и используем цифры `0` и `1`. Впереди допускается `+` или `-`.

- Целые **восьмеричные литералы**: начинаем с `0o` или `0O` и используем цифры от `0` до `7`. Впереди можно `+` или `-`.

Замечание о целых числах

- Максимальное целое число, хранимое **точно** = 2^{53}:

```
var biggestInt = 9007199254740992;

// проверьте, чему равно biggestIntAndOne
var biggestIntAndOne = 9007199254740993;
alert(biggestIntAndOne);
```

Литералы вещественных чисел

- Синтаксическая форма литерала **вещественных чисел** (используются только десятичные *цифры*):
  - [*цифры*][.*цифры*][(E|e)[(+|-)]*цифры*]

- Перед числом можно указать знак + или –:

- `var x = 3.1415926;`
- `var y = -4.08e75;`

Строковые литералы

- Строковый литерал – это последовательность Unicode-символов в парных одинарных или двойных кавычках:

```
var st1 = "Normal";
var st2 = 'Also normal';
var oneChar = "A";
var empty = "";
```

Управляющие символы внутри строки

- `\0` Символ NUL
- `\b` «Забой»
- `\t` Горизонтальная табуляция
- `\n` Перевод строки
- `\v` Вертикальная табуляция
- `\f` Перевод страницы
- `\r` Возврат каретки
- `\"` Двойная кавычка
- `\'` Одинарная кавычка
- `\\` Обратный слэш
- `\xXX` Символ Latin-1, заданный двумя шестнадцатеричными цифрами XX
- `\uXXXX` Символ Unicode, четыре шестнадцатеричных цифры XXXX

Обратный слэш – нюанс

- **Внимание**: если после обратного слэша записан «неожиданный» символ, то слэш игнорируется:

```
var x = "\A\L\E\X";
alert(x);  // выведет ALEX
```

Шаблонные литералы

- *Шаблонные литералы* – новый вид строковых литералов в ES2015. Записываются в *обратных кавычках*:

```
var str = `template string`;
```

- Шаблонный литерал может содержать перевод строки (строка так и отображается – с переводом):

```
var str = `template
string`;
```

Шаблонные литералы

- При помощи `${…}` в шаблонный литерал можно вставить произвольное выражение:

```
var x = 2;
var y = 3;
alert(`${x} + ${y} = ${x + y}`);    // 2 + 3 = 5
```

Литералы для типов boolean и null

- `true false` два литерала для типа `boolean`
- `null` один возможный литерал для типа `null`

Преобразование типов

- Большинство операторов JavaScript требуют операндов определённых типов.
- Например, оператор `.` («точка», доступ к свойству объекта) требует в качестве операнда объект.

- В случае несоответствия типов JavaScript выполняет **неявное преобразование** типов операндов.

Преобразование null, undefined, boolean

	В строку	В число	В boolean	В объект
`undefined`	`"undefined"`	`NaN`	`false`	ошибка `TypeError`
`null`	`"null"`	`0`	`false`	ошибка `TypeError`
`true`	`"true"`	`1`		`new Boolean(true)`
`false`	`"false"`	`0`		`new Boolean(false)`

Преобразование строк

	В число	В boolean	В объект
`""` (пустая строка)	`0`	`false`	`new String("")`
`"1.2"` (непустая строка, число)	`1.2`	`true`	`new String("1.2")`
`"one"` (непустая строка, не число)	`NaN`	`true`	`new String("one")`

Преобразование чисел

	В строку	В boolean	В объект
`0`	`"0"`	`false`	`new Number(0)`
`-0`	`"0"`	`false`	`new Number(-0)`
`NaN`	`"NaN"`	`false`	`new Number(NaN)`
`Infinity`	`"Infinity"`	`true`	`new Number(Infinity)`
`-Infinity`	`"-Infinity"`	`true`	`new Number(-Infinity)`
`1` (конечное, ненулевое)	`"1"`	`true`	`new Number(1)`

Преобразование массивов

	В строку	В число	В boolean
`[]` (пустой массив)	`""`	`0`	`true`
`[8]` (один элемент, у которого строковое представление преобразуется в число)	`"8"`	`8`	`true`
`["A"]` (любой другой массив)	метод `join()`	`NaN`	`true`

Преобразование объектов

- *Примитивное значение* – значение одного из типов `number`, `string`, `boolean`, `null`, `undefined`.

- В стандарте описана внутренняя функция `ToPrimitive(input, PreferredType?)`:
  - `input` – что преобразуем
  - `PreferredType` – целевой тип (необязательный параметр)

Алгоритм работы функции ToPrimitive()

- Если `input` – объект, а `PreferredType` – это `string`:

1) Если у объекта есть метод `toString()`, и этот метод возвращает примитивное значение `=>` вернуть это значение.
2) (иначе) Если у объекта есть `valueOf()`, который возвращает примитивное значение `=>` вернуть это значение.
3) (иначе) сгенерировать ошибку `TypeError`.

Алгоритм работы функции ToPrimitive()

- Если `input` – объект, а `PreferredType` – это `number`, в предыдущем алгоритме шаги 1-2 меняются местами.

- Если `input` – объект, а `PreferredType` не задано:
  - для объектов `Date` полагаем, что `PreferredType` – `string`;
  - для любых других объектов `PreferredType` – это `number`.

- Если `input` – примитивное значение, возвращаем `input`.

Преобразование объектов

- Преобразование объекта в строку (число):
  - вызвать `ToPrimitive()`, где второй аргумент `string` (`number`);
  - результат функции преобразовать в строку (число).

- Преобразование объектов в `boolean` – **всегда** `true`.

Неявные и явные преобразования

- Сейчас были описаны *неявные преобразования* типов.

- *Явное преобразование* инициируется программистом и выполняется при помощи специальных методов конвертации или вызовов функций-конструкторов объектов-обёрток.

Вспомогательные функции parseInt, toString

- `parseInt("11000", 2)` – переводит строку с двоичной записью числа в число.
- `n.toString(2)` – получает для числа n запись в 2-ной системе в виде строки.

Например:

```
var access = parseInt("11000", 2); // получаем число из строки
alert( access ); // 24, число с таким 2-ным представлением
var access2 = access.toString(2); // обратно двоичную строку из числа alert( access2 ); // 11000
```

Взаимодействие с пользователем: alert, prompt, confirm

базовые UI операции, которые позволяют работать с данными, полученными от пользователя.

alert

Синтаксис:

```
alert(сообщение)
```

`alert` выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

```
alert( "Привет" );
```

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «ОК».

prompt

Функция prompt принимает два аргумента:

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

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

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

Вызов `prompt` возвращает то, что ввёл посетитель – строку или специальное значение `null`, если ввод отменён.

```
alert('Вам ' + years + ' лет!')
```

Всегда указывайте default

Второй параметр может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию "undefined". Запустите этот код в IE, чтобы понять о чём речь:

```
var test = prompt("Тест");
```

Поэтому рекомендуется *всегда* указывать второй аргумент:

```
var test = prompt("Тест", ''); // <-- так лучше
```

confirm

Синтаксис:

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

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

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

Например:

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

Итог

- `alert` выводит сообщение.
- `prompt` выводит сообщение и ждёт, пока пользователь введёт текст, а затем возвращает введённое значение или `null`, если ввод отменён (CANCEL/`Esc`).
- `confirm` выводит сообщение и ждёт, пока пользователь нажмёт «OK» или «CANCEL» и возвращает `true/false`.

Условные операторы: if, '?'

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');
 if (year != 2011) alert( 'А вот и неправильно!' );
```

Оператор `if` («если») получает условие, в примере выше это `year != 2011`. Он вычисляет его, и если результат – `true`, то выполняет команду.

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

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

Рекомендуется использовать фигурные скобки всегда, даже когда команда одна.

Преобразование к логическому типу

Оператор `if (...)` вычисляет и преобразует выражение в скобках к логическому типу.

В логическом контексте:

- Число `0`, пустая строка `""`, `null` и `undefined`, а также `NaN` являются `false`.
- Остальные значения – `true`.

Преобразование к логическому типу

Например, такое условие никогда не выполнится:

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

…А такое – выполнится всегда:

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

Можно и просто передать уже готовое логическое значение, к примеру, заранее вычисленное в переменной:

```
var cond = (year != 2011); // true/false
if (cond)
{ ... }
```

Неверное условие, else

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

```
var year = prompt('Введите год появления стандарта ECMA-262 5.1', '');
if (year == 2011) { alert( 'Да вы знаток!' ); }
else { alert( 'А вот и неправильно!' ); // любое значение, кроме 2011 }
```

Несколько условий, else if

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

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');
if (year < 2011) { alert( 'Это слишком рано..' ); }
else if (year > 2011) { alert( 'Это поздновато..' ); }
else { alert( 'Да, точно в этом году!' ); }
```

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

Оператор вопросительный знак „?“

Иногда нужно в зависимости от условия присвоить переменную. Например:

```
var access;
var age = prompt('Сколько вам лет?',
'');
if (age > 14) { access = true; }
else { access = false; } alert(access);
```

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

Оператор вопросительный знак „?“

Он состоит из трех частей:

условие ? значение1 : значение2

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

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

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

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

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

Несколько операторов "?"

Последовательность операторов '?' позволяет вернуть значение в зависимости не от одного условия, а от нескольких.

Например:

```
var age = prompt('возраст?', 18);
var message = (age < 3) ? 'Здравствуй, малыш!' :
(age < 18) ? 'Привет!' :
(age < 100) ? 'Здравствуйте!' : 'Какой необычный возраст!';
alert( message );
```

Оператор сравнения ?

(условное выражение) ? операторы_1 : операторы_2

Присваивание значений переменным:

```
type_time = (hour >= 12) ? "PM" : "AM"
```

```
if (hour >= 12)
type_time="PM";
else
type_time="AM";
```

Преобразование типов для примитивов

- Строковое преобразование.
- Численное преобразование.
- Преобразование к логическому значению.

Строковое преобразование.

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.
Например, его производит функция `alert`

```
var a = true;
alert( a ); // "true"
```

Можно также осуществить преобразование явным вызовом `String(val)`:

```
alert( String(null) === "null" ); // true
```

Как видно из примеров выше, преобразование происходит наиболее очевидным способом, «как есть»: `false` становится `"false"`, `null` – `"null"`, `undefined` – `"undefined"` и т.п.

Также для явного преобразования применяется оператор `"+"`, у которого один из аргументов строка. В этом случае он приводит к строке и другой аргумент, например:

```
alert( true + "test" ); // "truetest"
alert( "123" + undefined ); // "123undefined"
```

Численное преобразование.

Численное преобразование происходит в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений ===, !==).
Для преобразования к числу в явном виде можно вызвать `Number(val)`, либо, что короче, поставить перед выражением унарный плюс "+":

```
var a = +"123"; // 123
var a = Number("123"); // 123, тот же эффект
```

Значение	Преобразуется в...
undefined	NaN
null	0
true / false	1 / 0
Строка	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки "считывается" число, при ошибке результат NaN.

Преобразование к логическому значению.

Преобразование к `true/false` происходит в логическом контексте, таком как `if(value)`, и при применении логических операторов.

Все значения, которые интуитивно «пусты», становятся `false`.

Их несколько: `0`, пустая строка, `null`, `undefined` и `NaN`.

Остальное, в том числе и любые объекты – `true`.

В отличие от многих языков программирования (например PHP), `"0"` в JavaScript является `true`, как и строка из пробелов:

```
alert( !!"0" ); // true
alert( !!" " ); // любые непустые строки, даже из пробелов - true!
```

Значение	Преобразуется в...
undefined, null	false
Числа	Все true, кроме 0, NaN -- false.
Строки	Все true, кроме пустой строки "" -- false
Объекты	Всегда true

Базовые операторы зыка JS

- Каждый оператор, если он занимает единственную строку, имеет разграничивающую точку с запятой **(;)**, обозначающую окончание оператора.
- Каждый оператор имеет собственный синтаксис.
- Синтаксис оператора - это набор правил, определяющих обязательные и допустимые для использования в данном операторе значения.
- Значения, присутствие которых является необязательным, при описании синтаксиса принято заключать в квадратные скобки, например [value].

При несоблюдении правил синтаксиса произойдет ошибка компиляции.

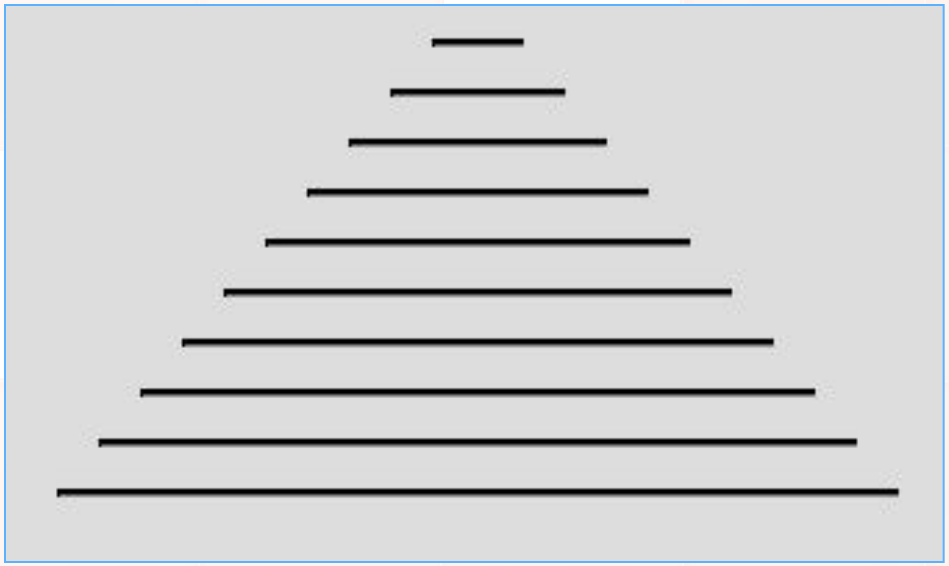
Операторы циклов

```
for ([инициализация начального значения;]
   [условие;] [механизм обновления счетчика,
   шаг]) {
программный блок
}
```

Оператор For позволяет многократно выполнять операторы в JS-программе. Оператор For может быть использован для выполнения одного или нескольких операторов. Фигурные скобки можно опустить, если тело цикла содержит только один оператор. Все параметры оператора For являются необязательными и используются для управления процессом выполнения цикла. При применении всех параметров каждую часть нужно отделять точкой с запятой (;).

Пример вывода в окне браузера горизонтальных линий

```
<html>
<head>
<script language ="JavaScript">
<!--
function testloop() {
var String1 = '<hr align="center" width="' ;
document.open();
for (var size = 5; size <= 50; size+=5)
document.writeln (String1+ size+'%">');
document.close();
}
//-->
</script>
</head>
<body>
<form>
<input type="button"
value="Test the loop"
onClick="testloop()">
</form>
</body>
</html>
```



Цикл while

```
while (условие) {
программный блок
}
```

При помощи оператора **while** можно выполнять один или несколько операторов до тех пор, пока не будет удовлетворено условие.

Если в теле цикла выполняется несколько операторов, их необходимо заключить в фигурные скобки.

Пример вывода таблицы умножения

```
<html>
<head>
<script language ="JavaScript">
function ftable(inum) {
   var iloop = 1;
   document.writeln ("ТАБЛИЦА УМНОЖЕНИЯ ДЛЯ: <b>" + inum + "</b><hr><pre>");
      /* в параметрах функции writeln применены теги HTML - это допустимо.
while (iloop <= 10) {
document.writeln(iloop + " x "+ inum + " = " + (iloop*inum));
iloop ++;
}
document.writeln("</pre>");
}
ftable(prompt ("Введите число: ", 10));
</script>
</head>
</html>
```

Теги HTML в тексте программы на JS недопустимы

Выход из цикла - оператор break

Оператор break используется для выхода из какого-либо цикла, например из цикла for или while.

