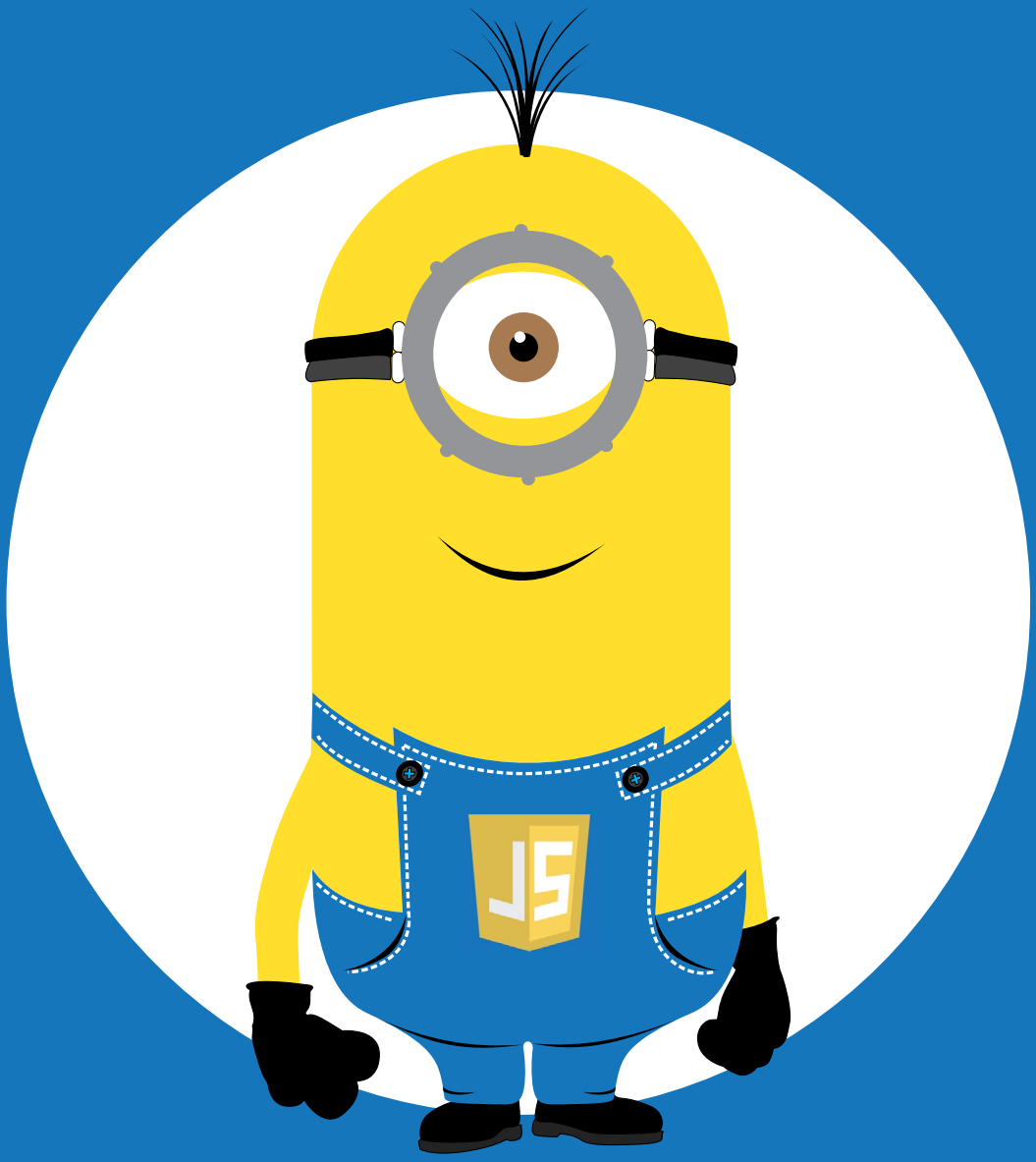

Основы Javascript #1

LoftSchool
от мыслителя к создателю

Содержание

1

Подключение JavaScript

В веб-разработке принято следующее «разделение ролей»: HTML отвечает за структуру документа, стили — за его внешний вид, а скрипты — за поведение. С помощью скриптов, например, можно «оживлять» страницу, добавляя анимацию и другие эффекты, которые создаются с помощью языка JavaScript.

