Интернет-технологии и распределённая обработка данных

Лекция 15

Объекты JavaScript (1)

1. Глобальный объект
2. Числа
3. Объект Boolean
4. Работа с датами
5. Строки и регулярные выражения
6. Объекты ошибок

Глобальный объект

Глобальный объект (далее для краткости Global) – специальный объект JavaScript, автоматически создаваемый при запуске транслятора (перед началом выполнения первого скрипта).

Global как глобальный контекст

- Инструкция объявления глобальной переменной или инструкция объявления функции = создание одноимённого **неудаляемого** свойства у Global;

  Инициализация необъявленной переменной = создание одноимённого **удаляемого** свойства у Global;

  Свойства и методы Global доступны из любого места скрипта.

Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами `window`.

```
var a = 5;
// объявление var создаёт свойство window.a
alert( window.a ); // 5

window.a = 5;
alert( a ); // 5
```

Global и фазы выполнения скрипта

0. Создание Global.

1. Инициализация. Скрипт сканируется в поисках глобальных инструкций `function`, а затем – в поисках глобальных инструкций `var`. Каждое объявление добавляется к Global. При этом функции готовы к работе, а вот переменные равны `undefined`.

2. Выполнение инструкций скрипта (в частности, присваивание объявленным переменным значений).

Шаги 1 и 2 выполняются для каждого скрипта на веб-странице.

Свойства-значения Global

1. `Infinity` – числовое значение, представляющее бесконечность;
2. `NaN` – числовое значение, представляющее не-число (Not-a-Number).
3. `undefined` – хранит значение undefined.

В стандарте ECMAScript все эти свойства описаны как доступные только для чтения.

Свойства-функции Global

`eval(string)`

Если аргумент не является строкой, он возвращается. Иначе происходит трансляции и выполнение строки как некоего JavaScript-кода. Результат работы – это результат последнего вычисленного выражения.

```
var y = eval("function f(x) { return x + 1; }");

alert(y);        // undefined
alert(f(10));    // 11
//лучше использовать new Function('a,b', '..тело..');
```

isFinite(number)

Возвращает `false`, если аргумент может быть приведён к NaN, $+\infty$ или $-\infty$, в противном случае возвращает `true`.

isNaN(number)

Возвращает `true`, если аргумент может быть приведён к NaN, в противном случае возвращает `false`.

Пустая строка и строка из пробельных символов преобразуются к числу 0, false, true, null – числа

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

parseInt(string, radix)

Преобразование (гибкое!) строки в целое число. Второй аргумент определяет систему счисления (целое число от 2 до 36 (включительно)).

```
var x = parseInt("123");         // 123
var y = parseInt("123.7");       // 123
(отбрасывает дробную)
var z = parseInt("0x123");       // 291
var a = parseInt("123", 5);      // 38
var b = parseInt("  123Tom");   // 123
var c = parseInt("Tom");        // NaN
```

parseFloat(string)

Преобразование строки в число.

```
var a = parseFloat("123.4");        // 123.4
var b = parseFloat("0x123");        // 0
var c = parseFloat("  123e2Cat");  // 12300
var d = parseFloat("Dog");         // NaN
```

toString(основание системы)
Преобразование числа в строку.

```
var n = 255;
alert( n.toString(16) ); // ff – цвет.знач. в браузере

var n = 4;
alert( n.toString(2) ); // 100 – отладка битовых операц

var n = 1234567890;
alert( n.toString(36) ); // kf12oi – 26 букв, 10 цифр
//«кодировать» число в виде буквенно-цифровой строки,
«укоротить» длинный цифровой идентификатор, например
чтобы выдать его в качестве URL
```

```
decodeURI(encodedUri)
decodeURIComponent(encodedUriComponent)
encodeURI(uri)
encodeURIComponent(uriComponent)
```

Набор функций для обработки URI (замена в строке «плохих» для URI символов на «хорошие»):

```
var uri = "my test.asp?name=ståle&car=saab";
var res = encodeURI(uri);
// res = my%20test.asp?name=st%C3%A5le&car=saab
```

```
decodeURI("https://developer.mozilla.org/ru/docs/JavaScript_%D1%88%D0%B5%D0%BB%D0%BB%D1%8B");
// "https://developer.mozilla.org/ru/docs/JavaScript_шеллы"
```

Свойства-конструкторы Global

Глобальный объект содержит набор методов:

```
Object()
Function()
Array()
String()
Boolean()
Number()
Date()
RegExp()
```

```
Error()
EvalError()
RangeError()
ReferenceError()
SyntaxError()
TypeError()
URIError()
```

Свойства-объекты Global

`Math` – этот объект содержит математические константы и функции. Рассмотрим его подробнее позже.