Выполнение цикла прекращается в той точке, в которой размещен этот оператор, а управление передается следующему оператору, находящемуся непосредственно после цикла.

Пример применения оператора break

```
<html>
<script language ="JavaScript">
function btest() {
var index = 1;
while (index <= 10) {
if (index = = 6)
break;
index ++;
}
//После отработки оператора break управление
   переходит сюда.
}
btest();
</script>
</html>
```

Цикл while будет всегда завершаться после первых шести итераций, а значение переменной index никогда не достигнет 10-ти

Продолжение цикла - оператор continue

- Оператор continue используется для прерывания выполнения блока операторов, которые составляют тело цикла и продолжения цикла в следующей итерации. В отличие от оператора break, оператор continue не останавливает выполнение цикла, а наоборот запускает новую итерацию.
- Если в цикле while идет просто запуск новой итерации, то в циклах for запускает с обновленным шагом.

Конструкция switch

Конструкция `switch` заменяет собой сразу несколько `if`.
Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис

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

- Переменная `x` проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее.
- Если соответствие установлено – switch начинает выполняться от соответствующей директивы `case` и далее, *до ближайшего* `break` (или до конца `switch`).
- Если ни один `case` не совпал – выполняется (если есть) вариант `default`.
  При этом `case` называют *вариантами* `switch`.

Пример использования switch (сработавший код выделен):

```
var a = 2 + 2;
switch (a)
{ case 3: alert( 'Маловато' );
break; case 4: alert( 'В точку!' );
break; case 5: alert( 'Перебор' );
break; default: alert( 'Я таких значений не знаю' );
}
```

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert('В точку!') и далее, до ближайшего break, который прервёт выполнение.

Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.

Определение функции

```
function functionname (arg, . . .) {
блок операторов
}
```

- **Функция - это блок из одного или нескольких операторов.**
- **Блок выполняет определенные действия, а затем, возможно, возвращает значение.**
- **В языке JS процедуры - подпрограммы не возвращающие значений, не различаются.**
- **Все подпрограммы описываются функциями, а если в функцию или из нее не передаются параметры - то после имени функции ставятся круглые скобки без параметров.**
- **Если функция имеет несколько аргументов, они отделяются запятой.**
- **В языке JS внутри одной функции не может существовать другой функции.**
- **Фигурные скобки определяют тело функции.**
- **Функция не может быть выполнена до тех пор, пока не будет явного обращения к ней.**
- **Если необходимо, чтобы функция возвращала определенное значение, следует использовать необязательный оператор return, при этом указав в нем выражение, значение которого требуется возвратить.**

Возврат значения функциями - оператор return

```
return (value);
return value;
```

Оператор return завершает выполнение функции и возвращает значение заданного выражения.

Скобки в этом операторе можно не использовать.

Оператор return может отсутствовать в функции, если функция не возвращает значение.

return для возврата массива

```
function retarray() {
var sarray = new Object();
sarray[1] = "Java";
sarray[2] = "Script";
return (sarray);
}
```

Обращение к аргументам функции при помощи массива argunents[ ]

```
function showargs() {
   arglist = "";
   for (var n=0; n <= arguments.length; n++)
{
        arglist += n +"." + arguments[n] + "\n";
}
alert(arglist);
}
```

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

- Если функция объявлена в *основном потоке кода*, то это Function Declaration.
- Если функция создана как *часть выражения*, то это Function Expression.

Между этими двумя основными способами создания функций есть следующие различия:

	Function Declaration	Function Expression
Время создания	До выполнения первой строчки кода.	Когда управление достигает строки с функцией.
Можно вызвать до объявления	Да (т.к. создаётся заранее)	Нет
Условное объявление в if	Не работает	Работает

Некоторые функции решают создавать как `var func = function()`,
но в большинстве случаев обычное объявление функции – лучше.

Если нет явной причины использовать Function Expression – предпочитайте Function Declaration.

- Если функция задана как Function Expression, ей можно дать имя.
- Оно будет доступно только внутри функции (кроме IE8-).
- Это имя предназначено для надёжного рекурсивного вызова функции, даже если она записана в другую переменную.
- Обратим внимание, что с Function Declaration так поступить нельзя. Такое «специальное» внутреннее имя функции задаётся только в синтаксисе Function Expression.

Условные операторы - if . . . else

```
if (condition); {
  Программный блок1
} [ else { программный блок2 }]
```

Оператор **if . . . else** - это условный оператор, который обеспечивает выполнение одного или нескольких операторов, в зависимости от того, удовлетворяются ли условия.

Часть **condition** оператора **if** является выражением, при истинности которого выполняются операторы языка в первом программном блоке.

Программный блок должен быть заключен в **фигурные скобки**, однако если используется только один оператор, можно скобки не ставить.

Необязательная часть **else** обеспечивает выполнение операторов второго блока, в случае, если условие **condition** оператора **if** является ложным.

Пример. Смена цвета фона в зависимости от системного времени: первая половина часа пусть будет синим, вторая - черным:

```
<html>
<head>
<script language ="JavaScript">
<!--
today = new date();
minutes = today.getMinutes();
if (minutes >=0 && minutes <= 30)
   document.write("<body text=white bgcolor=blue> Это написано
    белым на синем");
     else
       document.write("<body text=red bgcolor=black> Это написано
    красным на черном");
//-->
</script>
</body>
</html>
```

Оператор this

this[.property]

- Оператор this является не столько оператором, сколько внутренним **свойством языка JavaScript**.
- Значение this представляет собой текущий объект, имеющий стандартные **свойства**, такие как **name, length и value**.
- Оператор this **нельзя использовать вне области действия** функции или вызова функции. Когда аргумент property опущен, с помощью оператора this передается текущий объект. Однако при обращении к объекту, как правило, **нужно указать его определенное свойство**.
- Оператор this применяется для "**устранения неоднозначности**" объекта с помощью привязки его в область действия текущего объекта, а также для того, чтобы сделать программу более компактной.

Оператор with

```
with (objname); {
    statements
}
```

Оператор **with** делает объект, обозначенный как **objname**, текущим объектом для операторов в программном блоке **statements**. Удобство использования этого оператора заключается в том, что такая запись позволяет **сократить объем текста программы**.

Оператор with применяется к встроенному объекту Math языка JS

```
with (Math) {
    document.writeln(PI);
}
```

Такая запись позволяет избежать использования префикса Math при обращении к константам данного объекта.

Оператор with применительно к объекту document

```
with (parent.frames [1].document) {
    writeln("Пишем сюда текст");
    write("<hr>");
}
```

В этом случае оператор **with** избавляет нас от необходимости указывать перед методами **writeln()** и **write()** документ, к которому относятся вызовы этих методов.

Вывод.

- В данной лекции были рассмотрены и использованы объекты, методы, свойства и обработчики событий

Объектная модель языка. Объекты браузера

При создании HTML-документов и JavaScript-программ необходимо учитывать структуру объектов. Все объекты можно разделить на три группы:

- Объекты браузера
- Внутренние, или встроенные, объекты языка JavaScript
- Объекты, связанные с тегами языка HTML
- Объектами браузера являются зависимые от браузера объекты: window (окно), location (местоположение) и history (история). Внутренние объекты включают простые типы данных, такие как строки (string), математические константы (math), дата (date) и другие.
- Объекты, связанные с тегами HTML, соответствуют тегам, которые формируют текущий документ. Они включают такие элементы как гиперсвязи и формы.

Методы объектов

- С объектами связаны методы, которые позволяют управлять этими объектами, а также в некоторых случаях менять их содержимое. Кроме того в языке JavaScript имеется возможность создавать свои методы объектов. При использовании метода объекта, нужно перед именем метода указать имя объекта к которому он принадлежит.
- Например, правильным обращением к методу document является выражение

document.write(),

а просто выражение write() приведет к ошибке.

Свойства объектов языка JavaScript

- **Свойство** - это именованное значение, которое принадлежит объекту. Все стандартные объекты языка JS имеют свойства. Например, в прошлой главе мы использовали в одном из примеров свойство **bgColor** объекта **document**. Данное свойство соответствует атрибуту **bgColo**r тега **<body>** - цвет фона документа

.

- Для обращения к свойству необходимо указать имена объекта и свойства, разделив их точкой.
- Каждый объект имеет собственный набор свойств. Набор свойств нового объекта можно задать при определении объекта.
- Однако, некоторые свойства объектов существуют только для чтения, и вы не можете их менять. В таких случаях можно получить только значения этих свойств. Как показывает практика, такие свойства изменять обычно без надобности и проблем в связи с этим не возникает.

Для создания объекта есть два пути - создание напрямую экземпляра объекта и создание конструктора - отдельной функции, которая создает и инициализирует объект.

Для создания непосредственно экземпляра объекта мы можем воспользоваться двумя конструкциями:

```
var person = new Object();
```

или просто

```
var person = {}
```

Для создания объекта через конструктор мы должны описать функцию, которая будет создавать объект и присваивать значения его свойствам. Функция будет выглядеть следующим образом:

```
function Person(name, age, year) {
  this.name = name;
  this.age = age;
  this.year = year;
}
```

Внутри функции мы присваиваем переданные в функцию данные через выражение this - обращение к текущему экземпляру объекта. (Более подробно ключевое слово this мы рассмотрим в конце урока).

Создание экземпляра объекта с помощью нашего конструктора осуществляется следующим образом:

```
var employee1 = new Person("Ivan","25","2017");
var employee2 = new Person("Olga","21","2016");
var employee3 = new Person("Oleg","32","2010");
```

Важно отметить, что название конструктора принято писать с большой буквы, чтобы отметить что это конструктор и его необходимо вызывать с ключевым словом new. В противном случае this в теле конструктора будет указывать на что угодно, только не на созданный объект.

Для добавления нового свойства объекта также есть два варианта.

```
Person.name = "Ivan"
```

или

```
Person['name'] = "Ivan"
```

Также можно добавлять свойства сразу при создании объекта, указав список свойств в фигурных скобках:

```
var Person = {
    name : "Ivan",
    age : 25,
    hiredYear : 2017
}
```

Соответственно, для доступа к этому свойству тоже можно воспользоваться двумя вариантами:

```
console.log(Person.name);
```

или

```
console.log(Person['name'])
```

В случае если мы пытаемся обратиться к свойству, которого у объекта нет, то результат будет 'undefined', так называемое "неопределенное значение".

Для удаления свойства используется оператор delete:

```
delete Person.name;
```

Добавление метода в объект осуществляется с помощью следующего синтаксиса:

```
var person = {}                                    //Объявляем
объект person
person.sayAge = function(n) {                      //Объявляем
метод sayAge для объекта
                                                   //person
    console.log("Person is " + n + " years old");  //Тело
метода sayAge - вывод

//текста
};
```

В данном примере при вызове функции
person.sayAge(22);
Произойдет вывод в консоль текста "Person is 22 years old".

Добавление метода в объект - это фактически присвоение функции некоторому свойству объекта. В предыдущем примере мы присвоили функцию function(n) свойству sayAge объекта person .

Как правило нам необходимо чтобы метод не просто выполнял некоторые действия, а использовал какие-либо свойства объекта, в котором он хранится. Для того, чтобы получить доступ к свойствам объекта из метода используется ключевое слово this. Слово this никаким образом не связано с самим объектом, оно всего лишь обозначает объект, вызвавший эту функцию. В данном примере функция sayName будет выводить фразу "My name is Ivan":

```
var person = {
    name : "Ivan",
    age : 25,
    hiredYear : 2017
}

person.sayName = function() {
    console.log("My name is " + this.name);
}
```

Теперь мы можем более подробно рассмотреть оператор, осуществляющий перебор всех свойств и методов объекта, мельком упомянутый в предыдущем модуле - конструкцию for .. in.

Синтаксис команды выглядит следующим образом:

for (key in object), где key - название свойства, object - название объекта, а обращение к содержимому свойства осуществляется через выражение object[key].

Давайте рассмотрим пример, создадим объект и в нем метод, выводящий в консоль все свойства этого объекта:

```
var person = {
    name : "Ivan",
    age : 25,
    hiredYear : 2017
}

person.sayAll = function() {
  for (var i in this) {
          console.log(i + " is " + this[i]);
  }
}
```

Объекты браузеров

- HTML-объектами являются объекты, которые соответствуют тегам языка HTML:метки, гиперсвязи и элементы формы - **текстовые поля, кнопки, списки и др**.

- Объекты верхнего уровня, или объекты браузера, - это объекты, поддерживаемые в среде браузера: **window, location, history, document, navigator**.

Объекты, перечисленные в таблице, создаются автоматически при загрузке документа в браузер

Имя объекта	Описание
window	Объект верхнего уровня в иерархии объектов языка JavaScript. Фреймосодержащий документ также имеет объект window.
document	Содержит свойства, которые относятся к текущему HTML-документу, например имя каждой формы, цвета, используемые для отображения документа, и др. В языке JS большинству HTML-тегов соответствуют свойства объекта document.
location	Содержит свойства, описывающие местонахождение текущего документа, например адрес URL.
navigator	Содержит информацию о версии браузера. Свойства данного объекта обычно только для чтения. Например свойство: navigator.appname содержит строковое значение имени браузера.
history	Содержит информацию обо всех ресурсах, к которым пользователь обращался во время текущего сеанса работы с браузером.

Объект window

- Объект window обычно соответствует главному окну браузера и является объектом верхнего уровня в языке JavaScript, поскольку документы, собственно, и открываются в окне.
- В фреймосодержащих документах, объект window может не всегда соответствовать главному окну программы.
- Для обращения к конкретному окну следует использовать свойство frames объекта parent.
- Фреймы - это те же окна. Чтобы обратиться к ним в языке JavaScript, можно использовать массив frames.
- Например, выражение parent.frames[0] обращается к первому фрейму окна браузера. Предполагается, что такое окно существует, но при помощи метода window.open() можно открывать и другие окна и обращаться к ним посредством свойств объекта window.

Для обращения к методам и свойствам объекта window используют следующие варианты записи:

- window.propertyName
- window.methodName (parameters)
- self.propertyName
- self.methodName (parameters)
- top.propertyName
- top.methodName (parameters)
- parent.propertyName
- parent.methodName (parameters)
- windowVar.propertyName
- windowVar.methodName (parameters)
- propertyName
- methodName (parameters)

Свойства

Объект window имеет свойства:

- defaultStatus - текстовое сообщение, которое по умолчанию выводится в строке состояния (status bar) окна браузера.
- frames - массив фреймов во фреймосодержащем документе.
- length - количество фреймов во фреймосодержащем документе.
- name - заголовок окна, который задается с помощью аргумента windowName метода open().
- parent - синоним, используемый для обращения к родительскому окну.
- self - синоним, используемый для обращения к текущему окну.
- status - текст временного сообщения в строке состояния окна браузера.
- top - синоним, используемый для обращения к главному окну браузера.
- window - синоним, используемый для обращения к текущему окну.

Методы

- **Метод alert()** применяется для того, чтобы вывести на экран текстовое сообщение.
- Для открытия окна используется метод open(), а для закрытия - **метод close().**
- С помощью **метода confirm()** происходит вывод на экран окна сообщения с кнопками Yes и No, и возвращает булевое значение true или false, в зависимости от нажатой кнопки.
- Посредством **метода prompt()** на экран выводится диалоговое окно с полем ввода.
- Метод **setTimeout()** устанавливает в текущем окне обработку событий, связанных с таймером.
- Метод **clearTimeout()** отменяет обработку таких событий.

Обработчики событий

- Объект **window** не обрабатывает события до тех пор, пока в окно не загружен документ.
- Однако можно обрабатывать события, связанные с загрузкой и выгрузкой документов.
- Обработчики таких событий задаются как значения атрибутов **onLoad и onUnload**, определяемых в теге **<body>.**
- Эти же атрибуты могут быть определены в тегах **<frameset>** фреймосодержащих документов.