JavaScript — объектно-ориентированный язык программирования, в первую очередь предназначенный для создания интерактивных (т. е. изменяющихся без перезагрузки) веб-страниц.

Стандартизированная версия JavaScript, называемая **ECMAScript**, работает одинаково во всех приложениях, поддерживающих этот стандарт. Мы будем рассматривать стандарт **ECMA-262**, если не оговариваем особо, что рассматривается расширение, которое появилось как часть предложения **Harmony** (следующий вариант стандарта, известный как **ECMAScript 2015**)

Скрипты подключаются так же, как и стили: их или пишут внутри страницы, или подключают как внешние файлы.

Встроенные скрипты пишут внутри тега **<script>**. Например:

```
<script>
  JavaScript-код
</script>
```

Тег **<script>** можно использовать в любом месте HTML-документа, но лучше вставлять его в самом конце перед закрывающим тегом **</body>**.

Часть возможностей JavaScript постепенно переходит в CSS, например, возможность задавать плавное изменение значений свойств.

Скрипты чаще всего подключают из внешних файлов с расширением **.js**. Для этого используют тег **<script>** с атрибутом **src**, в котором указывается путь к файлу. Например:

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

Обратите внимание, что тег **<script>** парный. Если вы подключаете внешние скрипты, тогда то, что внутри тега, игнорируется браузером .

Внешние скрипты лучше подключать перед закрывающим тегом **</body>**.

2

Типы данных и переменные

Основные типы данных в JavaScript:

string	строка — какая-либо последовательность символов;
number	число (2, 4, 1e10, 3,14 и т. д.);
bool	логический тип данных; может принимать два значения: true или false;
object	объект;
function	функция;
undefined	тип данных не опредёлен

JavaScript — язык со свободными типами данных, т.е. можно проводить операции с различными типами данных (складывать число и строку и т. д.)

Переменные

Переменная в JavaScript начинается с буквы, знака доллара или подчеркивания и может содержать только буквы, знак доллара, подчеркивание и цифры. Регистр, в котором написана переменная, важен (**a** и **A** - разные переменные).

Создаются переменные очень просто:

- **var имя** — создается пустая переменная;
- **var name=value** — создается переменная с заданным значением.

Константы

В JavaScript нет констант, но поскольку необходимость в них всё же есть, то появилась договорённость: переменные, набранные в верхнем регистре через подчёркивание, не изменять:

```
var USER_STATUS_ACTIVE = 1;
var USER_STATUS_BANNED = 2;
```

Но в стандарте ECMAScript 2015 константы были введены. Ключевое слово **const** создаёт новую именованную константу, доступную только для чтения.

Синтаксис:

```
const name1 = value1
```

Эта синтаксическая конструкция создаёт новую константу. Имена констант подчиняются тем же правилам, что и обычные переменные. Значение константы нельзя менять/перезаписывать. Также её нельзя объявить заново.

Приведение типов

Для проведения некоторых операций требуется преобразование одного типа данных в другой. Для этого существуют специальные функции:

- ***Number(выражение\переменная)*** — преобразование в число; возвращает либо число, либо **NaN** (не число). Для логических величин возвращает **1,** если **true,** и **0,** если **false**. Для строк пытается привести к числу (возможно преобразование в число таких строк, как "**12312**", "**3,14**" и т.д., при передаче в качестве параметра строки типа "7 дней" вернет NaN.

- ***parseInt(string[, base]*** — пытается вернуть целое число, записанное в строке. Если указан необязательный параметр base, то считает число по основанию, указанному в **base** (8,16, по умолчанию, 10, указывать не обязательно);
- поставить знак **+** перед строкой, что тоже приведет к переводу строки к числу +"2";
- ***parseFloat (string)*** — пытается преобразовать строку в вещественное число;
- ***String (выражение\переменная)*** — преобразует данные в строковой формат. Для логических данных возвращает **true** или **false**, для **undefined** - пустую строку (**""**);
- Если к числу добавить пустую строку — это переведет данные в строковый формат: 23 + "";
- ***Boolean(данные)*** — приводит данные к логическому типу. **False** возвращается, если данные — пустая строка или **0**, а в остальных случаях возвращает **true**;
- ***Array(1[,2[,3...]])*** — возвращает массив, состоящих из переданных команде элементов;