`JSON` – объект для работы с JSON . Содержит два метода:

- `parse()` для преобразования JSON-строки в допустимые значения JavaScript;
- `stringify()` для получения JSON-строки по значению.

```
var person = {name: "John", age: 18};
var str = JSON.stringify(person);
alert(str); // { "name": "John", "age": 18 }
var newPerson = JSON.parse(str);
```

Реализации Global

Движок может дополнять Global новыми свойствами.

Например, движки в браузерах:

- снабжают Global свойством `window`, содержащим Global
- добавляют к Global свойство `document` для доступа к документу, отображаемому в окне браузера
- добавляют к Global методы `alert()`, `prompt()`, `confirm()`; `setInterval()`, `setTimeout()`, `clearInterval()`, `clearTimeout()`

Реализации Global

- `alert(msg)` – выводит модальное окно с сообщением
- `prompt(msg, txt)` – показывает окно с текстом `msg` для ввода строки (в поле ввода отображается необязательный параметр `txt`). Возвращает введённую строку (или `null` – если пользователь нажал Cancel или закрыл окно)
- `confirm(msg)` – выводит окно с сообщением и кнопками OK и Cancel. Возвращает `true`, если пользователь нажал OK; `false`, если пользователь нажал Cancel или закрыл окно

Объекты-обёртки

Для значений `number`, `boolean`, `string` существуют объекты-обёртки. Они нужны, когда работа со значением происходит как с объектом (например, если у значения вызывается метод).

Для получения обёрток используются функции `Number()`, `Boolean()`, `String()`, вызываемые как конструкторы.

Прямой вызов этих конструкторов не приветствуется – нужно полагаться на автоматическое приведение типов.

Объект-обёртка Number

Чтобы получить объектную обёртку над типом `number`, необходимо вызвать функцию `Number()` как конструктор.

```
var obj = new Number(10);
```

Вызов `Number()` как функции можно использовать для конвертации произвольного значения в число:

```
var x = Number("10"); // typeof x == "number"
```

Методы объекта Number

Методы объекта `Number` служат для получения строкового представления числа:

- `toExponential()` – в экспоненциальной форме
- `toString()` – в обычном виде (можно указать систему счисления)
- `toPrecision()` – с указанным общим количеством цифр
- `toFixed()` – с заданным количеством цифр после точки

Методы объекта Number – пример

```
var x = 123.125;
alert(x.toExponential());        // "1.23125e+2"
alert(x.toExponential(2));       // "1.23e+2"
alert(x.toString(8));            // "173.1"
alert(x.toPrecision(4));         // "123.1"
alert(x.toPrecision(2));         // "1.2e+2"
alert(x.toFixed(2));             // "123.13"
```

Статические свойства Number

- MAX_VALUE – наибольшее положительное число.

MIN_VALUE – самое близкое к нулю положительное число

NaN – «не-число».

NEGATIVE_INFINITY – это $-\infty$

POSITIVE_INFINITY – это $+\infty$

Объект Math

`Math` является встроенным объектом, хранящим в свойствах различные математические **константы**, а в методах – математические **функции**.

Несмотря на название, `Math` не является функцией-конструктором.

Свойства Math

Свойства `Math` представляют различные математические константы. И изменить их нельзя!

Свойство	Значение (приблиз.)	Свойство	Значение (приблиз.)
E		LOG10E	
LN2		PI	
LN10		SQRT1_2	
LOG2E		SQRT2	

Методы Math

Метод	Описание	Метод	Описание
`abs(x)`	Абсолютное значение x	`log(x)`	
`acos(x)`	Арккосинус x	`max(x,y,...)`	Максимум из аргументов
`asin(x)`	Арксинус x	`min(x,y,...)`	Минимум из аргументов
`atan(x)`	Арктангенс x	`pow(x,y)`	
`atan2(x,y)`	Арктангенс x/y	`random()`	Возвращает псевдослучайное число в диапазоне [0, 1)
`ceil(x)`	Наименьшее целое >= x	`round(x)`	Округление до ближайшего целого
`cos(x)`	Косинус x	`sin(x)`	синус числа
`exp(x)`		`sqrt(x)`	