пример:

- Загрузка страницы

http://my.site.ru в окно размером в 640x480 пикселов:

```
myWin = open ("http://my.site.ru",
"myWin",
"width=640, height=480");
```

Закрыть это окно можно из любого другого окна используя:

```
myWin.close();
```

Объект document

- Объект document соответствует всему гипертекстовому документ, вернее, той его части, которая заключена в контейнер <body> . . . </body>. Документы отображаются в окнах браузера, поэтому каждый из них связан с определенным окном. Все HTML-объекты являются свойствами объекта document, поэтому они находятся в самом документе. Например, в языке JS к первой форме документа можно обратиться, используя выражение:

```
document.forms[0]
```

в то время как к первой форме во втором фрейме следует обращаться выражением:

```
parent.frames[1].document.forms[0]
```

- Объект document удобно использовать для динамического создания HTML-документов.
- Для этого применяется HTML-контейнер

  <body> . . . </body>.
- Хотя в этом контейнере можно установить множество различных свойств документа, все же имеются такие свойства, значения которых нельзя установить с помощью этих тегов. Синтаксис тега я не буду приводить, - его можно найти в спецификации HTML. Мы же, будем считать, что синтаксис HTML знаем.

Свойства объекта document

- document.propertyName
- Объект document имеет достаточно много свойств, каждое из которых соответствует определенному HTML-тегу в текущем документе:
- alinkColor- соответствует атрибуту alink тега <body>;
- anchors- массив, который соответствует всем меткам в документе;
- bgColor- соответствует атрибуту bgColor (цвет фона) тега <body>;
- cookie- представляет собой фрагмент информации, записанный на локальный диск ("ключик");
- fgColor- соответствует атрибуту fgColor (цвет текста) тега <body>;

- fgColor- соответствует атрибуту fgColor (цвет текста) тега <body>;
- forms- массив, содержащий все теги <form> в текущем документе;
- images- масив изображений, ссылки на которые заданы в текущем документе;
- lastModified- дата последнего изменения текущего документа;
- linkColor- соответствует атрибуту linkColor (цвет гиперсвязи по умолчанию);
- links- масив, содержащий все гипервязи в текущем документе;
- location- соответствует адресу URL текущего документа;
- referrer- соответсствует адреу URL документа, из которого пользователь перешел к текущему документу;
- title- соответствует содержимому контейнера <title> . . . </title>;
- vlinkColor- соответствует атрибуту vlinkColor (цвет <FONT COLOR="#800080">посещенной связи) тега <body>.

Методы объекта document

document.methodName (parameters)

Метод **clear()** предназначен для очистки текущего документа.

Лучше использовать для очистки методы **open() и close()**.

Для записи информации в браузер применяют методы **write() и writeln()**. Поскольку эти методы записывают текст в браузер в HTML-формате, вы можете создавать любой HTML-документ динамически, включая готовые приложения на языке JavaScript.

Если в окно загружен документ, то запись данных поверх него может привести к сбою. Поэтому в окно следует записывать поток данных, для чего с помощью метода **document.open()** нужно открыть документ, а затем, вызвав необходимое количество раз метод **document.wtite()**, записать данные в документ.

В заключение, чтобы послать данные в браузер, следует вызвать метод **document.close()**.

Обработчики событий

В тегах <body> и <frame> можно использовать обработчики событий, связанных загрузкой и выгрузкой документа, onLoad и onUnload. Примеры использования событий будем разбирать позже.

Для записи текста в HTML-формате в браузер иногда применяют функцию document.writeln().

Например, можно динамически создавать теги изображений, выводя изображения на экран посредством следующего:

```
document.open();
document.writeln("<img
  sr='myimage.gif'>");
document.close();
```

- С помощью JavaScript программ, а в частности при помощи объекта document, можно создавать законченные HTML-документы и другие JavaScript программы. Например:
-

```
document.open();
document.writeln("<script language='JavaSript'>"+
"alert('Hello World!')"+
"</script>");
document.close();
```

- Заметьте, что в приведенных примерах неколько строк объединяются при помощи операции сложения +. Этот способ удобно применять, когда строки текста программы слишком длинны, чтобы поместиться в редактируемом окне, или когда сложные строки необходимо разбить на несколько простых.

Объект location

- Данный объект сохраняет местоположение текущего документа в виде адреса URL этого документа.
- При управлении объектом location существует возможность изменять адрес URL документа.
- Объект location связан с текущим объектом window - окном, в которое загружен документ.
- Документы не содержат информации об адресах URL.
- Эти адреса являются свойством объектов window.

Объект location

[windowVar.]location.prepertyName

где windowVar - необязательная переменная, задающая конкретное окно, к которому хотите обратиться. Эта переменная также позволяет обращаться к фрейму во фреймосодержащем документе при помощи свойства parent - синонима, используемого при обращении к объекту window верхнего уровня, если окон несколько. Объект location является свойством объекта window. Если вы обращаетесь к объекту location без указания имени окна, то подразумевается свойство текущего окна.

- Свойство location объекта window легко перепутать со свойством location объекта document. Значение свойства document.location изменить нельзя, а значение свойства location окна - можно, например при помощи выражения window.location.property. Значение document.location присваивается объекту window.location при первоначальной загрузке документа, потому, что документы всегда загружаются в окна.

Свойства

Объект location имеет следующие свойства:

- hash - имя метки в адресе URL (если задано);
- host - часть hostname:port адреса URL текущего документа;
- hostname - имя хоста и домена (или цифровой IP-адрес) в адресе URL текущего документа;
- href - полный адрес URL текущего документа;
- pathname - часть адреса URL, описывающая каталог, в котором находится документ;
- port - номер порта, который использует сервер;
- protocol - префикс адреса URL, описывающий протокол обмена, (например, http:);
- target - соответствует атрибуту target в теге <href>.

Методы и обработчики событий

- Для объекта location методы, не определены, также не связан с какими-либо обработчиками событий.

Примеры

- Чтобы присвоить свойству location текущего окна в качестве значения новый адрес URL, используйте такой вид:

```
self.location="http://wdstudio.al.ru";
```

- который в данном случае загружает в текущее окно Web-страницу. Вы можете опустить объект self, поскольку он является ссылкой на текущее окно.
- Чтобы загрузить ресурс в фреймосодержащий документ, можно записать так:

```
parent.frames[0].location = "http://my.site.ru";
```

- где parent.frames[0] соответствует первому фрейму в текущем документе.

Объект history

- Объект history содержит список адресов URL, посещенных в этом сеансе. Объект history связан с текущим документом. Несколько методов этого объекта позволяют загружать в браузер различные ресурсы и обеспечивают навигацию по посещенным ресурсам.
-
- Синтаксис:

history.propertyName

history.methodName (parameters)

Свойства: Значением свойства length является количество элементов в списке объекта history.

Объект history

- Методы
- Метод **back()** позволяет загружать в браузер предыдущий ресурс, в то время как метод **forward()** обеспечивает обращение к следующему ресурсу в списке.
- С помощью метода **go()** можно обратиться к ресурсу с определенным номером в списке объекта **history**.
- Обработчики событий для объектов history не определены.

Примеры использования объекта history:

- Чтобы посмотреть предыдущий загруженный документ, воспользуйтесь оператором:

history.go(-1);

или

history.back();

- Для обращения к истории конкретного окна или фрейма применяют объект parent:

parent.frames[0].history.forward();

загружает в первый фрейм предыдущий документ.

- А если открыто несколько окон браузера можно использовать вид:
  - window1.frames[0].history.forward();

здесь в первый фрейм окна window1 будет загружен следующий документ из списка объекта history

Объект navigator

- Объект navigator содержит информацию об используемой в настоящее время версии браузера. Этот объект применяется для получения информации о версиях.
- Синтаксис:

navigator.propertyName

- Методы и события, как и не трудно догадаться не определены для этого объекта. Да и свойства только для чтения, так как ресурс с информацией о версии недоступен для редактирования.

Свойства

- appCodeName - кодовое имя браузера;
- appName - название браузера;
- appVersion - информация о версии браузера;
- userAgent - кодовое имя и версия браузера;
- plugins - массив подключаемых модулей (похоже только для Netscape);
- mimeTypes - поддерживаемый массив типов MIME.

Выводы

- Здесь я попыталась ввести понятия объектов и связанных с ними методов, свойств и обработчиков событий.
- Также описала объекты браузера. В следующих лекциях будут описаны остальные объекты языка JavaScript.

Внутренние объекты

В этой лекции мы рассмотрим внутренние объекты языка JavaScript. В предыдущей части рассматривались объекты браузера.

Внутренние объекты не относятся к браузеру или загруженному в настоящее время HTML-документу. Эти объекты могут создаваться и обрабатываться в любой JavaScript-программе.

- Они включают в себя простые типы, такие как строки, а также более сложные объекты, в частности даты.
- Имя объекта Описание
- Array Массив. Не поддерживается в браузерах старых версий
- Date Дата и время
- Math Поддержка математических функций
- Object Обобщенный объект. Не поддерживается в старых версиях IE - до 4, NN - до 3.
- String Текстовая строка. Не поддерживается в старых версиях

Объект array

- Array - это многомерное упорядоченное множество объектов, обращение к объектам ведется при помощи целочисленного индекса. Примерами объектов-массивов в браузере служат гиперсвязи, метки, формы, фреймы.

Массив можно создать одним из следующих способов:

- используя определенную пользователем функцию для присвоения объекту многих значений;
- используя конструктор Array();
- используя конструктор Object().
- Объекты-массивы не имеют ни методов, ни свойств.

Объект Date

- Объект содержит информацию о дате и времени. Этот объект имеет множество методов, предназначенных для получения такой информации. Кроме того объекты Date можно создавать и изменять, например путем сложения или вычитания значений дат получать новую дату.

Для создания объекта Date применяется синтаксис:

dateObj = new Date(parameters)

где dateObj - переменная, в которую будет записан новый объект Date.

Объект Date

Аргумент parameters может принимать следующие значения:

- пустой параметр, например date() в данном случае дата и время - системные.
- строку, представляющую дату и время в виде: "месяц, день, год, время", например "March 1, 2000, 17:00:00" Время представлено в 24-часовом формате;
- значения года, месяца, дня, часа, минут, секунд. Например, строка "00,4,1,12,30,0" означает 1 апреля 2000 года, 12:30.
- целочисленные значения только для года, месяца и дня, например "00,5,1" означает 1 мая 2000 года, сразу после полночи, так, как значения времени равны нулю.

Как уже говорилось ранее данный объект имеет множество методов, свойств объект Date не имеет.

Методы.

Метод **Описание метода**

- getDate() Возвращает день месяца из объекта в пределах от 1 до 31
- getDay() Возвращает день недели из объекта: 0 - вс, 1 - пн, 2 - вт, 3 - ср, 4 - чт, 5 - пт, 6 - сб.
- getHours() Возвращает время из объекта в пределах от 0 до 23
- getMinutes() Возвращает значение минут из объекта в пределах от 0 до 59
- getMonth() Возвращает значение месяца из объекта в пределах от 0 до 11
- getSeconds() Возвращает значение секунд из объекта в пределах от 0 до 59
- getTime() Возвращает количество миллисекунд, прошедшее с 00:00:00 1 января 1970 года.
- getTimeZoneoffset() Возвращает значение, соответствующее разности во времени (в минутах)
- getYear() Возвращает значение года из объекта

Методы.

Метод Описание метода

- Date.parse(arg) Возвращает количество миллисекунд, прошедшее с 00:00:00 1 января 1970 года. Arg - строковый аргумент.
- setDate(day) С помощью данного метода устанавливается день месяца в объекте от 1 до 31
- setHours(hours) С помощью данного метода устанавливается часы в объекте от 0 до 23
- setMinutes(minutes) С помощью данного метода устанавливаются минуты в объекте от 0 до 59
- setMonth(month) С помощью данного метода устанавливается месяц в объекте от 1 до 12
- setSeconds(seconds) С помощью данного метода устанавливаются секунды в объекте от 0 до 59

Методы.

Метод **Описание метода**

- setTime(timestring)С помощью данного метода устанавливается значение времени в объекте.
- setYear(year) С помощью данного метода устанавливается год в объекте year должно быть больше 1900.
- toGMTString() Преобразует дату в строковый объект в формате GMT.
- toString() Преобразует содержимое объекта Date в строковый объект.
- toLocaleString() Преобразует содержимое объекта Date в строку в соответствии с местным временем.
- Date.UTC(year, month, day [,hours][,mins][,secs]) Возвращает количество миллисекунд в объекте Date, прошедших с с 00:00:00 1 января 1970 года по среднему гринвичскому времени.

Разберем пару примеров:

- В данном примере приведен HTML-документ, в заголовке которого выводится текущие дата и время.

```
<html>
<head>
<script language "JavaScript">
<--
function showh() {
  var theDate = new Date();
  document.writeln("<table cellpadding=5 width=100%
   border=0>" +
      "<tr><td width=95% bgcolor=gray align=left>" +
      "<font color=white>Date: " + theDate +
      "</font></td></tr></table><p>");}
showh();
//-->
</script>
</head>
</html>
```

Разберем еще один пример. Подобный мы уже разбирали, когда рассматривали условные операторы, просто вспомним его и немного изменим: пусть меняются графические бэкграунды в зависимости от времени суток.

```
<html>
<script language "JavaScript">
<--
theTime = new Date();
theHour = theTime.getHours();
if (18 > theHour)
  document.writeln("<body background='day.jpg'
   text='Black'>");
else
  document.writeln("<body background='night.jpg'
   text='White'>");
//-->
</script>
</body>
</html>
```

Вероятно, вы успели заметить, что тег <body> создается в JavaScript-программе, а закрывается уже в статическом тексте HTML. Это вполне допустимо, так, как все теги расположены в правильном порядке. В данном примере предполагается, что файлы рисунков находятся в том же каталоге. Вы можете здесь задать полный адрес URL.

Объект Math

Объект Math является встроенным объектом языка JavaScript и содержит свойства и методы, используемые для выполнения математических операций. Объект Math включает также некоторые широко применяемые математические константы.

Синтаксис:

```
Math.propertyName
Math.methodName(parameters)
```

Свойства

Свойствами объекта Math являются математические константы:

- E Константа Эйлера. Приближенное значение 2.718 . . .
- LN2Значение натурального логарифма числа два. Приближенное значение 0.693 . . .
- LN10 Значение натурального логарифма числа десять. Приближенное значение 2.302 . . .
- LOG2_E Логарифм e по основанию 2 (не вижу смысла в этой константе - это же корень из двух.) Приближенное значение 1.442 . . .)
- LOG10_E Десятичный логарифм e. Приближенное значение 0.434 . . .
- PI Число ПИ. Приближенное значение 3.1415 . . .
- SQRT2Корень из двух, (ыгы, это все равно, как еще и натуральный логарифм 2*e в степени 1/2)

Методы

Методы объекта Math представляют собой математические функции.