Перевод числа в строку и обратно:

```
var n = parseInt("3.14"); // 3
var n = parseFloat("3.14"); // 3.14
var n = 5;
var m = n.toString();
var m = n+'';
var m = new String(n);
```

Узнать, к какому типу данных принадлежит значение переменной **myVar**, можно с помощью оператора **typeof**:

```
typeof myVar;
```

Вот список того, что этот оператор может вернуть:

- **undefined** (для undefined);

- **string** (для String);

- **number** (для Number);

- **boolean** (для Boolean);

- **object** (для всех объектных типов данных и null);

- **function** (для функций).

3

Арифметические операторы

Операторы языка:

+ сложение (для строк — конкатенация (склейка) строк);	
- вычитание;	
*****	умножение;
/	деление;
%	остаток от деления (a mod b);
++	увеличение на 1;
--	уменьшение на 1;
+= a	увеличение на a;
-=a	уменьшение на a;
***=a**	умножение на a;
/=a	деление на a;
%=a	остаток от деления на a;

Для работы с математическими функциями есть специальный объект **Math**, у которого есть свойства и методы. Этот объект — свойство глобального объекта.

Math.функция

Math.abs(number)	модуль числа
Math.acos(number)	арккосинус
Math.asin(number)	арксинус
Math.atan (number)	арктангенс числа

Math.cos (number)	косинус числа
Math.sin(number)	синус числа
Math.tan(number)	тангенс числа
Math.exp(number)	e^{number} (e в степени number)
Math.floor(number)	округление вниз
Math.ceil(number)	округление числа вверх
Math.round(a)	округление до ближайшего целого
Math.log(number)	натуральный логарифм (логарифм a по основанию b находится так: $\log_b a = \ln a / \ln b$
Math.min(a,b), Math.max(a,b)	минимальное или максимальное из двух чисел
Math.pow(a,b)	a в степени b (a^b)
Math.sqrt(a)	квадратный корень из a
Math.random()	случайное вещественное число в промежутке от 0 до 1

Math.константа

Math.E	постоянная Эйлера
Math.PI	число Пи

Примеры использования объекта **Math**

Синус угла в 30

```
var x = 30;
var y = Math.sin(x * 180 / Math.PI);
```

Корень 5-й степени из числа

```
var x = 30;
var y = Math.pow(x, 1 / 5);
```

Для арифметических операций существуют следующие специальные значения:

NaN (not-a-number)	результат числовой операции, которая завершилась ошибкой
Infinity	бесконечность 1.7976931348623157E+10308 (т.е. больше)
-Infinity	бесконечность -1.7976931348623157E+10308 (т.е. меньше)

Уникальность **NaN** — это значение не равно ничему, даже самому себе. Для проверки «нечисла» используется функция глобального объекта **isNaN()**

```
isNaN(n)
```

где параметр **n** — проверяемый результат. Функция проверяет результат на «нечисло», и в случае «нечисла» возвращает **true.**

```
isFinite(n)
```

где **n** — проверяемый результат. Функция проверяет результат на «бесконечность» и «нечисло» . Для конечных чисел возвращает **true.**

```
parseFloat(n)
```

где **n** — строка, из которой пытаемся извлечь число. Если строка начинается с цифр, извлекает вещественное число и возвращает его. Иначе возвращает **NaN**

```
parseInt(n[, s])
```

где **n** — строка, из которой пытаемся извлечь число, а **s** — основание системы счисления (по умолчанию 10). Если строка начинается с символов, допустимых для системы счисления, извлекает число в этой системе и возвращает его. Иначе возвращает **NaN.**

Применять на практике следует метод **isFinite()** — он не только проверит результат на «нечисло», но и на превышение допустимого диапазона (**Infinity** и **–Infinity**). Проверку на «нечисло» с помощью **isNaN()** можно применять только для случаев, когда в результате не может появиться +- бесконечность.