Объект Math – пример

```
var data = [];
for (var i = 0; i < 10; i++) {
    // случайные числа в диапазоне [-50, 50]
    data[i] = Math.random() * 100 - 50;
}

// как apply() помогает найти максимум в массиве
var max = Math.max.apply(Math, data);
```

Объект-обёртка Boolean

Работа с функцией `Boolean()` похожа на работу с функцией `Number()` (вызов и как конструктора, и как функции, автоматическое «оборачивание»).

```
var a = new Boolean("1"); // оборачивает true
var b = new Boolean();    // оборачивает false
var c = Boolean("0");     // конвертирует в true
```

У объекта-обёртки `Boolean` нет полезных свойств или методов.

Объект Date

Конструктор `Date` порождает объект, представляющий дату и время (хранится как количество миллисекунд с 01.01.1970 00:00:00 в часовом поясе UTC).

- Без параметров – текущая
- Один числовой – миллисекунды от 01.01.1970
- Один строковый – разбор строки как даты
- От двух до семи числовых – значения года, месяца, дня, часов, минут, секунд, миллисекунд

- Создает объект Date с текущей датой и временем:

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

Thu Apr 06 2017 23:30:39 GMT+0300 (Беларусь (зима))

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

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

Fri Jan 02 1970 02:00:00 GMT+0200 (Беларусь (зима))

new Date(datestring)

Если единственный аргумент – строка, используется вызов Date.parse (см. далее) для чтения даты из неё.

Дату можно создать, используя компоненты в местной временной зоне.

new Date(year, month, date, hours, minutes, seconds, ms)

- обязательны только первые два аргумента (year из 4 цифр)
- month начинается с нуля 0.

Отсутствующие, начиная с hours = нулю, а date – единице.

```
//1 января 2011, 00:00:00

new Date(2011, 0, 1, 0, 0, 0, 0);

// то же самое, часы/секунды по умолчанию равны 0

new Date(2011, 0, 1);
```

с точностью до миллисекунд:

```
var date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Получение компонентов даты

для местной временной зоны

- **`getFullYear()`** Получить год(из 4 цифр)
- **`getMonth()`** Получить месяц, **от 0 до 11**.
- **`getDate()`** Получить число месяца, от 1 до 31.
- **`getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()`**
- **`getDay()`** Получить номер дня в неделе. **от 0(воскресенье) до 6(суббота)**.

для зоны GMT+0 (UTC)

getUTCFullYear(), getUTCMonth(), getUTCDay().

```
// текущая дата
var date = new Date();
// час в текущей временной зоне
alert( date.getHours() );
// сколько сейчас времени в Лондоне?
// час в зоне GMT+0
alert( date.getUTCHours() );
```

методы без UTC-варианта:

- **`getTime()`** Возвращает число миллисекунд, прошедших с 1 января 1970 года GMT+0, то есть того же вида, который используется в конструкторе `new Date(milliseconds)`.
- **`getTimezoneOffset()`** Возвращает разницу между местным и UTC-временем, в минутах.
- `alert( new Date().getTimezoneOffset() );`

```
// Для GMT-1 выведет 60
```

Установка компонентов даты

- setFullYear(year [, month, date])
- setMonth(month [, date])
- setDate(date)
- setHours(hour [, min, sec, ms])
- setMinutes(min [, sec, ms])
- setSeconds(sec [, ms])
- setMilliseconds(ms)
- setTime(milliseconds) //всю дату по миллисекундам с 01.01.1970 UTC

```
var today = new Date;
today.setHours(0);
alert( today ); // сегодня, но час изменён на 0
today.setHours(0, 0, 0, 0);
alert( today ); // сегодня, ровно 00:00:00.
```

Автоисправление даты

```
var d = new Date(2013, 0, 32); //32 января 2013 ?!?
alert(d); // ... это 1 февраля 2013!
```

Fri Feb 01 2013 00:00:00 GMT+0300 (Беларусь (зима))

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

Wed Mar 02 2011 00:00:00 GMT+0200 (Беларусь (зима))

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

var d = new Date;
d.setDate(1); // поставить первое число месяца
alert( d );
d.setDate(0); // нулевого числа нет, будет последнее
число предыдущего месяца
alert( d );
var d = new Date;
d.setDate(-1); // предпоследнее число предыдущего
месяца
alert( d );
```