- abs() Возвращает абсолютное значение аргумента.
- acos() Возвращает арккосинус аргумента
- asin() Возвращает арксинус аргумента
- atan() Возвращает арктангенс аргумента
- ceil() Возвращает большее целое число аргумента, округление в большую сторону. Math.ceil(3.14) вернет 4
- cos() Возвращает косинус аргумента
- exp() Возвращает экспоненту аргумента
- floor() Возвращает наибольшее целое число аргумента, отбрасывает десятичную часть

Методы

- max() Возвращает больший из 2-х числовых аргументов. Math.max(3,5) вернет число 5
- min() Возвращает меньший из 2-х числовых аргументов.
- pow() Возвращает результат возведения в степень перврго аргумента вторым. Math.pow(5,3) вернет 125
- random() Возвращает псевдослучайное число между нулем и единицей.
- round() Округление аргумента до ближайшего целого числа.
- sin() Возвращает синус аргумента
- sqrt() Возвращает квадратный корень аргумента
- tan() Возвращает тангенс аргумента

Обработчиков событий нет для внутренних объектов.
Синтаксис очень прост, вызывается метод как любая функция, но это все же метод и не забывайте указывать префикс Math перед методом:

```
var mpi = Math.Pi
```

В данном случае переменной mpi присвоится значение Пи.
Или, например,

```
var myvar = Math.sin(Math.Pi/4)
```

Строковые объекты

- Строка (string) в языке JavaScript представляется в виде пследовательности символов, заключенных в двойные или одинарные кавычки. Для управления строковыми объектами используется синтаксис:

stringName.propertyName

stringName.methodName(parameters)

Здесь stringName - имя объекта String. Строки можно создавать тремя способами:

- 1. Создать строковую переменную при помощи оператора var и присвоить ей строковое значение;
- 2. Присвоить значение строковой переменной только посредством оператора присваивания (=);
- 3. Использовать конструктор String().

Свойства

- Значением свойства length является длина строки.
- Например, выражение

  "Script".length

- вернет значение 6, поскольку строка "Script" содержит 6 символов.

Методы

- Вызов метода осуществляется обычно: **"Строка или строковая переменная".метод()**,
- в данном случае метод без параметров, имеются методы и с параметрами. Заметьте, строка или строковая переменная, к которой применяется метод - объект, и никак не аргумент!

Метод Описание метода

- **big()** Аналогично тегам HTML <big> . . .</big>. позволяет отобразить более крупным шрифтом.
- **blink()** Заставляет строку мигать. (Этим почти никто не пользуется).
- **bold()** Название говорит за себя - делает символы жирными.
- **charAt(arg)** Возвращает символ, находящийся в заданной позиции строки. Пример: vpos = "Sc<FONT COLOR="#FF0000">ript".charAt(3); переменной vpos будет присвоено значение "r".
- **fixed()** Анологично <tt> . . .</tt> вывод строки фиксированного размера.
- **fontcolor(arg)** Анологично . . .. Аргумент метода может быть как триплетом RGB, так и зарезервированным словом.
- **fontsize(arg)** Позволяет изменять размер шрифта. Аргумент в условных единицах. Анологично . . ..
- Также можно использовать вид **+size** или **-size** для увеличения или уменьшения шрифта на size единиц, например: **"строка".fontsize(+1)**.

- **indexOf(arg1[,arg2])** Возвращает позицию в строке, где впервые встречается символ - arg1, неябязательный числовой аргумент arg2 указывает начальную позицию для поиска.
- **italics()** Аналогично тегам HTML . . .. позволяет отобразить италиком.
- **lastIndexOf(arg1[,arg2])** Возвращает либо номер позиции в строке, где в последний раз встретился символ - arg1, либо -1, если символ не найден. Arg2 задает начальную позицию для поиска.
- **link()** Аналогично тегам HTML <a href> . . .. позволяет преобразовать строку в гиперсвязь.
- **small()** Аналогично тегам HTML <small> . . .</small>. позволяет отображать строку мелким шрифтом.
- **strike()** Аналогично тегам HTML <strike> . . .</strike>. позволяет отображать строку зачеркнутой.

- **sub()** Аналогично тегам HTML _{. . .}. позволяет отображать строку нижним индексом.
- **substring(arg1,arg2)** Позволяет извлеч подстроку длинной arg2, начиная с позиции arg1
- **sup()** Аналогично тегам HTML ^{. . .}. позволяет отображать строку верхним индексом.
- **toLowerCase()** Преобразует символы строкового объекта в строчные
- **toUpperCase()** Преобразует символы строкового объекта в прописные

Вот, пожалуй, и весь список методов объекта String. Примеры их использования будут приводиться по ходу рассмотрения других объектов. К строковым методам, как видно из таблицы относятся методы-функции операций над строками и в то же время как методы форматирования.

Объекты, соответствующие тегам HTML - 1

- Многие объекты языка JavaScript соответствуют тегам, формирующим HTML-документы. Каждый такой объект состоит во внутренней иерархии, поскольку все они имеют общий родительский объект - браузер, который представлен объектом window.
- Некоторые объекты языка JavaScript имеют потомков. В частности, гиперсвязь является объектом, наследованным из объекта document. В языке JS наследованные объекты называются также свойствами. Например, множество гиперсвязей является свойством объекта document, а links - именем этого свойства. Таким образом, трудно провести четкую границу между объектами и свойствами. **Гиперсвязь, являясь объектом, в то же время представляет собой свойство links объекта document.**

Рассмотрим пример. Напишем простенькую программку и посмотрим, как будут создаваться объекты HTML. Тоесть, при загрузке HTML-документа в браузер соответственно будут созданы HTML-объекты на JavaScript. Теги HTML собственно служат исключительно для удобства написания документа:

```
<html>
<head>
<title>Пример программы</title>
</head>
<body bgcolor="White">
<form>
<input type="checkbox" checked
  name="chck1">Item 1
</form>
</body>
</html>
```

Посмотрим эквивалентную запись на JavaScript

```
document.title="Пример программы"
document.bgColor="White"
document.forms[0].chck1.defaultChecked=true
```

Как видно из примера, тегу <title> соответствует свойство document.title, а цвету фона документа, установленному в теге <body>, - свойство document.bgColor.
Переключателю checkbox с именем chck1, определенному в форме, соответствует выражение document.forms[0].chck1.
Свойство defaultChecked принадлежит объекту check-box и может принимать значения true или false в зависимости от того, указан ли в теге <input> атрибут checked. Когда этот атрибут задан, переключатель отображается на экране как включенный по умолчанию.

- Поскольку документ может включать несколько таких объектоов, как гиперсвязи, формы и другие объекты, многие из них являются массивами. Для **обращения к определенному элементу массива** нужно указать его индекс. Например, **forms[0]** - первая форма текущего документа. Ко второй форме можно обратиться соответственно **forms[1]**. Заметьте, что индексы массивов в JS программах всегда начинаются с **нуля**.

- В нашем примере объект верхнего уровня - window, потому, что любой документ загружается в окно. Например, выражения document.forms[0] и window.document.forms[0] обращаются к одному и тому же объекту, а именно к первой форме текущего документа. Однако если необходимо обратиться к форме в другом окне (фрейме), следует использовать выражение вида

  parent.frames[n].document.forms[n]

  где n является индексом нужного элемента массива.

Имя объектаКраткое описание

- **anchor (anchors[])** Множество тегов <a name> в текущем документе
- **button** Кнопка, создаваемая при помощи тега <input type=button>
- **checkbox** Контрольный переключатель, создаваемый при помощи тега <input type=checkbox>
- **elements[]** Все элементы тега <form>
- **form (forms[])** Множество объектов тега <form> языка HTML
- **frame (frames[])** Фреймосодержащий документ
- **hidden** Скрытое текстовое поле, создаваемое при помощи тега <input type=hidden>
- **images (images[])** Множество изображений (тегов <img>) в текущем документе
- **link (links[])** Множество гиперсвязей в текущем документе

Имя объектаКраткое описание

- **navigator** Объект, содержащий информацию о браузере, загрузившем документ
- **password** Поле ввода пароля, создаваемое при помощи тега <input type=password>
- **radio** Селекторная кнопка (radio button), создаваемая при помощи тега <input type=radio>
- **reset** Кнопка перегрузки, создаваемая при помощи тега <input type=reset>
- **select (options[])** Элементы <option> объекта <select>
- **submit** Кнопка передачи данных, создаваемая при помощи тега <input type=submit>
- **text** Поле ввода, сооздаваемое при помощи тега <input type=text>
- **textarea** Поле текста, создаваемое при помощи тега <textarea>

Объекты, которым соответствуют массивы, являются многомерными объектами.

В некоторых HTML-тегах можно определить несколько элементов, например множество элементов списка в теге <select>.

Рассмотрим тег <select>, содержащий два элемента:

```
<form>
<select name="primer">
<option>Опция1
<option>Опция2
</select>
</form>
```

- Тег <select> сам по себе является объектом, однако для обращения к отдельным элементам этого объекта (тегам <option>) используется массив option.
- Данный массив представляет собой множество значений, которые соответствует тегам <option>, содержащимся в теге <select>. В нашем примере создается два объекта: первый - объект select в целом (к нему обращаются, чтобы выяснить, сколько элементов он фактически соодержит), второй - массив options (он позволяет обращаться к отдельным элементам, содержащимся в первом объекте).
- Таким образом, некоторые объекты могут использовать объекты-массивы для обращения к содержащимся в них элементам.

- Однако это не является правилом, все зависит от структуры рассматриваемых объектов и тех объектов, из которых они унаследованы. Например, HTML-тегам <a name> . . . соответствует объект anchor, являющийся элементом массива anchors, и в то же время эти теги встречаются сами по себе, а не в других тегах.
- Просто родительским объектом (parents) для объекта anchors является объект document, а в документе может быть определено множество меток.
- Окна тоже могут содержать множество документов, связанных с ними через фреймы.

Объект anchor и массив anchors

- Anchor - это элемент текста, который является объектом назначения для тега гиперсвязи <a href>, а также свойством объекта document. Тегу <a name> в языке JavaScript соответствует объект anchor, а всем тегам <a name>, имеющимся в документе, - массив anchors.
- Являясь объектами назначения для гиперсвязей <a name>, метки в основном используются для индексирования содержимого гипертекстовых документов, обеспечивая быстрое перемещение к определенной части документа при щелчке мыши на гиперсвязи, которая обращается к данной метке. Тег <a>, задающий метки и гиперсвязи, имеет синтаксис:

```
<a [href=location]
    [name="anchorName"]
    [target="windowName"] >
    anchorText
    </a>
```

- Как вы успели заметить, обычная схема построения гиперсвязей. Значение location задает имя метки.
- Когда значение определено, данный тег задает гиперсвязь.
- Location может также включать и URL, например: **href="http://wdstudio.al.ru/index.htm#netscape".**
- Заметьте, что перед и после знака # пробелы не допустимы.
- Атрибут **name="anchorName"** определяет имя метки, которая будет объектом назначения для гипертекстовой связи в текущем HTML-документе: .
- В данном случае netscape - имя метки, которая будет объектом назначения для гипертекстовой связи.
- Атрибут **target="windowName"** - имя объекта-окна или его синонима: self, top и др., в котрое загружается документ, запрашиваемый при активизации гиперсвязи.

- Значение **anchorText з**адает текст, который будет отображаться на экране в том месте, где находится метка, и является необязательным.
- Например: ** **.
- А вот с атрибутом **href -** текст в большинстве случаев должен быть виден на экране, иначе как активизировать гиперсвязь.
- Атрибут **target** также может существовать только с атрибутом **href.**

Массив anchors

- Посредством массива anchors программа на языке JavaScript может обращаться к метке текущего гипертекстового документа. Каждому тегу <a name> текущего документа соответствует элемент массива anchors. Для того чтобы программа выполнялась правильно, в соответствующих атрибутах name должны быть заданы имена всех меток. Если документ содержит именованную метку, определенную HTML-тегом

  < a name="s1">Selection1

- то этой метке в JS-программе соответствует объект **document.anchors[0]**.

Массив anchors

- Чтобы перейти к этой метке посредством гиперсвязи, пользователь должен щелкнуть мышью на тексте, определенном в контейнере . . . . К массиву anchors можно обращаться при помощи следующих операторов:

document.anchors[i]

document.anchors.length

- где i - индекс запрашиваемой метки. Свойство length позволяет определить количество меток в документе, хотя элементы, соответствующие отдельным меткам, будут содержать значение null. Это значит, что нельзя обращаться к именам отдельных меток через элементы массива.

Свойства

- Массив anchors имеет только одно свойство **length**, которое возвращает значение, соответствующее количеству меток в документе.
- Массив anchors является структурой только для чтения.
- Методов и обработчиков событий объекты anchors не имеют.

Объект button

- Кнопка - это область окна, которая реагирует на щелчки мыши и может активизировать оператор или функцию языка JavaScript при помощи атрибута события onClick.
- Кнопки являются свойствами объекта form и должны быть заключены в теги *<form> . . . </form> языка HTML*.

Синтаксис:

```
<input type="button"
name="buttonName"
value="buttonText"
[onClick="handlerText"]>
```

Атрибут name задает имя кнопки и в языке JS ему соответствует свойство name нового объекта buttun. Атрибут value определяет надпись на кнопке, которой соответствует свойство value. К свойствам и методам объекта button можно обратиться одним из способов:

-- buttonName.propertyName

-- buttonName.methodName (parameters)

-- formName.elements[i].propertyName

-- formName.elements[i].methodName (parameters)

- Здесь **buttonName** - значение атрибута name, а formName - либо значение атрибута name объекта form, либо элемент массива forms. Переменная i является индексом, используемым для обращения к отдельному элементу массива, в данном случае к элементу button.

Свойства

- Свойства **name** и **value** объекта button соответствует атрибутам name и value HTML-тега <input>.
- Обратившись к значениям этих свойств, можно вывести полный список кнопок, имеющихся в текущем документе.
- Свойство **type** объекта button всегда имеет значение "button".

Методы и обработчики событий

- Объект button имеет метод **click()** - о нем будем говорить позже.
- Обработчик событий **onClick** позволяет выполнить оператор или вызвать функцию языка JavaScript при щелчке мыши на кнопке, которой соответствует в программе определенный объект button.

Приведем простой пример, например, выведем текущую дату и время посредством нажатия кнопки. Будем использовать событие onClick для вызова метода alert() и конструктора Date()
Пример схематичный, объект должен быть определен

```
<form>
<input type="button"
value="Date and Time"
onClick='alert(Date())'>
</form>
```

Объект checkbox

- Контрольный переключатель - это кнопка(флажок), которую можно установить в одно из двух состояний: включено или выключено.
- Объекты checkbox являются свойствами объекта form и должны быть помещены в теги **<form> . . . </form>.**

Простой контрольный переключатель:

- Checkbox1
- Checkbox2
- Checkbox3

Синтаксис:

```
<input name="checkboxName"
type="checkbox"
value="checkboxValue"
[checked]
[onClick="handlerText"]>textToDisplay
```

- где атрибут name является именем объекта checkbox.
- Ему соответствует свойство *name* объекта языка JavaScript. Атрибут *value* определяет значение, которое передается серверу при пересылки значений элементов формы, если контрольный переключатель включен.
- Необязательный атрибут *checked* указывает, что контрольный переключатель должен быть включен по умолчанию.
- Если этот атрибут задан, свойство *defaultChecked* имеет значение true.
- При помощи свойства *checked* можно определить, включен ли контрольный переключатель.
- Текст, отображаемый рядом с контрольным переключателем, задается строкой *textToDisplay*.

К объекту checkbox можно обращаться одним из способов:

checkboxName.propertyName
checkboxName.methodName (parameters)
formName.elements[i].propertyName
formName.elements[i].methodName (parameters)

- где checkboxName - значение атрибута name объекта checkbox, а formName - имя объекта form или формы, которой принадлежит данный контрольный переключатель. Другими словами, к форме можно обращаться как к элементу массива forms, например forms[0] - для обращения к первой форме документа, либо по имени объекта form, если оно определено в атрибуте name HTML-тега <form>.

- К любому элементу формы можно обратиться так же, как к элементу массива elements, который является свойством объекта form.
- В этом случае для обращения к определенному контрольному переключателю следует использовать его порядковый номер (i) в массиве всех элементов формы.

Свойства

- Если контрольный переключатель включен, свойство checked имеет значение true.
- Когда в теге <input> используется атрибут checked, например <input checked type=checkbox>, свойсту defaultChecked также присваивается значение true.
- Свойство name соответствует атрибуту name тега <input name= . . . type=checkbox>,
- а свойство value - атрибуту value тега <input>. (оно и правильно: ключевые слова и должны соответствовать чтобы путаницы не было). Свойство type объекта checkbox всегда содержит значение "checkbox".