4

Логические операторы и операторы сравнения

Экземпляры логического типа данных Boolean имеют два возможных значения — true и false. Этот тип данных используется там, где есть проверка на соответствие условию. JavaScript легко преобразует типы данных из одного в другой, что следует учитывать в таких операциях.

==	равно
!=	не равно
>	больше
>=	больше либо равно
<	меньше
<=	меньше либо равно
!	отрицание
&&	и
||	или

Из любого типа данных можно получить логический тип данных с помощью двойного отрицания:

```
var x = 15;
var y = !!x; //вернет true
```

5

Условные операторы

«Проверка условия» представляет собой операцию, в результате которой возвращается значение логического типа Boolean. Это может быть любая проверка — наличие переменной или ее значения, сравнение двух переменных и т. д. Если в блоке инструкций стоит единственная инструкция, фигурные скобки можно опустить. Но лучше этого не делать. Хотя бы на первых порах. Работает это так: ключевое слово **if** принимает некое выражение для проверки, и, если выражение возвращает **true**, выполняется блок инструкций, стоящий за этим ключевым словом. Совместно с ключевым словом **else** предыдущая конструкция работает как триггер (переключатель).

Если выражение проверки вернуло **false**, то выполняется блок инструкций, стоящий после **else**. Выполниться может всегда только один блок инструкций в зависимости от результатов проверки условия.

If (условие) {code} — если условие истинно, то выполняется код в **{ }**

```
if (условие) {
   code
} else {
   code 2
}
```

Если условие истинно, то выполняется **code**, иначе **code 2.**

Для экономии ресурсов машины применяется инструкция переключения — выбор между заранее заданными вариантами. Эта инструкция может быть полезна в случае заранее ограниченного выбора. Синтаксис этой инструкции отличается от других и немного более сложен (наборы инструкций не объединяются в блоки).

```
switch (выражение) {
   case value1:
        код;
        break;
   case value2:
        код2;
        break;
   default:
   код
}
```

Значение переменной сравнивается со значениями, стоящими после ключевых слов **case**. В случае равенства выполняются все инструкции, стоящие после найденного совпадения. Ключевое слово **default** нужно для того, чтобы выполнился код в том случае, когда не найдено ни одно совпадение. Например, вывел бы отрицательный результат поиска.

Применение **default** необязательно. Это ключевое слово может стоять в любом месте, а не только после всех **case** (в этом случае следует заканчивать стоящий после него код инструкцией **break**).

Применение инструкции **break** приводит к тому, что все инструкции, идущие за ней, игнорируются и происходит переход к коду, идущему за инструкцией переключения **switch**. Если не применять инструкцию **break**, то выполнятся все инструкции после первого найденного совпадения, что требуется не всегда. В этом принципиальное отличие **swith/case** от **if/else if/else** — в условной инструкции выполняется блок инструкций, соответствующий условию, а в инструкции переключения определяется только точка входа в исполняемый код.

Тернарный оператор. Этот оператор состоит из 3-х частей:

- Проверка условия.
- Значение, возвращаемое оператором в случае успешной проверки (**true**).
- Значение, возвращаемое оператором в противном случае (**false**).
- условие **?** код 1 : код 2 — если условие истинно, выполняется код 1, иначе код 2.

```
(a > b) ? alert(“Maximum a!”) : alert(“Maximum b!”);
```

6

Операторы цикла

Для выполнения повторяющихся действий применяются инструкции цикла. Любая такая инструкция связана, прежде всего, с тремя обязательными шагами:

- Инициализация переменной (или переменной цикла).
- Проверка условия, связанного с переменной цикла.
- Изменение переменной цикла.

Любая инструкция цикла выполняет блок инструкций, если проверка условия возвращает **true**. В противном случае выполнение цикла прекращается.

Цикл for (инициализация, условие, изменение) {code} — цикл выполняется до тех пор, пока условие истинно. Например:

```
for (i = 0; i < 13; i++) {
   a+=a;
}
```

Основное отличие — инициализация, проверка и изменение переменной цикла стоят в одном месте, что облегчает код. В разделе инициализации можно применить оператор **“,”**, объединив инициализацию или объявление нескольких переменных. Ровно так же можно поступить и в разделе изменения — с помощью того же оператора **“,”** можно изменять значения нескольких переменных.