Преобразование к числу, разность дат

```
alert(+new Date)
// +date то же самое, что: +date.valueOf()
//миллисекунды!
var start = new Date; // засекли время
// что-то сделать
for (var i = 0; i < 100000; i++) {
  var doSomething = i * i * i;
}var end = new Date; // конец измерения
alert( "Цикл занял " + (end - start) + " ms" );
```

Бенчмаркинг

```
var arr = [];
for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {
 for (var key in arr) arr[key]++;}

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

```
function bench(f) {
var date = new Date();
for (var i = 0; i < 1000; i++) f(arr);
return new Date() - date;}
// bench для каждого теста запустим много раз, чередуя
var timeIn = 0,  timeLength = 0;
for (var i = 0; i < 100; i++) {
timeIn += bench(walkIn);
timeLength += bench(walkLength);}
alert( 'Время walkIn: ' + timeIn + 'мс' );
alert( 'Время walkLength: ' + timeLength + 'мс' );
```

```
alert( performance.now());
```

возвращает количество миллисекунд с начала загрузки страницы (с момента выгрузки предыдущей страницы из памяти)

- время включает в себя всё, включая начальное обращение к серверу.
- Его можно посмотреть в любом месте страницы, чтобы узнать, сколько времени потребовалось браузеру, чтобы до него добраться, включая загрузку HTML.

Форматирование и вывод дат

```
date.toLocaleString(локаль, опции)
var date = new Date(2014, 11, 31, 12, 30, 0);
var options = {  era: 'long',  year: 'numeric',
month: 'long',  day: 'numeric',  weekday: 'long',
timezone: 'UTC',  hour: 'numeric',  minute:
'numeric',  second: 'numeric'};
alert( date.toLocaleString("ru", options) );
// среда, 31 декабря 2014 г. н.э. 12:30:00
alert( date.toLocaleString("en-US", options) );
// Wednesday, December 31, 2014 Anno Domini 12:30:00
PM
```

Методы вывода без локализации

toString(), toDateString(), toTimeString()

```
var d = new Date();
alert( d.toString() );
```

Fri Apr 07 2017 11:17:06 GMT+0300 (Беларусь (зима))

toUTCString() Fri, 07 Apr 2017 08:15:47 GMT

toISOString() //дата в формате ISO

2017-04-07T08:18:06.981Z

Разбор строки, Date.parse

формат ISO 8601 `YYYY-MM-DDTHH:mm:ss.sssZ`

- `YYYY-MM-DD` – дата в формате год-месяц-день.
- Обычный символ T используется как разделитель.
- `HH:mm:ss.sss` – время: часы-минуты-секунды-миллисекунды.
- Часть `'Z'` обозначает временную зону – в формате `+-hh:mm`, либо символ Z, обозначающий UTC.

2017-04-07T08:18:06.981Z

`Date.parse(str)` разбирает строку `str` в таком формате, возвращает миллисекунды или `NaN`.

```
var msUTC = Date.parse('2012-01-26T13:51:50.417Z');
// зона UTC

alert( msUTC ); //1327571510417 (число миллисекунд)
```

С таймзоной `-07:00 GMT`:

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

```
//работает везде
var ms = Date.parse("January 26, 2011 13:51:50");
alert( ms );
```

```
Date.now() //возвращает дату сразу в виде миллисекунд
//не создаёт промежуточный объект даты
```

У объекта `Date` более сорока методов. Можно посмотреть здесь:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/prototype#Methods

Две трети методов – для получения или установки отдельного компонента даты, одна треть – для конвертирования даты в строку.

Объект-обёртка String

Напоминание: тип `string` в JavaScript хранит **неизменяемые** строки в UTF-16, является примитивным.

Для `string` существует объект-обёртка. Работа с функцией `String()` похожа на работу с функцией `Number()` (вызов и как конструктора и как функции, автоматическое «оборачивание»).

У объекта `String` есть свойство `length` (длина строки) и индексатор (введён в ECMAScript 5) для обращения к отдельным символам:

```
var s = new String("Tom");
for(var i = 0; i < s.length; i++)
  alert(s[i]);
```

Также можно использовать метод `charAt(позиция)`

`charAt(n)` – возвращает символ в позиции n

`charCodeAt(n)` – возвращает код символа в позиции n

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

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

Методы объекта String

`toLowerCase()` – к нижнему регистру

`toUpperCase()` – к верхнему регистру

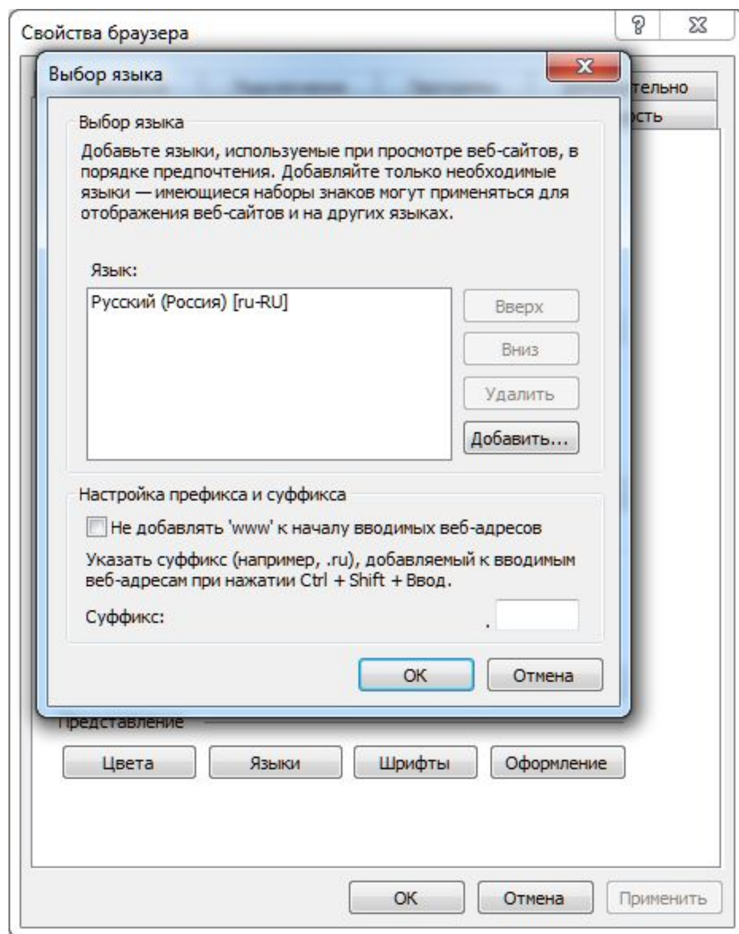
`toLocaleLowerCase()` – к нижнему с учётом локали

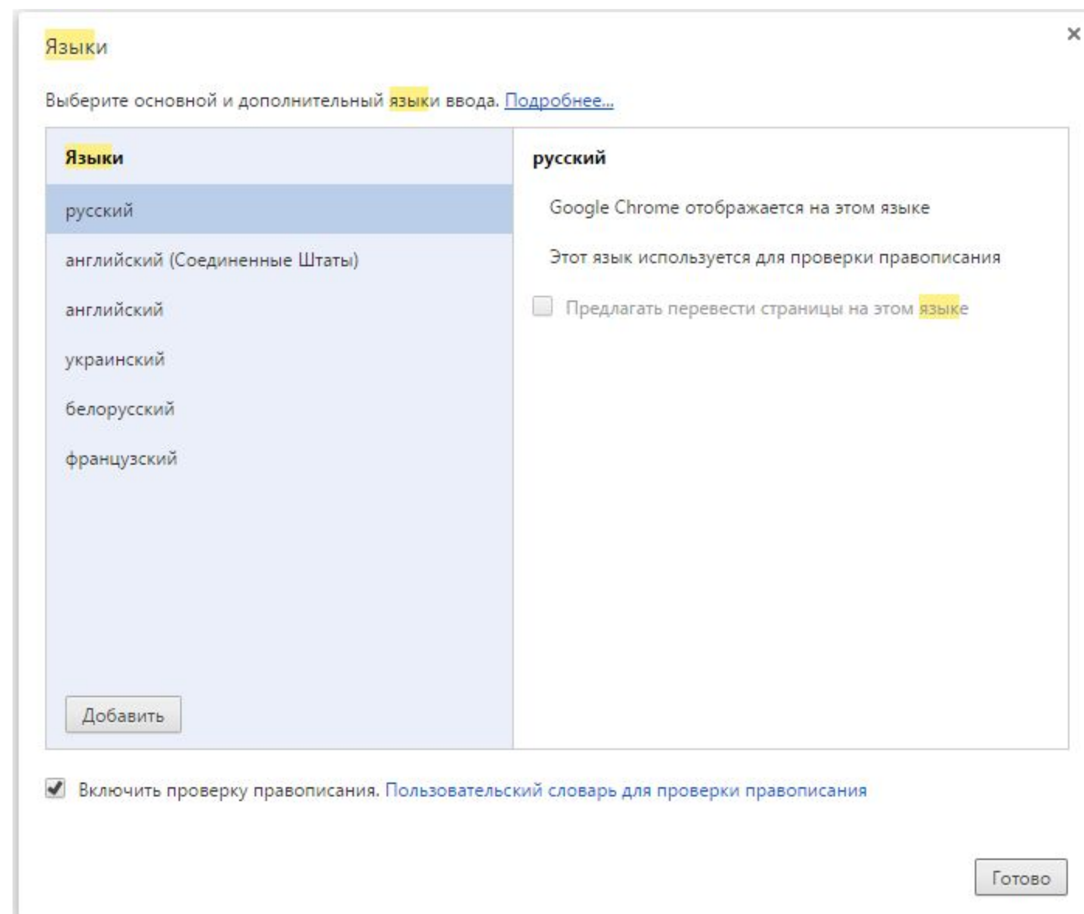
`toLocaleUpperCase()` – к верхнему с учётом локали

`toString()` – возвращает примитивную строку