Методы и обработчики событий

- Метод ***Click()*** может использоваться с объектом ***checkbox***, мне не приходилось его использовать, но есть много замечаний в адрес этого метода, - в некоторых браузерах он должным образом не работает. Но тем не менее он имеется.
- Для объекта ***checkbox*** предусмотрен только один обработчик - ***onClick()***.

Массив elements

- Массив elements содержит все элементы HTML-формы - контрольные переключатели (checkbox), селекторные кнопки (radio-button), текстовые объекты (text) и другие, - в том порядке, в котором они определены в форме. Этот массив можно использовать для доступа к элементам формы в JS-программе по их порядковому номеру, не используя свойства name этих элементов. Массив elements, в свою очередь, является свойством объекта forms, поэтому при обращении к нему следует указывать имя формы, к элементу которой вы хотите обратиться:

  formName.elements[i]

  formName.elements[i].length

- Здесь formName может быть либо именем объекта form, определенным при помощи атрибута name в теге <form>, либо элементом массива forms, например forms[i], где i - переменная, которая индексирует элементы массива. Значением свойства length является количество элементов, содержащихся в форме. Массив elemments включает данные только для чтения, т.е. динамически записать в этот объект какие-либо значения невозможно.

Свойства

- Объект **elements** имеет только одно свойство, **length**, значением которого является количество элементов объекта form.

  document.forms[0].elements.length

- возвратит значение, соответствующее количеству элементов в первой форме текущего документа.

Пример

Создадим пару элементов, например поля ввода для имени и адреса:

Имя: Адрес:

Нажав на эту кнопку, можно увидеть элементы формы, назовем ее "Форма для примера". Третьим элементом будет кнопка, вызывающая функцию на JavaScript. Она также находится в данной форме.

Внимание: не корректно работает в Internet Explorer-е. Дело в том, что в этом браузере элементы формы хранятся не в виде строки. В NN должно быть нормально. IE 3.01 может даже вызвать ошибку. IE 4 и выше ошибки не выдает.

Теперь рассмотрим текст этой программы:

```
<html>
<head>
<script language="JavaScript">
        function showElem(f) {
var formEl = " ";
for (var n=0; n < f.elements.length; n++) {
formEl += n + ":" + f.elements [n] +"\n";
}
alert("Элементы в форме '" + f.name + "' :\n\n" + formEl );
}
</script>
</head>
```

- Здесь функция перебирает все элементы массива elements заданной формы, в данном примере их три, формирует строку formEl, содержащую информацию об элементах данного массива. IE покажет здесь в виде "n:[object]" то есть этот браузер не содержит в массиве elements строки с информацией об объекте формы. Созданная строка (для удобства читаемости разделена "переводом строки \n") выводится в окне предупреждения с помощью метода alert().
- Функция showEl() вызывается с аргументом this.form, который обращается к текущей форме. Если оператор this опустить, то из функции showEl() к форме придется обращаться с помощью выражения document.forms[n], - это не очень удобно, так как мы обращаемся из текущей формы.

Объект form и массив forms

- **Форма** - это область гипертекстового документа, которая создается при помощи контейнера **<form> . . . </form>** и содержит элементы, позволяющие пользователю вводить информацию.
- Многие HTML-теги, например теги, определяющие поля ввода (text field), области текста (textarea), контрольные переключатели (checkbox), селекторные кнопки (radio button) и списки (selection list), располагаются только в контейнере <form> . . . </form>.
- Всем перечисленным элементам в языке JavaScript соответствуют отдельные объекты.
- Программы на языке JS могут обрабатывать формы непосредственно, получая значения, содержащиеся в необходимых элементах (например для проверки ввода обязательных данных). Кроме того, данные из формы обычно передаются для обработки на удаленный Web-сервер.

Синтаксис:

```
<form name="formName"
target="windowname"
action="serverURL"
method="get" | "post"
enctype="encodingType"
[onSubmit="handlerText"]>
</form>
```

- Атрибут **name** - строка, определяющая имя формы. Атрибут **target** задает имя окна, в котором должны обрабатываться события, связанные с изменением элементов формы. Для этого требуется наличие окна или фрейма с заданным именем. В качестве значений данного атрибута могут использоваться и зарезервированные имена **_blank, _parent, _self и _top.**
- Атрибут **action** задает адрес URL сервера, который будет получать данные из формы и запускать соответствующий CGI-скрипт. Также можно послать данные из формы по электронной почте, указав при этом значения этого атрибута адрес URL типа mailto: . . .

Формы, передаваемые на сервер, требуют задания метода передачи (submission), который указывается при помощи атрибута method.

Метод GET присоединяет данные формы к строке адреса URL, заданного в атрибуте action.

При использовании **метода POST** информация из формы посылается как отдельный поток данных.

В последнем случае CGI-скрипт на сервере считывает эти данные из стандартного входного потока (standard input stream).

Кроме того, на сервере устанавливается переменная среды с именем QUERY_STRING, что обеспечивает еще один способ получения этих данных.

- **Атрибут enctype задает тип кодировки MIME (Multimedia Internet Mail Extensions) для посылаемых данных. Типом MIME по умолчанию является тип application/x-www-form-urlencoded.**
- К свойствам и методам формы в JavaScript-прграмме можно обратиться одним из способов:

  formName.propertyName

  formName.methodName (parameters)

  forms[i].propertyName

  forms[i].methodName (parameters)

- Здесь formName соответствует атрибуту name объекта form, а i является целочисленной переменной, используемой для обращения к отдельному элементу массива forms, который соответствует определенному тегу <form> текущего документа.

Использование массива forms

- К любой форме текущего гипертекстового документа можно обращаться как к элементу массива forms. Для этого необходимо указать индекс запрашиваемой формы. Например, forms[0] - первый тег <form> в текущем документе.

```
document.forms[i]
document.forms.length
document.forms['name']
```

- Переменная i - это индекс, соответствующий запрашиваемой форме.
- Выражение вида

```
document.forms[i]
```

- можно также присвоить переменной

```
var myForm = document.forms[i];
```

Теперь, если в форме имеется, к примеру, поле ввода, определенное в HTML-теге

```
<form>
<input type=text name=myField size=40>
. . .
</form>
```

- то в JS-программе к этому полю позволяет обращаться переменная myForm. В частности, при помощи следующего оператора содержимое данного поля ввода присваивается новой переменной с именем result:

```
var result = myForm.myField.value;
```

- Значение свойства length соответствует количеству форм в документе:

  var numForms = document.forms.length
- Массив forms содержит данные, которые используют только для чтения.

Свойства

Объект form имеет шесть свойств, большинство из них соответствуют атрибутам тега <form>:

- action - соответствует атрибуту action;
- elements - массив, содержащий все элементы формы;
- encoding - соответствует атрибуту enctype;
- length - количество элементов в форме;
- method - соответствует атрибуту method;
- target - соответствует атрибуту target

Массив forms имеет только одно свойство length - количество форм в документе.

Методы

- Метод submit() применяется для передачи формы из JavaScript-программы.
- Его можно использовать вместо тега **<input type=submit>**, имеющегося в большинстве форм, информация которых должна передаваться на сервер.

Обработчики событий

- Обработчик события onSubmit() позволяет перехватывать события, связанные с передачей данных формы. Такие события возникают либо после нажатия кнопки передачи данных, определенной тегом <input type=submit> в контейнере <form>, либо при передаче данных формы с помощью метода submit(), вызванного из JS-прграммы.

Пример. При нажатии кнопки передачи данных содержимое текстового поля посылается адресату по электронной почте:

- Отсюда вы можете послать почту. Перед отправкой последует запрос на отправку почты, - это срабатывает защита на вашем компьютере. Ничего, кроме содержимого формы не отправит!

```
<form nethod="post"
action="mailto:my@mail.ru"
enctype="text/plain">
<input type="submit" value="Отправить почту">
<input type="reset" value="Очистить форму">
<textarea name="email" rows=5 cols=60>
</textarea>
</form>
```

Объекты, соответствующие тегам HTML - 2

Массив frames

- К отдельным фреймам можно обращаться при помощи массива frames и свойства parent.
- Например, если имеется два фрейма, определенных в HTML-тегах:

```
<frameset rows="50%,50%">
<frame name="top" src="file1.htm">
<frame name="bot" src="file2.htm">
</frameset>
```

- Для обращения к первому фрейму вы можете использовать выражение parent.frames[0], и соответственно ко второму - parent.frames[1]. Таким образом, для обращения к отдельным фреймам и к свойству length массива frames используются выражения вида:

  frameRef.frames[i]

  frameRef.frames.length

  windowRef.frames[i]

  windowRef.frames.length

- Для определения количества фреймов во фреймосодержащем документе применяется свойство length. Все данные массива frames предназначены только для чтения.

Свойства

Объект frame имеет следующие свойства:

- frames - массив, содержащий все фреймы в окне;
- name - соответствует атрибуту name тега <frame>;
- length - количество дочерних фреймов в родительском окне (фрейме).

Кроме того, можно использовать такие синонимы:

- parent - синоним для окна или фрейма с текущим фреймосодержащим документом;
- self - синоним для текущего фрейма;
- window - синоним для текущего фрейма.

Массив frames имеет всего одно свойство length, значением которого является количество дочерних фреймов в родительском фрейме.

Методы и обработчики событий

- Во фреймосодержащих документах могут быть использованы методы **clearTimeout()** и **setTimeout()**.
- В теге **<frameset>** определяют обработчики событий, связанные с загрузкой и выгрузкой документов **onLoad** и **onUnLoad**.
- Об этих методах и событиях будем говорить позже. Пока мы ими пользоваться не будем. Забегать вперед тоже не очень хорошо.

Скрытый объект

- Что это такое.
- Это поле, которое может передаваться из формы например на сервер, находиться в тегах <form> . . . </form>, при этом не отображаться на экране. Для чего оно нужно? Ну например, что-то формируется JS программой и это нужно передать, при этом выводить эту информацию нет смысла. Это текстовые поля позволяют сохранять определенные значения в структурах, отличных от переменных языка JS, хотя данные значения существуют до тех пор, пока загружен текущий документ. Скрытое поле, как уже говорилось является свойством объекта form и должно помещаться в тегах <form> . . . </form>.

HTML-тег имеет синтаксис:

<input type="hidden"
[name="hiddenName"]
[value="textValue"]>

- Атрибут name задает имя поля и является необязательным. Значение текстового поля указывают при помощи атрибута value, который позволяет задавать и значение поля по умолчанию. К свойствам скрытых объектов можно обращаться посредством одного из следующих выражений:

  fieldName.propertyName

  formName.elements[i].propertyName

- где fieldName - имя скрытого поля, заданное в атрибуте name тега <input>, а formName - имя формы, в которой определено скрытое поле.

Атрибут name задает имя поля и является необязательным. Значение текстового поля указывают при помощи атрибута value, который позволяет задавать и значение поля по умолчанию. К свойствам скрытых объектов можно обращаться посредством одного из следующих выражений:

fieldName.propertyName

formName.elements[i].propertyName

где fieldName - имя скрытого поля, заданное в атрибуте name тега <input>, а formName - имя формы, в которой определено скрытое поле.

Свойства

Скрытый объект имеет свойства:

- name - соответствует атрибуту name тега <input>;
- value - соответствует атрибуту value тега <input>;
- type - соответствует атрибуту type и содержит значение "hidden".

Скрытые объекты не имеют методов и обработчиков событий.

Пример

- В следующей форме определено скрытое поле hfield шириной 20 символов, по умолчанию имеет значение "page 1":

```
<form name="hiddenField">
<input name="hfield" type="hidden"
  size=20 value="page 1">
</form>
```

Значение этого поля можно изменить с помощью оператора следующего вида:

```
document.hiddenField.hfield.value = "page 2";
```

Объект image и массив images

Браузер Microsoft Internet Explorer версии ниже 4, не поддерживает массив images.

В браузере рисунки рассматриваются как объекты image, а все рисунки, содержащиеся в текущем документе, помещаются в массив images, который можно использовать для обращения к любому рисунку, определяемому тегом <img>.

В частности, можно динамически обновлять изображения, изменяя их свойство src.

Для начала приведем тег <img>, распишем полностью:

```
<img src="fileOrUrl"
    [alt="altText"]
    [lowsrc="fileOrUrl"]
    [width=w]
    [heigth=h]
    [border=b]
    [vspace=v]
    [hspace=h]>
```

В атрибуте src содержится имя или адрес URL файла, который нужно вывести в документе. Рисунок должен храниться в формате GIF, JPEG, или PNG. С помощью атрибута **alt** задается альтернативный текст, появляющийся на экране: в момент загрузки текста, если пользователь заблокировал вывод изображений и поясняющая надпись под курсором мыши. Атрибут **lowsrc**, NN его поддерживает, IE не имеет смысла его использовать. Он позволяет предварительно выводить на экран изображение с низким разрешением. При этом рисунок загружается в два этапа. Атрибуты **width** (ширина) и **height** (высота) позволяют задать размеры рисунка в пикселах, атрибут **border** - ширину рамки в пикселах, а атрибуты vspace и **hspace** - размеры вертикального и горизонтального зазоров между границами изображения и другими элементами документа.

Для обращения к свойствам объекта image используется следующий синтаксис:

document.images[i].propertyName

где i - индекс элемента массива, который соответствует нужному рисунку.

Первым рисунком в документе будет **document.images[0]**.

Массив images является свойством объекта document, поэтому при обращении к рисунку необходим префикс **document** к имени массива.

Тег **<img>** не имеет атрибута name, поэтому выражение вида "**document.imgName**" приведет к ошибке.

Свойства. Все свойства объектов image соответствуют атрибутам тега <img>, за исключением свойства complete.

Эти свойства, кроме свойств src и lowsrc, значения которых могут быть изменены динамически, имеют значения только для чтения:

- src - соответствует атрибуту src тега <img>;
- lowsrc - соответствует атрибуту lowsrc тега <img>;
- height - соответствует атрибуту height тега <img>;
- width - соответствует атрибуту width тега <img>;
- border - соответствует атрибуту border тега <img>;
- vspace - соответствует атрибуту vspace тега <img>;
- hspace - соответствует атрибуту hspace тега <img>;
- complete - содержит булево значение, которое указывает, загружен рисунок в браузер или нет (true - загружен, false - нет);
- type - для объектов image содержит значение "image".

Перед загрузкой рисунка появляется его рамка, внутри которой отображается строка, заданная в атрибуте **alt** (в версии 5 и выше IE, пользователь при желании может отключить рамки с alt-текстом отображаемые в момент загрузки рисунка).

Рисунок можно изменять динамически, присваивая атрибуту src или lowsrc в качестве значения новый адрес URL

(локально проверить это не удается, так как lowsrc загрузится мгновенно. Ошибок по крайней мере при применении этого атрибута не выдает.)

Методы и обработчики событий.

Объект image не имеет методов.

Обработчики событий:

- **onAbort** - обработка события, возникающего при прерывании загрузки рисунка, т.е. при нажатии клавиши [Esc] или активизации новой гиперсвязи, в то время, когда рисунок загружается;
- **onError -** обработка события, связанного с ошибкой загрузки рисунка, т.е. когда невозможно найти рисунок по указанному адресу URL ;
- **onLoad** - соответствующее событие, инициализируется в начале загрузки рисунка. При загрузке анимированного GIF-а это событие возникает несколько раз и зависит от числа кадров анимационной последовательности.