Цикл while (условие) {код} — пока условие истинно, выполняем код.

```
var count = 0; //инициализация переменной цикла
//проверка переменной цикла блок инструкций
while (count < 10) {
   count++; //изменение переменной цикла
}
```

Если проверка вернет **false**, блок инструкций не выполнится ни разу. Если в фигурных скобках стоит единственная инструкция, скобки можно опустить (на первых порах лучше этого не делать). Когда это возможно? Очевидно, если единственной целью выполнения цикла является изменение переменной цикла.

Цикл do { код } while (условие) — аналогично предыдущему, только сначала выполнится код, а потом проверится условие:

```
var count = 0;
do {
   блок инструкций
   count++;
} while (count < 10);
```

Главное отличие от инструкции **while** — выполнение блока инструкций хотя бы один раз. Проверка условия стоит после исполняемого кода. Такая задача очень специфична. Небольшие отличия — **";"** в конце этой инструкции, поскольку она заканчивается проверкой, а блок инструкций стоит до проверки.

Инструкция break в инструкциях цикла

Аналогично инструкции переключения **break** вызывает прекращение выполнения цикла (все инструкции, идущие за **break** в теле цикла, игнорируются) и переход к коду, стоящему за этим циклом.

Инструкция continue в инструкциях цикла

Эта инструкция применяется только в инструкциях цикла. Встретив её, программа игнорирует все инструкции, стоящие в теле цикла за **continue** и переходит к следующей проверке условия цикла. Это приводит к разному поведению для разных инструкций цикла.

- ***for***

  Перед проверкой условия производится автоматическое изменение переменной цикла. Поэтому происходит переход к следующему шагу проверки.

- ***while, do/while***

  Всё зависит от того, где стоит инструкция с изменением переменной цикла. Если до continue, то происходит переход к следующему шагу проверки. Если после, то произойдет возврат к предыдущему шагу — изменения переменной цикла не произошло. Очевидно, что последний вариант приведет к бесконечному циклу.

7

Функции

Функция — блок инструкций, который можно выполнить в любой момент, вызвав функцию. Этот блок инструкций может быть вызван многократно. Блок инструкций часто называют телом функции. Как правило, для функций так же, как и для переменных, используют имена.

Создание функции в JavaScript:

```
function name(parametr1[,parametr2...]) {
   code
}
```

Вызов

```
name(параметры);
```

Примечание: запись **`func(a1[,a2[,a3..]])`** означает, что в **`[]`** находятся необязательные параметры, которые можно опустить. Т. е. если написано **`s(a[,b])`**, то можно писать и **s(a)**, и **s(a,b)**.

Вот элементарный пример:

```
function hello() {
   alert("Hello world");
}
hello(); // Hello world
```

Способы создания функций

- С помощью ключевого слова **function (Function Declaration)**

```
function f(x, y) { блок инструкций; }
```

- С помощью функционального литерала (**Function Expression**)

```
var f = function(x, y) { блок инструкций; };
//обратите внимание на “;” – это просто инструкция
```

- С помощью вызова класса конструктора (используется крайне редко)

```
var f = new Function('x, y', 'блок инструкций в строку');
```

Анонимные функции

В JavaScript можно создавать анонимную функцию (т. е. функцию без имени), для этого достаточно слегка изменить предыдущую конструкцию:

```
function() {
   alert("Hello world");
}
```

Так как функция — это вполне себе объект, то её можно присвоить переменной и (или) передать в качестве параметра в другую функцию:

```
var myAlert = function(name) {
   alert("Hello " + name);
}
function helloMike(myFunc) {
// тут функция передаётся как параметр
   myFunc("Mike");
}
helloMike(myAlert);
```

Анонимную функцию можно создать и тут же вызвать с необходимыми параметрами:

```
(function(name) {
  alert("Hello " + name);
})("Mike");
```

С функциями и глобальным объектом связано понятие контекста выполнения и области видимости переменных. До применения функций мы создавали переменные в глобальном контексте. Эти переменные называются глобальными. Переменная, создаваемая внутри тела функции с помощью оператора var, является локальной. Локальные переменные видны только внутри тела функции. Это означает следующее:

- Невозможно обратиться к локальной переменной вне тела функции, в котором эта переменная была определена.
- Для названия локальных переменных можно использовать те же имена, что уже были использованы для глобальных переменных или внутри других функций.