```
var str = "Cat";
alert(str.toLowerCase());          // cat
alert(str.toLocaleLowerCase());    // cat
alert(str.toUpperCase());          // CAT
alert(str[0].toLowerCase());          // c
```

Установка локали в браузере





`indexOf(substr [,from])` – поиск подстроки от начала (или от указанной позиции `from`) к концу строки

`lastIndexOf(substr [,from])` – поиск подстроки от конца (или от указанной позиции `from`) к началу строки

возвращает позицию, на которой
находится подстрока или `-1`, если ничего не найдено

```
 var str = "Widget with id";
alert( str.indexOf("Widget") ); ,
// 0, т.к. "Widget" найден прямо в начале str
alert( str.indexOf("id") );
// 1, т.к. "id" найден, начиная с позиции 1
alert( str.indexOf("widget") );
// -1, не найдено, так как поиск учитывает регистр

var str = "Widget with id";
alert(str.indexOf("id", 2))
// 12, поиск начат с позиции 2
```

```
var str = "Ослик Иа-Иа посмотрел на виадук";
// ищем в этой строке
var target = "Иа"; // цель поиска
var pos = 0;
while (true) { //все вхождения подстроки
  var foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;
   alert( foundPos ); // нашли на этой позиции
   pos = foundPos + 1;
// продолжить поиск со следующей
}
```

`concat(s,...)` – соединяет строки (текущую и аргументы)

`split([separator] [,limit])` – разбивает строку на части по сепаратору (можно ограничить число частей)

`substring(start [,end])` – возвращает подстроку, начиная с позиции `start` до, но не включая, `end`

`substr(start [, length]) - start` –такой же смысл, как и в `substring`, `length -` количество символов.

`slice(start [,end])` – аналог `substring()`, но при отрицательном `end` позиция считается от конца строки

`trim()` – удаление начальных и концевых пробельных символов

```
alert("A".concat(" B", " C"));  // "A B C"

arr = "a,b,c".split(',');          // ["a", "b", "c"]
arr = "a,b,c".split(); // ["a,b,c"]-один элемент со всей строкой
arr = "a,b,c".split(',', 2);      // ["a", "b"]
```

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

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

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

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

`slice`

Отрицательные значения отсчитываются от конца строки:

```
alert( "testme".slice(-2) );
// "me", от 2 позиции с конца
alert( "testme".slice(1, -1) );
// "estm", от 1 позиции до первой с конца.
```

Метод String.fromCharCode()

Этот статический метод конструирует и возвращает примитивную строку (не объект `String`) по заданным числовым кодам символов:

```
var s = String.fromCharCode(65, 66, 67); // "ABC"
```

Регулярные выражения

Объект для работы с регулярным выражением, можно получить, вызвав конструктор `RegExp()` или использовав литерал регулярного выражения:

```
// полная форма
var expr = new RegExp("pattern", "flags");

// сокращенная форма
var expr = /pattern/flags;
```

Регулярные выражения

Возможные флаги:

`g` поиск всех совпадений (а не первого);

`i` игнорирование регистра;

`m` символы начала и конца (`^` и `$`) начинают работать отдельно для каждой визуальной строки (`\n`).

Свойства RegEx

`global`, `ignoreCase`, `multiline` – эти свойства равны `true`, если установлены соответствующие флаги регулярного выражения;

`source` – регулярное выражение в виде строки;

`lastIndex` – позиция в строке, соответствующая следующему совпадению с регулярным выражением (это значение первоначально всегда равно нулю).

Методы RegEx

`test(str)` – выясняет, есть ли в строке `str` совпадения для регулярного выражения и возвращает `true` или `false`

`exec(str)` – возвращает массив подстрок, удовлетворяющих регулярному выражению (или `null`)

Если регулярное использует флаг **g**, можно использовать метод `exec()` несколько раз для нахождения всех сопоставлений в строке.

Метод `test` проверяет, есть ли хоть одно совпадение в строке `str`. Возвращает `true/false`.

Работает, по сути, так же, как и проверка

`str.search(reg) != -1`, например:

```
var str = "Люблю регэкспы я, но странною любовью";
// эти две проверки идентичны
alert( /лю/i.test(str) ) // true
alert( str.search(/лю/i) != -1 ) // true
```

Пример с отрицательным результатом:

```
var str = "Ой, цветёт калина...";
alert( /javascript/i.test(str) ) // false
alert( str.search(/javascript/i) != -1 ) // false
```

Метод `regexp.exec` позволяет искать и все совпадения и скобочные группы в них.

Он ведёт себя по-разному, в зависимости от того, есть ли у регэкспа флаг `g`.

Если флага `g` нет, то `regexp.exec(str)` ищет и возвращает первое совпадение, является полным аналогом вызова `str.match(reg)`.

Если флаг `g` есть, то вызов `regexp.exec` возвращает первое совпадение и *запоминает* его позицию в свойстве `regexp.lastIndex`. Последующий поиск он начнёт уже с этой позиции. Если совпадений не найдено, то сбрасывает `regexp.lastIndex` в ноль.

поиск всех совпадений в цикле:

```
var str = 'Многое по JavaScript можно найти на сайте http://javascript.ru';

var regexp = /javascript/ig;

var result;

alert( "Начальное значение lastIndex: " + regexp.lastIndex );

while (result = regexp.exec(str)) {

alert( 'Найдено: ' + result[0] + ' на позиции:' + result.index );

alert( 'Свойство lastIndex: ' + regexp.lastIndex );}

alert( 'Конечное значение lastIndex: ' + regexp.lastIndex );
```

Здесь цикл продолжается до тех пор, пока `regexp.exec` не вернёт `null`, что означает «совпадений больше нет».

`regexp.exec` ищет сразу с нужной позиции, если поставить `lastIndex` вручную:

```
var str = 'Многое по JavaScript можно найти на сайте http://javascript.ru';
var regexp = /javascript/ig;
regexp.lastIndex = 40;
alert( regexp.exec(str).index );
// 49, поиск начат с 40-й позиции
```

search(regexp) – позиция первой подстроки, удовлетворяющей заданному регулярному выражению

match(regexp) – возвращает массив подстрок, удовлетворяющих регулярному выражению (или null)

replace(regexp, newSubStr) – поиск и замена подстрок, удовлетворяющих регулярному выражению

str.search(reg)

возвращает позицию первого совпадения или `-1`, если ничего не найдено.