Объект link и массив links

- Объект **link (гиперсвязь)** отображается как участок текста или графического объекта, щелчок мыши на котором позволяет перейти к другому Web-ресурсу. Тег языка HTML, а мы помним, что рассматриваем объекты соответствующие тегам HTML, имеет следующий вид:

```
<a href=locationOrURL
[name="anchorName"]
[target="windowOrFrameName"]
[onClick="handlerText"]
[onMouseOver="handlerText"]>
linkText
</a>
```

- Атрибут **href** определяет имя файла, или адрес URL для объекта, который загружается при активизации гиперсвязи.
- Атрибут **name** задает имя гиперсвязи, превращая ее в объект **anchor** (метку).
- С помощью атрибута **target** в определенный фрейм текущего фреймосодержащего документа можно загрузить документ, URL которого указан в значении атрибута href.
- Атрибут **linkText** представляет собой текст, отображаемый в HTML-документе как гиперсвязь, которая активизируется щелчком мыши.
- Для обращения к свойству объекта link используются выражения типа:

document.links[i].propertyName

где i - индекс данной связи в массиве гиперсвязей links текущего документа.

Масив links

- В программе на языке JavaScript к гиперсвязям можно обращаться как к элементам массива links. Например, если в документе определены два тега < a href>, то в JS-программе к этим гиперсвязям можно обращаться с помощью выражений **document.links[0]** и **document.links[1]**.
- Синтаксис выражений для обращений к массиву links следующий:

document.links[i]

document.links.length

где переменная i - индекс гиперсвязи. Значением свойства length является количество гиперсвязей в текущем документе.

Объекты link представляют собой объекты только для чтения, поэтому динамически изменять гиперсвязи в документе нельзя.

Свойства

Для объекта link определены следующие свойства:

- hash - задает имя метки в адресе URL, если она существует ;
- host - задает часть hostname:port адреса URL, определенного в гиперсвязи;
- hostname - задает имя хоста и домена (или IP-адрес) в адресе URL, определенном в гиперсвязи;
- href - задает полный адрес URL, определенный в гиперсвязи;
- pathname - задает часть адреса URL, которая описывает путь к документу и находится после части hostname:port;
- port - задает коммуникационный порт, который использует сервер;
- protocol - задает начало адреса URL, включая двоеточие, например http:;
- target - соответствует атрибуту target тега <a href>.

Массив links имеет всего одно свойство, length, значением которого является количество гиперсвязей в текущем документе.

Методы и обработчики событий

- Для **объекта link** методы не определены.
- В **тегах <a href>** могут использоваться обработчики событий щелчка мыши и ее перемещения - onClick и onMouseOver.

Пример

- При подведении указателя мыши на гиперсвязь, в строке состояния браузера появится текст "Текст в строке состояния при подведении мыши на гиперсвязь".

```
<a href="#" onMouseOver="window.status='Текст в строке состояния при подведении мыши на гиперсязь';
  return true">
Подведите сюда курсор мыши
</a>
```

- В данном случае гиперсвязь указывает на пустой документ - "#". Это выбрано для примера в случае щелчка на гиперсвязи ничего не грузилось.

Модифицирование веб-страниц

Для генерирования нового и модификации уже имеющегося HTML-кода на странице первым делом вы должны идентифицировать элемент (тег) на странице, а далее выполнить над ним какие-либо действия.

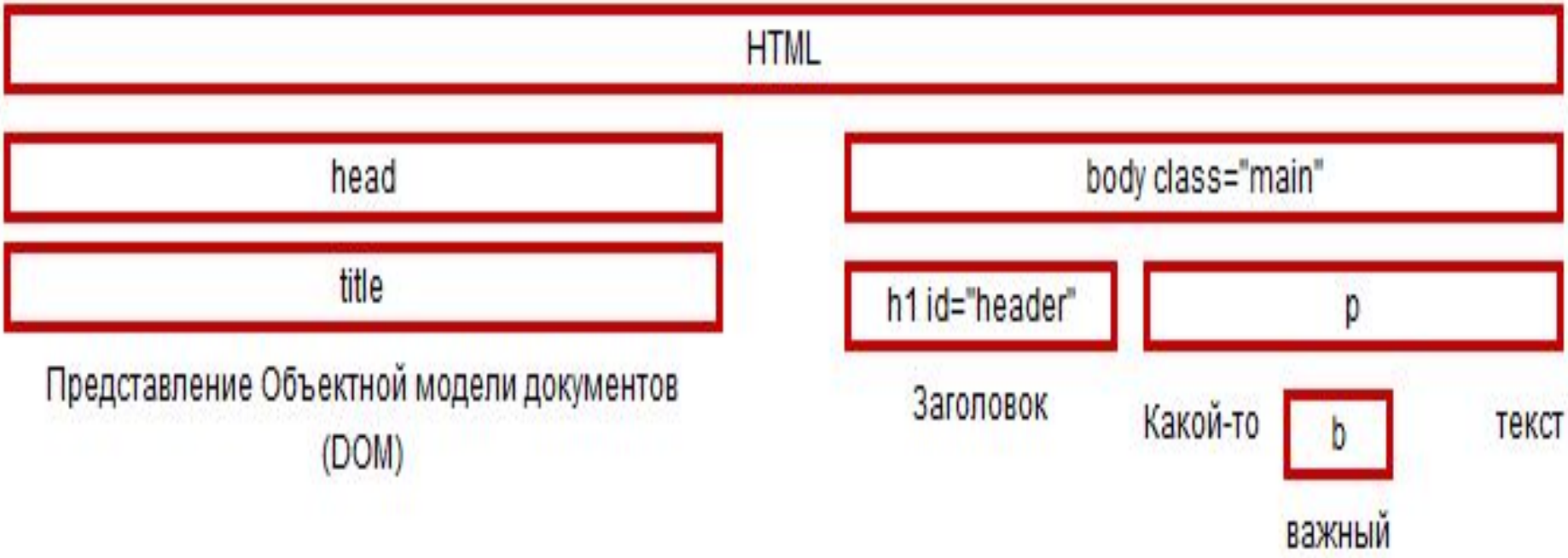
При загрузке HTML-документа на страницу выводится его содержимое, при этом браузер запоминает модель HTML, т.е. теги, их атрибуты и последовательность их появления на странице.

Во время загрузки страницы Web-обозреватель создает экземпляр объекта **HTMLDocument** и сохраняет его в переменной **document**, дополнительно считывает код HTML элементов и создает их внутреннее представление в виде экземпляров соответствующих объектов. Так, для абзаца создается экземпляр объекта **HTMLParagraphElement**, для гиперсылки - **HTMLLinksElement**, для картинки - **HTMLImageElement**, для таблицы - **HTMLTableElement** и т.д.

Текстовое содержимое (узлы) представлено как экземпляр объекта Text. Все объекты, представляющие элементы страницы являются потомками объекта HTMLElement.

Такое представление содержимого называется **объектной моделью документа (Document Object Model), сокращенно DOM.**

```
<html>
<head>
<title>Представление Объектной модели документа
  (DOM)</title>
</head>
<body class="main">
<h1 id="header">Заголовок</h1>
<p>Какой-то <b>важный</b> текст</p>
</body>
</html>
```



- Структура HTML-страницы часто изображается в виде генеалогического древа, где одни теги включают в себя другие и называются родительскими (предками), а вложенные теги - дочерними (потомками). Теги h1 и p называются сестринскими (братскими) и они также являются дочерними элементами по отношению к body.

Война браузеров

- Объектная модель документов сама по себе не является частью диалекта JavaScript, это стандарт консорциума W3C, к которому производители большинства браузеров привели свои программы.
- Объектная модель документов позволяет JavaScript обмениваться информацией с веб-страницей и изменять на ней HTML-код. Но проблема в том, что Web-обозреватели по-разному поддерживают объектную модель документа.
- К примеру, Internet Explorer (IE) никак не обрабатывает промежутки между тегами, незаполненные текстом, а Opera и Firefox подсчитают их за пустой текст. Web-сценарий тоже представлены по-разному.

Получение доступа к узлам

- Для того, чтобы получить доступ к отдельному элементу на странице нужно его как-нибудь обозвать.
- В примере выше мы присвоили тегу <h1> необязательный атрибут id со значением header и теперь можем получить к нему доступ.
- Существует три способа: прямой доступ; через коллекции и с помощью свойств и методов объектной модели документа (DOM).
- *Работа с последним - является хорошим тоном программирования.*

Прямой доступ

- Используя такой способ, мы обращаемся к элементу прямо по имени, затем пишем свойство или метод:

```
header.style.color = '#cc0000';
// устанавливаем бордовый цвет заголовка
```

Доступ через коллекции

Коллекция - представлена в виде ассоциативного массива. Объект HTMLdocument поддерживает большое количество коллекций:

// Экземпляры объекта HTMLCollection, кроме последнего

- all Все элементы страницы
- anchors Все якоря страницы
- applets Все элементы ActiveX
- embeds Все модули расширения
- forms Все Web-формы
- images Все графические изображения
- links Все гиперссылки
- scripts Все Web-сценарии (только IE & Opera)
- styleSheets Все таблицы стилей

- Доступ к нашему элементу мы можем получить по строковому индексу, который совпадает с именем элемента страницы:

document.all['header'];

// получение доступа через коллекции

- Также доступ можно получить подставив числовой индекс элемента страницы. Например, код доступа к первой картинке на странице следующий:

document.images[0];

// доступ к самому первому изображению, если оно есть

Доступ с помощью свойств и методов объектной модели документа (DOM)

Существует два основных метода доступа к узлам:

getElementById()

getElementsByTagName()

Метод getElementById()

Находит нужный элемент с определенным идентификатором. В нашем случае заголовок h1 имеет уникальный id со значением header:

```
// объектная модель документа (DOM)
var ourHeader = document.getElementById('header');
ourHeader.innerHTML = 'Объектная модель документа';
```

Команда getElementById() - это метод объекта document, а 'header' - простой литерал переданный как параметр, обозначающий уникальность имени идентификатора. Причем в качестве параметра может быть и переменная.

В примере выше мы получили доступ к нашему заголовку и произвели его замену, используя свойство innerHTML.

Атрибут name

Аналогичный подход можно применить и с помощью атрибута name:

```
<p name="newAtr">Новый параграф</p>
```

В этом случае для получения доступа к узлу применяется метод **getElemenstByName(),** который возвращает массив экземпляров объекта HTMLElement с данным именем:

```
var newPar = document.getElementsByName('newAtr');
var result = newPar[0];
```

Свойства и методы объекта HTMLElement

Объектная модель документа предлагает несколько способов доступа к соседним узлам.

Рассмотрим их.

Свойство childNodes

Содержит все дочерние элементы по отношению к текущему и при этом вложены непосредственно в него.

Похож на массив, возвращенный методом **getElementsByTagName**:

```
// Объектная модель документа
  var head = document.getElementById('header');
// получаем доступ к тегу h1
var kinder = head.childNodes;
// находим дочерний узел (сам вложенный текст)
var textKinder = kinder[0].nodeValue;
// вытаскиваем текст с помощью свойства nodeValue
alert(textKinder);
// выводим результат в модальное окно
```

В нашем примере первым делом получаем доступ к заголовку h1 с уникальным идентификатором header. Первый и единственный дочерний элемент - сам текст.

Определяем его с помощью свойства **.childNodes**.

Стоит отметить, что мы получили только доступ к тексту, чтобы его вывести мы используем свойство

.nodeValue.

Свойство firstChild

- Возвращает первый дочерний элемент по отношению к текущему.
- Если дочерних элементов нет, возвращается значение null:

```
// Объектная модель документа
var head = document.getElementById('header');
var firstKind = head.firstChild;
var val = firstKind.nodeValue;
alert(val);
```

Свойство lastChild

- Возвращает последний дочерний элемент по отношению к текущему, т.е. **антипод** свойства **firstChild**.
- Если текущий элемент не содержит дочерних элементов, возвращается значение null:

```
// Объектная модель документа
    var lastKind = document.body.lastChild;
// ссылка на последний элемент тела страницы
```

Свойство parentNode

- Возвращает родительский элемент по отношению к текущему:

// Объектная модель документа

```
var head = document.getElementById('header');
var predok = head.parentNode;
```

// в нашем примере ссылаемся на тег body

Свойство nextSibling

- Указывает на узел, следующий за текущим. Если элемент последний, то возвращает значение null:

```
// Объектная модель документа
  var head = document.getElementById('header');
var nextel = head.nextSibling;
// в нашем примере ссылаемся на след. тег p
if(! nextel) {                    // если элемент последний, то...
          alert('Элемент является последним!');
          }
else {
     var val = nextel.lastChild.nodeValue;  // вытаскиваем
   последний дочерний элемент
     alert(val); // выводим результат в модальное окно
  };
```

Свойство previousSibling

- Указывает на узел, предыдущий по отношению к текущему.
- Если элемент первый, то возвращает значение null:

```
// Объектная модель документа
var x = document.body.lastChild;
var prev = x.previousSibling;
```

Метод hasChildNodes

- Не принимает параметров и возвращает значение true, если находит дочерние элементы, в противном случае возвращает значение false:

```
// Объектная модель документа
var head = document.getElementById('header');
var nextel = head.nextSibling;
// в нашем примере ссылаемся на след. тег p
var result = nextel.hasChildNodes();
alert(result);
```

Определение событий

- Все действия пользователя (нажатие на кнопки клавиатуры, клики мыши или ее перемещение, загрузка страницы, наведение фокуса и т.п.), на которые реагирует веб-обозреватель, именуются **событиями**.
- Язык JavaScript - клиентский язык, т.е. событийно-управляемый. Без него страницы были бы не в состоянии отвечать на действия посетителя или предлагать что-либо интерактивное, динамичное и впечатляющее.
- Подготовка Web-страницы к ответу на события проходит в два этапа:

  - идентифицируем элемент страницы, реагирующий на событие;

  - присваиваем событие обработчику

Сразу отметим, что существуют разные модели обработки событий.

Одна из которых стандартизирована объектной моделью документа, ее еще называют модель Firefox, она более прогрессивная и в ней больше возможностей, но зато не поддерживается Internet Explorer (IE < 8). Другие более простые, но зато поддерживается всеми современными Web-обозревателями. Рассмотрим их.

Встроенные javascript события

- Один из самых простых и непрофессиональных способов исполнения функции в момент запуска события называют регистрацией встроенных событий, когда обработчик события присваивается прямо в HTML-код:

```
// javascript события
<p> Нажмите на ссылку и получите результат! А вот и сама
<a href="#" ONCLICK="alert('событие onclick javascript');">Ссылка на javascript события мыши</a>
</p>
```

Привязка через свойства объектов

Обработчики событий оформляются в виде функции в случае их привязки к событиям через соответствующие свойства объектов, представляющих элементы страницы:

```
<p id="par">Наведите курсор мышки на текст</p>
<script type="text/javascript">
var text = document.getElementById('par');
text.onmouseover = function() {
                        this.style.color = '#ff0000';
                      };
text.onmouseout = function() {
                        this.style.color = '#000000';
                      };
</script>
```

Привязка через свойства объектов

```
// javascript события  (пример №2)

   <html>
   <head>
   <meta http-equiv="Content-Type" content="text/html; charset=windows-1251" />
   <title>Введение в JS</title>
<script type="text/javascript">
function parClick() {
        alert('Вы кликнули по абзацу и получили ответ от javascript события мыши');
                };
</script>
  </head>
  <body>
<p id="par2">Кликни меня</p>
<script type="text/javascript">
var text = document.getElementById('par2');
text.onclick = parClick;
</script>
  </body>
  </html>
```

Привязка обработчика к событию с помощью функции-слушителя

- Стандартами DOM рекомендуется использование именно этого способа, но, к большому сожалению, не все Web-обозреватели его поддерживают.
- В этой модели привязка к заданному элементу страницы и событию осуществляется с использованием метода **addEventListener()** объекта HTMLElement:

```
// javascript события
    <html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-1251" />
  <title>javascript события</title>
<script type="text/javascript">
                var n = 0;
function parClick() {
                var val = document.getElementById('par3');
                if(n == 1) {
                        val.style.color = '#000000';
                        n--;
                    }
                else {val.style.color = '#ff0000';
                        n++;
                    };
            };
</script>
 </head>
 <body>
<p id="par3">Кликни меня два раза или более.</p>
<script type="text/javascript">
var text = document.getElementById('par3');
text.addEventListener('click', parClick, false);
</script>
 </body>
 </html>
```

В качестве первого параметра передается имя события в виде строки формата DOM (строка кода №25). Вторым параметром передается сама функция-слушитель (parClick). Третий параметр указывает Web-обозревателю, следует ли перехватывать события, возникающие в дочерних по отношению к текущему элементах страницы (булевое значение false - отключает перехват, true - включает).