Переменная, объявленная без оператора var, автоматически является глобальной. Присвоив такой переменной некоторое значения, мы присвоим это значение глобальной переменной. Если глобальной переменной с таким именем не было, она будет создана и её значение будет равно новому значению. Ни в коем случае не следует определять локальные переменные без оператора var, так как это может привести к непредсказуемым последствиям. Если внутри функции мы обращаемся к переменной с некоторым именем, эта переменная сначала ищется среди локальных переменных. Если такая переменная среди локальных переменных этой функции (в её контексте) не найдена, следует переход в контекст внешней функции и поиск среди переменных этого контекста. Эта операция происходит, пока программа не доходит до глобального объекта.

Все аргументы функции являются локальными переменными этой функции. Внутри функции переданные значения доступны под именами переменных, написанных при определении функции.

Что делать, если мы хотим передавать в функцию произвольное число аргументов, например, вычислять сумму любого количества чисел? Для этого у каждой функции есть локальная переменная – объект arguments, предоставляющий доступ ко всем переданным аргументам.

С помощью свойства **arguments.length** у нас есть возможность узнать количество аргументов, которое функция ожидает получить.

Строго говоря, функция не обязана ничего возвращать. Но иногда нужно, чтобы результатом выполнения функции было некоторое значение, и это значение можно было бы присвоить некоторой переменной. В этом случае нужно применить инструкцию **return**. Эта инструкция прерывает выполнение функции и возвращает то, что написано правее. Если ничего справа не указывать, произойдет просто прерывание выполнения функции. В любом случае, выполнение программы вернется в ту точку, откуда эта функция была вызвана.

```
function f(x, y) {
   return (x + y);
//все дальнейшие инструкции не будут выполняться
}
var a = f(2, 4);
//переменной a будет присвоено значение 6
```

Встретив инструкцию return, программа игнорирует код, следующий в теле функции после этой инструкции. В этом плане она похожа на инструкцию break для циклов. Кроме этого, инструкция возвращает исполнение кода в точку, где произошел вызов функции.

8

Работа со строками

Строка – набор символов, заключенных в одинаковые кавычки (двойные или одинарные).

Способы создания экземпляров String

- С помощью вызова класса-конструктора

```
var s = new String("text");
var s = String("text");
```

- С помощью строкового литерала (преимущественно)

```
var s = "text";
```

Специальные функции для работы со строками:

encodeURIComponent(*str*)	кодирует компонент универсального идентификатора ресурса (URI)
decodeURIComponent (*encodedURI*)	обратная операция
eval	вычисляет значение выражения, записанного в строку. Например, eval("2+5") вернет 7

Свойства и методы объекта String:

Записывается так: **имя переменной.свойство (метод)**

- ***length*** — длина строки;

- ***prototype*** — создает свое свойство или метод к объекту;

- ***charAt (index)*** — возвращает символ, находящийся на позиции index;

- ***concat(string)*** — присоединяет к строке переданный параметр (аналог +=);

-- ***indexOf(подстрока[,начиная откуда])*** - ищет индекс первого вхождения подстроки в строку, начиная от переданного параметра (по умолчанию 0);

- ***lastindexOf(подстрока[,начиная откуда])indexOf(подстрока[,начиная откуда])*** - аналогично предыдущему, но поиск идет с конца, т. е. возвращается последнее вхождение, а не первое;

- ***slice (i1[,i2])*** — возвращает подстроку от i1 до i2, если i2 не задан, то до конца. Последний символ не включается.

- ***substring(i1,i2)*** — тоже самое, только включая последний символ.

- ***substr(index[,length])*** — возвращает подстроку с заданной длиной, начиная от заданного индекса.

- ***toLowerCase(string), toUppercase(string)*** — переводит строку в верхний или нижний регистр.