```
var str = "Люблю регэкспы я, но странною любовью";

alert( str.search( /лю/i ) ); // 0
```

Ограничение метода `search` – он всегда ищет только первое совпадение.

Метод `str.match` работает по-разному, в зависимости от наличия или отсутствия флага g

```
var str = "ОЙ-Ой-ой";
var result = str.match( /ой/i );
alert( result[0] ); // ОЙ  (совпадение)
alert( result.index ); // 0 (позиция)
alert( result.input ); // ОЙ-Ой-ой (вся поисковая строка)
```

Если часть шаблона обозначена скобками, то она станет отдельным элементом массива.

```
var str = "javascript - это такой язык";
var result = str.match( /JAVA(SCRIPT)/i );
alert( result[0] );
// javascript (всё совпадение полностью)
alert( result[1] );
 // script (часть совпадения, соответствующая скобкам)
alert( result.index ); // 0
alert( result.input ); // javascript - это такой язык
```

```
var str = "The rain in SPAIN stays mainly in the plain";
var res = str.match(/ain/g); // ["ain", "ain", "ain"]

res = str.match(/ain/); // ["ain"]
res = str.match(/ain/gi); // ["ain", "AIN", "ain", "ain"]

str = "Прочтите: Глава 3.4.5.1";
res = str.match(/Глава (\d+(\.\d)*)/)
// ["Глава 3.4.5.1", "3.4.5.1", ".1"]
// скобки в регулярном выражении – это группы захвата
```

Обычно мы используем метод `split` со строками, вот так:

```
alert('12-34-56'.split('-')) // [12, 34, 56]
```

Можно передать в него и регулярное выражение, тогда он разобьёт строку по всем совпадениям.

Тот же пример с регэкспом:

```
alert('12-34-56'.split(/-/)) // [12, 34, 56]
```

STR.REPLACE(REG, STR|FUNC)

Швейцарский нож для работы со строками, поиска и замены любого уровня сложности.

Его простейшее применение – поиск и замена подстроки в строке, вот так:

```
// заменить дефис на двоеточие
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

При вызове со строкой замены `replace` всегда заменяет только первое совпадение.

Чтобы заменить *все* совпадения, нужно использовать для поиска не строку `"-"`, а регулярное выражение `/-/g`, причём обязательно с флагом `g`:

```
// заменить дефис на двоеточие
alert( '12-34-56'.replace( /-/g, ":" ) )
 // 12:34:56
```

В строке для замены можно использовать специальные символы:

Спецсимволы	**Действие в строке замены**
$$	Вставляет "$".
$&	Вставляет всё найденное совпадение.
$`	Вставляет часть строки до совпадения.
$'	Вставляет часть строки после совпадения.
$*n*	где n -- цифра или двузначное число, обозначает n-ю по счёту скобку, если считать слева-направо.

Пример использования скобок и $1, $2:

```
var str = "Василий Пупкин";
alert(str.replace(/(Василий) (Пупкин)/, '$2, $1'))
// Пупкин, Василий
```

Ещё пример, с использованием $&:

```
var str = "Василий Пупкин";
alert(str.replace(/Василий Пупкин/, 'Великий $&!'))
// Великий Василий Пупкин!
```

Объекты ошибок

Для описания исключительных ситуаций *возможно* использование объектов, порождённых стандартными функциями-конструкторами:

```
Error()      EvalError()
RangeError()     ReferenceError()
SyntaxError()    TypeError()
URIError()
```

Объект Error

Конструктор `Error()` порождает базовый объект описания ошибки. Он имеет необязательный параметр для указания текстового сообщения об ошибке.

```
if (x < 0)
    throw new Error("Что-то пошло не так");
```

Сам объект ошибки имеет строковое свойство `name`, равное `"Error"`, и строковое свойство `message`.

Объект Error

Некоторые движки дополняют объект ошибки нестандартными свойствами, а конструктор `Error()` – дополнительными опциональными параметрами.

Mozilla:

`fileName` путь к файлу, в котором возникла эта ошибка

`lineNumber` номер строки в файле, в котором возникла ошибка

`columnNumber` номер колонки, на которой возникла ошибка

`stack` стек вызовов

Другие стандартные ошибки

`SyntaxError` – ошибка синтаксического разбора

`TypeError` – переменная или параметр неправильного типа

`RangeError` – значение выходит за пределы диапазона

`ReferenceError` – попытка обратиться к переменной, которая не была объявлена

`URIError` – функции `encodeURI()` или `decodeURI()` были вызваны с неправильными аргументами

`EvalError` – ошибка при выполнении `eval()` (устарело)