Метод removeEventListener объекта HTMLElement позволяет удалить подключенную ранее функцию-слушателя:

```
// javascript события
  text.removeEventListener('click', parClick, false);
```

Internet Explorer использует метод **attachEvent()** для выполнения той же задачи:

```
// javascript события
  text.attachEvent('onclick', parClick);
```

сделаем так, чтобы работало и в Explorer:

```
// javascript события
    <html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-1251" />
  <title>javascript события</title>
<script type="text/javascript">
var textColor = ['#330000','#006600','#cc0000',‘
#3366ff','#660033','#cc6600'];
var textBgColor = ['#99fff','#cccc66','#33ffff','#cc9999',
'#ff9999'];
function parClick() {
            var val = document.getElementById('par3');
            val.style.color = textColor[Math.round(Math.random() * 6)];
            val.style.backgroundColor = textBgColor[Math.round(Math.random() * 5)];
          };




</script>
  </head>
  <body>
```

Объект Event

- Web-обозреватели позволяют нам получить дополнительную информацию о событиях, например, о том, была ли нажата клавиша или координаты курсора мыши.
- Для этих целей объектная модель документа предусматривает особый объект **Event**, который поддерживает весьма большой набор свойств, дозволяющих нам отслеживать каждое наступившее событие. Следует также отметить, что получение такой информации в разных браузерах выполняется по разному и модель обработки события в этом случае не играет никакой роли!

Предположим, что у нас есть элемент DIV, в котором мы поместили абзац со ссылкой внутри:

```
<div id="main">
<p id="par1">Это абзац со ссылкой внутри
<a id="link1"
href="10-2.html">ССЫЛКА</a></p>
</div>
```

Теперь присвоим обработчик событию щелчка по ссылке, где пропишем, что никуда переходить нам не нужно и выведем сообщение в модальное окно:

```
// Всплытие и перехват событий   *** JS-код ***
  window.onload = scriptAfterLoad; // выполнит ф-цию scriptAfterLoad после загрузки стр.

function scriptAfterLoad() {
        var link1 = document.getElementById('link1');
        if(link1.addEventListener) {
        link1.addEventListener('click', clickLink, false);
        }
        else {
        link1.attachEvent('onclick', clickLink)
        };
}; // end scriptAfterLoad Func
  function clickLink(event) {
        alert('Кликнули по ссылке!');
        (window.event) ? event.returnValue = false : event.preventDefault();
}; // end Func
```

- Если вы сохраняете сценарий в отдельный файл, который прикрепляете к странице между парным тегом **head** или просто прописываете вначале, то нужно исполнить его только после загрузки страницы.
- В этом случае нам поможет событие **load** объекта **window**. После этого первым делом идентифицируем наш элемент - ссылку.
- После определяем метод с которым будем работать: **attachEvent()** для Internet Explorer или **addEventListener()** для остальных, поддерживающих стандарты консорциума W3C, внутри присваиваем событие, на которое нужно реагировать и функцию обработчик **clickLink**.
- В обработчике с помощью команды **alert()** выводим в модальное окно сообщение, что кликнули именно по ссылке и далее пишем условие, в зависимости от типа браузера, т.к. в Web-обозревателе Firefox свои свойства и методы для событий, которые отличаются от остальных, но есть и схожие, правда их не так много.
- Чтобы отменить поведение для данного события в Firefox используют вызов метода **preventDefault** объекта **Event**, который не принимает никаких параметров и ничего не возвращает.
- Для этих же целей в остальных браузерах поддерживается свойство **returnValue**. Значение false отменяет поведение.

Теперь представим, что к тегу <p> вам нужно привязать еще одно событие click. Допишем наш скрипт и выведем на экран сообщение о щелчке по абзацу:

```
// Всплытие и перехват событий   *** JS-код ***
  window.onload = scriptAfterLoad;
  function scriptAfterLoad() {
     var mainDiv = document.getElementById('main');
     mainDiv.style.border = '1px solid #cc0000'; // сделали рамку для div
          var link1 = document.getElementById('link1');
     if(link1.addEventListener) {
     link1.addEventListener('click', clickLink, false);
     }
     else {
     link1.attachEvent('onclick', clickLink)
     };
```

```
//*************************************************************//
	var par1 = document.getElementById('par1');
	par1.style.background = '#f9f9f9'; // background для абзаца
	if(par1.addEventListener) {
	par1.addEventListener('click', clickPar, false);
	}
	else{
		par1.attachEvent('onclick', clickPar)
		};
}; // end scriptAfterLoad Func


function clickPar(event) {		// для ссылки
	alert('Кликнули по абзацу!');
}; // end Func

  function clickLink(event) {		// для абзаца
	alert('Кликнули по ссылке!');
	(window.event) ? event.returnValue = false : event.preventDefault();
}; // end Func
```

Все необходимые сведения о Web-обозревателе и системе у пользователя можно узнать при помощи объекта **Navigator**.

Данный объект поддерживаем множество свойств и один бесполезный метод javaEnabled, который не принимает параметров и возвращает true, если выполняются сценарии JavaScript и false - в противном случае:

- appCodeName Возвращает имя исходного кода программного ядра обозревателя.
- appName Возвращает имя программы обозревателя.
- appVersion Возвращает версию программы обозревателя
- browserLanguage Возвращает код языка программы обозревателя*
- cookiEnabled Возвращает true, если разрешен прием куки, false - нет.
- cpuClass Возвращает наименование процессора клиентского компьютера*
- lanquage Возвращает код языка программы обозревателя*
- onLine Возвращает true, если клиент подключен к инету, false - нет*
- platform Возвращает обозначение операционной системы клиентского компьютера.
- systemLanquageВозвращает код языка операционной системы клиента.
- userAgent Возвращает строку индексирующую обозреватель
- userLanquage Аналог browserLanquage*

Примечание * - свойство поддерживается отдельными браузерами.

Введение в регулярные выражения

Что такое регулярные выражения?

Регулярные выражения (RegExp - regular expressions) это объект, который может описывать шаблон символов.

Например, если вы ищете в строке подстроку, вы можете описать шаблон того, что вы ищете.

Простой шаблон может состоять даже из одного символа, большие и сложные шаблоны могут использоваться для парсинга, проверки формата введенных данных, замены и других различных целей.

Содержание шаблона регулярного выражения состоит из последовательности символов. Большая их часть (все буквы и цифры) описывают сами себя - непосредственно указывают свое присутствие. Например, регулярное выражение

/hello/

совпадет со всеми строками, в которых есть слово "hello". Другие символы обозначают не себя, а имеют некоторое модифицирующее значение. Например, выражение

/hello$/

будет соответствовать строкам, которые ЗАКАНЧИВАЮТСЯ на слово "hello". Это обеспечивает метасимвол "$", обозначающий конец строки.

Создание объектов RegExp

Объекты RegExp могут быть созданы с помощью конструктора RegExp(), или с помощью литералов. Но если, например, строковые литералы задаются в виде символов, заключенных в кавычки, то литералы регулярных выражений задаются заключенными в пару символов слэш "/". Например, вот так:

```
var myPattern = /q$/;   //Создание регулярного выражения с помощью литерала
```

В данном примере мы создали с помощью литерала новый объект типа RegExp и присвоили его переменной myPattern. Данный шаблон соответствует любой строке, заканчивающейся символом q.

Для создания такого же объекта с помощью конструктора нам нужно написать вот такое выражение:

```
var myPattern = new RegExp("q$");  //Создание регулярного выражения с помощью //конструктора
```

Теперь давайте рассмотрим из чего можно составлять регулярные выражения. Первая группа это конечно же символы. Как мы уже говорили все алфавитные и цифровые символы обозначают сами себя. Также можно вводить некоторые не алфавитные символы с помощью последовательностей, начинающихся с обратного слэша.
Цифры и буквы - соответствуют сами себе
\0 - Символ NUL (Соответствует \u0000 в Unicode)
\t - Табуляция (\u0009)
\n - Перевод строки (\u000A)
\v - Вертикальная табуляция (\u000B)
\f - Перевод страницы (\u000C)
\r - Возврат каретки (\u000D)
\xnn - Символ из набора Latin, задаваемый шестнадцатиричным номером nn
\unnnn - Символ Unicode, задаваемый шестнадцатиричным номером nnnn
\cX - Управляющий символ "X", например \cJ эквивалентна \n
Также в регулярных выражениях используются следующие символы:^ $. * + ? = ! : | \ / ( ) [ ] { }

В следующих шагах мы более подробно рассмотрим их значение и применение в комбинации с другими символами, однако сейчас нужно запомнить, что для определения смысла этих символов буквально, т.е. "самих себя", необходимо перед ними ставить символ обратного слэша.
Например, если вы хотите написать регулярное выражение, по которому будет находиться символ обратного слэша, то вы должны в выражение поставить этот символ, предваряемый таким же символом обратного слэша. В результате такое регулярное выражение будет выглядеть следующим образом:/\\/

Задача на разминку.
Укажите регулярные выражения, которые будут соответствовать какой-либо подстроке в строке "http://www.stepik.org";

- /\//
- ¯_(ツ)_/¯
- /WWW/
- /\/\/\//
- /www/
- /\\/
- /\\\//

Отдельные символы могут быть объединены в классы. Это обозначается набором символов, заключенных в квадратные скобки. Например регулярное выражение
/[0123456789]/
соответствует любой цифре.
Или же можно указав перед набором символов знак "^" определить регулярное выражение, которое будет соответствовать любому символу, КРОМЕ тех, которые указаны в скобках - класс с отрицанием. Например выражение
/[^0123456789]/
будет соответствовать любому символу КРОМЕ цифр.
Также в классах можно задавать диапазон с помощью знака дефиса "-", чтобы не перечислять все символы. Например все цифры можно обозначить таким выражением:
/[0-9]/

Некоторые классы из наборов символов настолько часто используются, что для них определили специальные обозначения:
[...] - любой из символов, указанных в скобках
[^...] - любой кроме символов, указанных в скобках
. (точка) - любой символ кроме перевода строки или другого разделителя строки
\w - эквивалентно [a-zA-Z0-9_] (Любой текстовый символ ASCII)
\W - эквивалентно [^a-zA-Z0-9_] (Любой символ кроме текстовых символов ASCII)
\s - любой пробельный символ из Unicode
\S - любой НЕпробельный символ из Unicode
\d - эквивалентно [0-9] (любые цифры ASCII)
\D - эквивалентно [^0-9] (все символы кроме цифр ASCII)
[\b] - обозначение символа "забой“

Последовательности таких управляющих символов также можно объединить в класс, например регулярное выражение
/[\w\s]/
соответствует любому пробельному символу или символу ASCII - букве или цифре.

Отметьте выражения, которые будут соответствовать строкам или подстрокам.

- /[^qwerty]/ для строки “123qwerty456”
- /[^qwerty]/ для строки “qwerty”
- /\W/ для строки “qwerty”
- /[7-9]/ для строки “123qwerty456”
- /qwerty/ для строки”123qwerty456”
- /\w/ для строки “qwerty”
- /qwerty/ для строки “qwerty”
- /\d/ для строки “123qwerty456”

- Описанные выше шаблоны можно использовать не только для описания одиночных комбинаций символов, но и для сколь угодно многократных повторений. Это называют "квантификацией".
  Для квантификации в регулярных выражениях есть набор специальных комбинаций, заключаемых в фигурные скобки. Эта комбинация в фигурных скобках должна следовать сразу за описанным шаблоном. Например комбинация
  /\d{4}/
  соответствует числу, состоящему из 4-х цифр.
  Давайте рассмотрим какие бывают управляющие комбинации для повторений:
- {n} - обозначает ровно n экземпляров шаблона
- {n,} - обозначает n или больше экземпляров шаблона
- {n,m} - обозначает не менее n и не более m экземпляров шаблона
- ? - обозначает ноль или один экземпляр шаблона (эквивалентно выражению {0,1})
- + - обозначает 1 или более экземпляров шаблона (эквивалентно выражению {1,})
- * - обозначает ноль или более экземпляров шаблона (эквивалентно выражению {0,})

Эти комбинации повторения соответствуют максимально возможному количеству совпадений. Например выражение

/x{1,}/

примененное к строке "xxx" будет соответствовать максимальному количеству совпадений, т.е. всем трем буквам "x", встреченным в строке. Это называется "жадным" повторением. ("жадной" квантификацией).
Если же мы хотим ограничить поиск первым же вхождением, то может использовать так называемую "нежадную", или "ленивую" квантификацию. Для этого после управляющей комбинации повторений ставится символ "?". Таким образом выражение

/x{1,}?/

будет соответствовать только первому соответствию, т.е. только первой букве "x" в строке.
Важный момент! Признак "ленивости" действует только на тот квантификатор (подшаблон) в шаблоне, после которого стоит, все остальные квантификаторы остаются "жадными".

- Синтаксис регулярных выражений содержит специальный символ для определения альтернативы, т.е. можно указать больше одного варианта шаблона, соответствие которому будет проверяться. Для разделения альтернатив используется символ "|" - вертикальная черта.
- Например, выражение
- /ma|pa|da/

- будет соответствовать либо строке "ma" либо строке "pa" либо строке "da".

- Альтернативы конечно также могут комбинироваться и с классами и с повторениями. Указанный ниже шаблон соответствует либо двум цифрам либо двум строчным буквам либо двум заглавным буквам:
- /\d{2}|[a-z]{2}|[A-Z]{2}/

- Необходимо обратить внимание, что альтернативы обрабатываются слева направо до первого соответствия. После нахождения первого соответствия остальные альтернативы будут игнорироваться. На практике это означает что, например, шаблон /1|12|123/ примененный к строке "123", будет соответствовать первому символу, хотя в альтернативах есть гораздо более полное соответствие.

- Рассмотрим еще одну возможность, которую мы можем использовать в регулярных выражениях - группировку.
- Группировка обозначается заключением подшаблона в круглые скобки ( ). При этом элементы, используемые совместно со специальными символами, например |, +, *, ? и другие, будут рассматриваться как одно целое.
- Например шаблон
- /regular(expression)?/

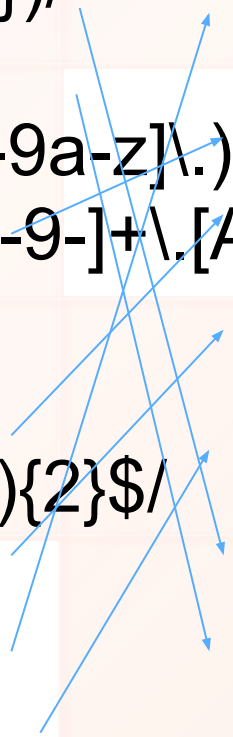
- будет соответствовать слову "regular" за которым следует необязательное слово "expression".