Использование апострофа в строках:

```
var n = 'The dog took it\'s bone outside';
```

Примеры применения строковых функций:

- ***charAt()*** сообщает, какой символ находится в определённой позиции строки. Поэтому **`'Test'.charAt(1) = 'e'`**.
- ***length*** сообщает длину строки . **`'Test'.length = 4`**.
- ***substring()*** выдаёт строку между двумя индексами. **`'Test'.substring(1, 2) = 'e'`**.
- ***substr()*** аналогична substring(), только второе число является не индексом, а длиной возвращаемой строки. Если это число указывает на позицию за пределами строки, то substr() вернёт существующую часть строки. **`'Test'.substr(1, 2) = 'es'`**;
- ***toLowerCase()*** и ***toUpperCase()*** делают то, что обозначают: преобразуют строку в нижний или верхний регистр символов соответственно. **`'Test'.toUpperCase() = 'TEST'`**;

Примеры всех приведённых выше функций:

```
alert('This is a Test'.indexOf('T'));     //  0
alert('This is a Test'.lastIndexOf('T')); //  10
alert('This is a Test'.charAt(5));        //  i
alert('This is a Test'.length);           //  14
alert('This is a Test'.substring(5, 9));  //  is a
alert('This is a Test'.substr(5, 9));     //  is a Test
alert('This is a Test'.toUpperCase());    //  THIS IS A TEST
alert('This is a Test'.toLowerCase());    //  this is a test
```

9

Массивы

Массивы - множество переменных, объединённых общим именем.

Создание массива: **name = new Array ([длина])**

Пример объявления:

```
var a=new Array(); // как объект
var a = []; // с помощью литерала
```

Пример присваивания:

```
a[0]=0;
a[1]="one";
a[2]=true;
```

И т. д.

Свойства и методы массивов в JavaScript:

length — длина массива;

concat(array2) — прибавить второй массив к первому;

join(разделитель) — создать из элементов массива строку с указанным разделителем;

pop — выкинуть (удалить) последний элемент массива, вернув его значение;

push (значение) — добавить элемент в конец массива и вернуть новую длину массива;

shift — удалить первый элемент и вернуть его значение;

unshift (what) — добавить элемент в начало массива;

reverse() — инвертировать массив (из массива 1 2 3 получится 3 2 1);

slice (index1, index2) — подмассив от index1 до (index2 - 1).

Массив является списком элементов. Каждый элемент массива может быть чем угодно, но обычно они связаны друг с другом. Если, например, необходимо отследить 30 студентов класса, то можно создать массив студентов:

```
var students = new Array();
students[0] = 'Sam'; students[1] = 'Joe'; students[2] =
'Sue'; students[3] = 'Beth';
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
```

Пример:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
var suffixes = ['1st', '2nd', '3rd', '4th'];
for(var i = 0; i < students.length; i += 1){
   alert(suffixes[i] + ' студент --  ' + students[i]); }
```

Важный момент, который необходимо знать о массивах, состоит в том, что каждый элемент массива может содержать любой произвольный объект.

Чтобы добавить новый элемент, надо просто задать значение для 5-го элемента:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
students[4] = 'Mike';
students[students.length] = 'Sarah';
students.push('Steve');
/* теперь массив содержит 7 элементов: ['Sam', 'Joe',
'Sue', 'Beth', 'Mike', 'Sarah', 'Steve'] */
```

Иногда необходимо удалить объект из массива. В этом случае задействуется функция **splice**:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth', 'Mike',
'Sarah', 'Steve'];
students.splice(4, 1);
```

Splice в этом примере получает два аргумента: начальный индекс и число элементов для удаления. В результате имеем массив с удалённым Mike:

```
['Sam', 'Joe', 'Sue', 'Beth', 'Sarah', 'Steve'];
```

Часто бывает необходимо преобразовать массив в строку или строку в массив. Имеется две функции, которые могут это сделать: **join** и **split**.

Функция **join** получает массив и преобразует его в строку с помощью разделителя, заданного в **join**. Функция **split** действует в обратном направлении и делает массив из строки, определяя новый элемент с помощью разделителя, заданного в **split**:

```
var myString = 'apples are good for your health';
var myArray = myString.split('a');
// строка myString разбивается на элементы на каждом
// найденном символе 'a'.
alert(myArray.join(', '));
// преобразуем myArray снова в строку с помощью запятой,
// так что можно видеть каждый элемент
alert(myArray.join('a'));
// теперь преобразуем myArray снова в строку с
помощью // символа 'a',
// так что снова получается исходная строка
```

Функция **pop** удаляет последний элемент из массива и возвращает его. Функция **shift** удаляет первый элемент из массива и возвращает его.