- Еще одна возможность регулярных выражений - указание границы соответствия. Для этого используются **якорные выражения**.
- Часто нам нужно найти слово, находящееся на отдельной строке, или в начале строки. Как вариант, может понадобиться найти отдельное слово, однако просто задать шаблон, в котором слово будет обрамлено пробелами мы не можем - в выборку не попадут слова с которых начинаются строки, или которыми заканчиваются. Также граница слова может определяться любым знаком препинания в тексте и перечислять в шаблоне все возможные комбинации было бы достаточно утомительно. Давайте рассмотрим, какие специальные символы используются для определения границ:
- **^** - соответствует началу строки при многострочном поиске или началу строкового выражения
  $ - соответствует концу строки при многострочном поиске или концу строкового выражения
  \b - соответствует границе слова, т.е. позиции между текстовым (aA-zZ) и не-текстовым символом, либо между текстовым символом и началом или концом строки.
  \B - Соответствует позиции, не являющейся границей слов.
  (?=p) - Позитивная опережающая проверка на последующие символы - убеждается в том, что последующие символы соответствуют шаблону "p" но не включает их в результат поиска.
  (?!p) - Негативная опережающая проверка на последующие символы - требует чтобы последующие символы НЕ соответствовали шаблону "p".

- Ну и последний элемент синтаксиса регулярных выражений - **флаги**. Флаги задают глобальные правила для всего шаблона и указываются не внутри символов слэша, в которые заключен шаблон, а ПОСЛЕ них.
- JavaScript поддерживает три варианта флагов:
  i - указывает на то, что поиск по шаблону должен быть не чувствительным к регистру
  g - указывает что поиск должен быть глобальным, т.е. должны быть найдены ВСЕ соответствия в строке
  m - указывает на то, что поиск должен производиться в многострочном режиме.

Сопоставьте регулярное выражение типу данных, которым оно соответствует

/([\d]{2})\.([\d]{2})\.([\d]{4})/
/^\d{6}$/
/^([0-9a-z]([0-9a-z\-])*[0-9a-z]\.)+[a-z]{2,4}$/i
/[A-Z0-9._%+-]+@[A-Z0-9-]+\.[A-Z]{2,4}/igm
/\#([a-fA-F]|[0-9]){3,6}/
/-?\d{1,3}\.\d+/
/^([0-1]\d|2[0-3])(:[0-5]\d){2}$/



- Широта или долгота
- Доменное имя
- Адрес электронной почты

Использование регулярных выражений

В данном уроке сначала мы рассмотрим методы класса String, позволяющие использовать регулярные выражения:

search(regexp)
replace(regexp, newString)
match(regexp)
split(divider)

Первый и самый простой это метод search()
В качестве аргумента мы передаем ему регулярное выражение, а он нам в ответ возвращает номер позиции, с которой найдено соответствие шаблону, либо "-1" если соответствие не найдено.

```
var myString = "This is just test string";
result = myString.search(/is/);
```

В указанном примере в переменной result окажется число 2 (отсчет позиций начинается с 0).
Два важных момента:
1. Метод search() не поддерживает глобальный поиск и флаг g в составе регулярного выражения будет игнорирован.
2. Если аргумент не является регулярным выражением, то он будет преобразован в него передачей конструктору RegExp. (Объекты RegExp мы рассмотрим чуть позже).

Следующий метод, с которым мы познакомимся, это replace()
С его помощью можно выполнить операцию поиска с заменой.
В качестве аргументов он принимает регулярное выражение и строку замены.

```
var myString = "This is just test string";
result = myString.replace(/is/,"as");
```

Данный пример заменит первое найденное соответствие шаблону ("is") на подстроку "as", в результате в переменной result окажется строка "Thas is just test string".
Нужно отметить, что этот метод поддерживает глобальный поиск и при использовании флага "g" поменяет все найденные соответствия.

```
var myString = "This is just test string";
result = myString.replace(/is/g,"us");
```

В этом примере в переменной result окажется строка "Thus us just test string".
Еще необходимо отметить, что в качестве второго аргумента метода replace() может использоваться функция, в этом случае мы получим возможность динамического изменения строки замены.
Если в качестве первого аргумента окажется не регулярное выражение, то он будет также как и у метода search() преобразован в регулярное выражение с помощью конструктора RegExp. А если, например, вовсе забыть указать второй аргумент, то будет произведена замена всех найденных совпадений на undefined.

- Следующий интересующий нас метод это match().
- Он принимает в качестве аргумента регулярное выражение (или преобразовывает в него аргумент подобно предыдущим методам), а в качестве результата возвращает массив всех найденных соответствий.

- var myString = "У дедушки в деревне было 12 яблонь, 5 кустов смородины, 10 кур и 33 коровы";
- result = myString.match(/\d{2}/g);

- В данном примере мы записали в регулярное выражение обозначение цифры "\d", указали что ищем её двойное повторение "{2}" и включили флаг глобального поиска "g". В результате в переменной result окажется массив [12, 10, 33].

- Однако если флаг глобального поиска не будет указан, то в массив попадет только первое совпадение, оно запишется нулевым элементом. Остальными элементами массива будут подстроки, соответствующие всем подвыражениям, если таковые имеются.

- И последний из методов объекта String, позволяющих работать с регулярными выражениями, является split()
- Он разбивает строку на массив подстрок, используя в качестве разделителя содержимое аргумента, которое в том числе может быть и регулярным выражением. Например, использовав в качестве разделителя два слэша можно отделить в веб-адресе протокол от собственно наименования сайта:

- var myString = "http://www.example.com/download/pictures";
- result = myString.split(/\/{2}/g);

- Результатом данного выражения станет массив из двух элементов: "http:" и "www.example.com/download/pictures"
- Таким же образом, например, можно разбить на массив строку состоящую из цифр:

- var myString = "12345678909876543 21"
- result = myString.split(/\B/g);

- Используя в качестве разделителя определение символа, не являющегося границей слова (странное решение, но почему нет) получим в переменной result массив [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1].

понятно был изложен материал по методам объекта String, работающим с регулярными выражениями.

- Результатом работы метода split() является массив
- Метод search(regex) находит подстроку, соответствующую шаблону regex и возвращает её
- Если в методы replace и search передать вместо регулярного выражения строку, то она будет преобразована в регулярное выражение
- Метод search(regex) возвращает “0” если подстрока не найдена
- Метод replace(regex, str) поддерживает глобальный поиск
- Метод math() возвращает true, если regexp соответствует строке

Во второй части урока мы более внимательно рассмотрим объект RegExp - его свойства, методы и конструктор.

Начнем с конструктора. Как вы помните, конструктор, это метод, который принимает на вход некоторый аргумент, а на выходе создает объект необходимого вида.

Конструктор RegExp может принимать на вход строку, содержимое которой будет преобразовано в регулярное выражение. В строке должно быть содержимое регулярного выражения, т.е. тот текст, который обычно находится между двух слэшей.

```
var myPattern = new RegExp("q$");        // Создаем шаблон, находящий букву "q" в конце строки
```

Также в конструктор можно передать второй, необязательный аргумент, в котором можно указать флаги. Например:

```
var myPattern = new RegExp("q$","g");    // Создаем такой же шаблон, но добавляем флаг глобального поиска
```

Важный момент! Если в тексте присутствует символ обратного слэша "\", то его необходимо предварять таким же символом (т.е. писать "\\"), поскольку, как мы помним, обратный слэш используется в регулярных выражениях для указания управляющих символов.

- **Объект RegExp** в JavaScript имеет следующие **свойства**:
- **source** - собственно текст регулярного выражения
  ignoreCase - логическое значение обозначающее наличие флага "i", доступно только для чтения
  global - логическое значение обозначающее наличие флага "g", доступно только для чтения
  multiline - логическое значение обозначающее наличие флага "m", доступно только для чтения
  lastIndex - счетчик, указывающий, с какой позиции в строке начинать поиск

- **Методов** у объекта **RegExp** всего два :
- **exec**(*text*) - выполнение поиска в строке, указанной в качестве параметра, возвращает массив найденных соответствий.
  test(*text*) - проверка соответствия регулярному выражению, возвращает true\false.
- **Метод exec**() выполняет регулярное выражение по отношению к строке-аргументу, результатом его работы является массив, в который попадают соответствия. Если соответствий не найдено, то результатом будет null. А если соответствие есть, то оно попадает в массив нулевым элементом, при этом свойство lastIndex объекта сместится на позицию, следующую непосредственно за найденной подстрокой. Давайте рассмотрим на примере :

```
ar myString = "This is just a test text";  // Задаем строку для поиска
var myPattern = /te|is/g;                  // Задаем шаблон - либо "te" либо "is"

result = myPattern.exec(myString);

//result будет равен "is" - первому совпадению шаблона, свойство lastIndex примет значение 4

result = myPattern.exec(myString); //result == "is" - второму совпадению шаблона, lastIndex == 7
result = myPattern.exec(myString); //result == "te" - третьему совпадению шаблона, lastIndex == 17
result = myPattern.exec(myString); //result == "te" - четвертому совпадению шаблона, lastIndex == 22
```

В этом примере мы четыре раза подряд вызываем метод exec(), каждый раз он сдвигает указатель начала поиска на позицию, следующую за найденным совпадением и присваивает переменной result само найденное совпадение.

Метод test() выполняет регулярное выражение по отношению к строке-аргументу, результатом его работы является логическое значение - true если совпадение есть, и false если нет. Свойство lastIndex объекта также как и у метода exec() сместится на позицию, следующую непосредственно за найденной подстрокой.

Важный момент! Если совпадение не найдено, то lastIndex будет смещен на позицию 0 и поиск можно будет начинать сначала.

Давайте рассмотрим на таком же примере :

```
var myString = "This is just a test text";  // Задаем строку для поиска
var myPattern = /te|is/g;                   // Задаем шаблон - либо "te" либо "is"

result = myPattern.test(myString);

// result будет равен true, поскольку будет найдено первое совпадение, свойство lastIndex примет значение 4

result = myPattern.test(myString); // result == true, lastIndex == 7
result = myPattern.test(myString); // result == true, lastIndex == 17
result = myPattern.test(myString); // result == true, lastIndex == 22
result = myPattern.test(myString); // result == false, lastIndex == 0
```

- В этом примере мы четыре раза подряд вызываем метод test(), каждый раз он сдвигает указатель начала поиска на позицию, следующую за найденным совпадением и присваивает переменной result булево значение - true если совпадение найдено и false - если нет. В последнем запуске совпадение не найдено, поэтому указатель lastIndex получает значение 0

Давайте проверим, что мы усвоили про объект RegExp

- Конструктор объекта RegExp может принимать на вход либо один либо два аргумента
- Если совпадений не найдено, метод exec() возвращает "Undefined«
- Объект RegExp имеет пять свойств
- Конструктор объекта RegExp превращает регулярное выражение в строку
- Метод test() смещает указатель точки

первым параметром передается случайная строка(переменная s), а вторым - случайная подстрока(переменная sub_s), которую нужно использовать в качестве шаблона регулярного выражения. Вам нужно вернуть из функции строку, в которой будут перечислены через запятую все совпадения шаблона с первой строкой.

```
function testRegExp(s, sub_s) {
//Тут написать своё решение
}
```

Решение

- В данной теме мы разберем основы такого понятия как **замыкания**.

- Если в двух словах - **замыкание** это такая функция, которая была объявлена внутри другой функции.
- Есть еще одно условие - эта функция должна иметь доступ к переменным функции, внутри которой она была объявлена.
-

- Таким образом такая функция (**замыкание**) **имеет доступ к данным внутри себя и внутри родительской функции**.
- Также внутренняя функция может обращаться не только к переменным, но и к входным параметрам своей внешней функции.

Замыкания

- Давайте рассмотрим практический пример.

```
function greetPirate(pirateName) {          // Объявление родительской функции
    var greeting = "Hello ";
    function checkCaptain() {               // Объявление замыкания
      if (pirateName != "Jack Sparrow")
        return greeting + pirateName;
      else
        return greeting + "CAPTAIN " + pirateName + "!";
    }
    return checkCaptain();
  }
```

В этом примере мы написали функцию, которая выдает приветствие пирату - добавляет "Hello " к имени, которое подается на вход. Внутри этой функции мы создали замыкание (checkCaptain), которое проверяет входной параметр родительской функции - если имя пирата - "Jack Sparrow", то замыкание добавляет к имени слово "CAPTAIN" и для капитана Джека Воробья выводит персональное приветствие "Hello CAPTAIN Jack Sparrow!".

В результате при вызове мы получим следующий результат:

```
console.log(greetPirate("Mad Dog"));          // Выведет в консоль "Hello Mad Dog«
console.log(greetPirate("Jack Sparrow"));     // Выведет в консоль "Hello CAPTAIN Jack Sparrow!"
```

- У замыканий есть два важных свойства.
- 1. Замыкание может обращаться к переменным своей внешней функции даже после ее окончания выполнения.
- На практике это означает что даже после того как выполнение внешней функции завершено, внутренняя все еще может быть вызвана и имеет доступ к переменным внешней функции.
- Давайте рассмотрим пример:

```
function pirate() {
    var pirateName = "noname";
    return {
        getName: function() {
            return pirateName;
        },
        setName: function(newName) {
            pirateName = newName;
        }
    }
}
```

- Мы описали функцию с двумя замыканиями: одно возвращает значение переменной из вызывающей функции, второе - изменяет его. Посмотрим, что получится при практическом использовании:

- var newPirate = pirate();
- console.log(newPirate.getName()); // Выводим текущее содержимое переменной - там изначальный "noname"
- newPirate.setName("Jack Sparrow"); //Изменяем значение переменной на "Jack Sparrow"
- console.log(newPirate.getName()); //Выводим текущее содержимое переменной - получаем "Jack Sparrow"

- В данном примере мы видим что замыкания получили доступ к переменной внешней функции после ее завершения.

- 2. Замыкания хранят не содержимое переменных внешней функции, а ссылки на эти переменные.
- Давайте в этом контексте рассмотрим классический пример замыкания, описываемый в большинстве источников - счетчик.

```
function makeCounter(initialValue) {
  var currentState = initialValue;
  return function () {
    currentState = currentState + 1;
    return currentState;
  }
}
```

- В данном случае мы описали функцию, внутри которой находится замыкание, увеличивающее каждый раз счетчик и возвращающее его.
- Давайте посмотрим, что получится если вызывать ее много раз пордряд.

```
var myCounter = makeCounter(5);  // Создаем экземпляр счетчика и устанавливаем его начальное значение = 5
console.log(myCounter());       // В консоль будет выведено значение 6
console.log(myCounter());       // В консоль будет выведено значение 7
console.log(myCounter());       // В консоль будет выведено значение 8
```

Для начала - небольшой тест!

- Если внутри обычной функции, вложенной в другую функцию, сделать обращение к выходному параметру внешней функции, то она будет называться замыканием
- Замыкание хранит ссылку на переменные, находящиеся в функции, внешней по отношению к ней
- После окончания выполнения функции замыкание теряет доступ к её данным
- Замыкание хранит фактическое значение переменных, полученных из функции, в которую оно вложено
- Если внутри обычной функции, вложенной в другую функцию, сделать обращение к переменной, находящейся во внешней функции, то она будет называться замыканием

- А теперь - задача на... программирование!)
  В тексте внизу описана анонимная функция, внутри которой находится замыкание, изменяющее значение переменной во внешней функции и возвращающее ее. Но вот проблема - строчки перепутались местами! Восстановите необходимую последовательность строк

- }
  return actual;
  var actual = 0;
  return function() {
  var enumerator = (function() {
  actual ++;
  })();

Решение

- ```
  var enumerator = (function() {
  var actual = 0;
  return function() {
  actual ++;
  return actual;
  }
  })();
  ```
  ```

- В этом курсе, после небольшого исторического обзора, мы рассмотрели самые базовые понятия языка программирования **JavaScript**:
- Понятия **переменных** и простейшие **операции** над ними.
- Основные приемы влияния на ход работы программы - **ветвление и циклы.**
- Узнали глобальные понятия **объекта** и **функции**.
- Подробно рассмотрели базовые классы, позволяющие работать со **строками, массивами, датой и временем**.
- Выяснили какие математические функции поддерживает объект **Math** и попробовали с ними работать.
- Научились обрабатывать **ошибки**.
- Изучили что представляют из себя **регулярные выражения** и как ими пользоваться.