```
var students  = ['Sam', 'Joe', 'Sue', 'Beth'];
while(students.length > 0){
   alert(students.pop());
}
```

Существуют также «ассоциативные массивы», в которых каждый элемент массива ассоциирован с именем в противоположность индексу:

```
var grades = [];
grades['Sam']  = 90;
grades['Joe']  = 85;
grades['Sue']  = 94;
grades['Beth'] = 82;
```

Ассоциативные массивы действуют немного иначе, чем индексные. Прежде всего, длина массива в этом примере будет равна **0**. Как же узнать, какие

элементы находятся в массиве? Распространённый способ сделать это - использовать цикл **for/in**:

```
for(student in grades){
  alert("Оценка " + student + " будет: " +
  grades[student]);}
```

Здесь в цикле переменная **student** будет индексом массива grades. Эта инструкция цикла сильно отличается от рассмотренных выше. Во-первых, мы нигде не инициализируем переменную цикла и нигде ее не изменяем. Во-вторых, **grades** (в примере) – некий объект. Что же происходит в цикле **for/in**? Происходит автоматический перебор названий свойств и методов объекта – на каждом шаге цикла переменной student присваивается название свойства или метода рассматриваемого объекта. Эта инструкция цикла позволяет узнать имена и значения перечисляемых свойств и методов любого объекта (программа, в отличие от нас, знает полный перечень свойств и методов объектов).

10

Объект

Объект — это коллекция именованных свойств и методов.

Способы создания объектов

- С помощью вызова класса конструктора

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

- С помощью объектного литерала (рекомендуем) **`var obj = {}`**;

С помощью второго способа можно сразу создать свойства и методы.

```
var obj = {
   title: 'Название',
   show: function() {
       блок инструкций
   },
   price: 200
};
```

Пары «имя свойства/метода : значение свойства/метода» разделяются запятой. После последней пары запятая не ставится (иначе возникнут проблемы в IE). Если не писать ничего внутри фигурных скобок, будет создан пустой объект. В отличие от массивов, где каждый элемент имеет индекс – номер, по которому можно его найти в любой момент, в объекте элементы представляют свойства (методы) с именем. При создании свойства (метода) мы должны дать ему имя. В результате созданное свойство (метод) хранится с данным именем.

Значением свойства объекта может быть любой тип данных, кроме функций.

Если свойством объекта является функция, то она называется методом. Их можно добавлять и удалять в любой момент, в том числе и явным присваиванием. Создаваемые объекты могут иметь и имеют уже встроенные (унаследованные) методы.

Важный встроенный метод во всех создаваемых объектах:

```
hasOwnProperty(s)
```

где **s** – название свойства или метода. Возвращает логическое значение **true**, если свойство или метод не унаследованы, или **false**, если свойства или метода с таким именем нет, или они были унаследованы

Обращение к свойствам и методам

Получить значение любого свойства или метода объекта можно, обратившись к нему с помощью двух абсолютно равнозначных способов.

```
obj['title']; //строка – значение свойства
obj.title;
obj['show']; //ссылка на функцию – метод объекта
obj.show;
```

Любой метод объекта можно вызвать с помощью этого же синтаксиса и оператора "**()**".

```
obj['show']();
obj.show();
```

Создание свойств и методов

Как создать новое свойство у существующего объекта? С помощью уже рассмотренных вариантов обращения к ним.

```
obj['title'] = 5;
obj.title = 5;
```

Как создать метод объекта? Присвоить ему в качестве значения функциональный литерал.

```
obj.show = function() {
   блок инструкций
};
```

Или присвоить в качестве значения ссылку на глобальную функцию.

```
obj.show = test;
function test() {
   блок инструкций
}
```

Удаление свойств и методов

Для удаления свойств и методов объекта вызывается оператор **delete**.

```
var x = { a: 5 };
delete x.a;
alert(x.a); //вернёт undefined – свойство удалено
```