# Курс «Базовый JavaScript»

Евгений Петров
petroveg@gmail.com

# Содержание


| | |
|---|---|
| 3 | Введение в JavaScript, базовые и тривиальные типы данных |
| 33 | Инструкции и функции |
| 58 | Массивы и объекты |
| 78 | Даты и углубление понятия объекта |
| 91 | Факультатив – регулярные выражения |

# Введение в JavaScript, типы данных

## Введение в JavaScript

### Краткая история JavaScript, текущая версия (1.5)

История названия (начальное LiveScript), итоговое JavaScript в силу «раскрученности» на тот момент Java.

Основные положения:

- Регистрозависимый язык
- Поддерживаемые кодировки (ASCII  для исполняемого кода, Unicode для значения строк и комментариев)
- Интерпретируемый язык (процессор или, по-другому, анализатор JavaScript браузера, другой программной среды)
- Для форматирования кода может использоваться любое количество пробельных символов и знаков переноса строки
- “Camel Style” или «верблюжья нотация» - несколько слов, объединенных в одно, начинаются с символа в верхнем регистре, первое – с символа в нижнем регистре

Язык JavaScript называют языком сценариев – программ, которые должны выполняться при тех или иных условиях.

Сценарии JavaScript выполняются в контексте программной среды (это справедливо для любого интерпретируемого языка). Мы изучаем сначала базовый язык, в дальнейшем – клиентский.

Это означает, что в конечном итоге нас интересует исполнение сценариев JavaScript в браузере пользователя (связь JavaScript и документов, составленных с помощью языков разметки).

Браузеры, в зависимости от даты выхода релиза, поддерживают различные версии реализации JavaScript. В настоящее время можно сказать, что подавляющее большинство браузеров поддерживают версию 1.5 (к примеру, IE 5 поддерживает версию 1.3).

### Стандарты и спецификации

Все возможности базового языка определяются в стандарте ECMA (European Computer Manufacturers Association) *http://www.ecma-international.org/publications/standards/Ecma-262.htm*. Этот стандарт описывает поведение и реализацию скриптовых языков. Вот и говорите потом, что только США рулят в развитии технологий:).

Кроме того, существует и описание скриптового языка для документов, использующих синтаксис XML http://www.ecma-international.org/publications/standards/Ecma-357.htm, известного как E4X.

Связь сценариев с документами, использующими языки разметки, описывается в стандарте DOM (*Document Object Model*) http://www.w3.org/TR/DOM-Level-2-Core/. В следующем стандарте языка разметки HTML 5 эта связь включена в сам стандарт. Этот стандарт будет изучаться во второй части курса.

## Внедрение сценариев в HTML

Существует несколько способов, перекликающихся со способами подключения стилей в HTML-документах.

**Вложение (inline)**

Код сценариев пишется в значении атрибутов событий (изучается во второй части курса).

**Встраивание (embedding)**

Код сценариев пишется непосредственно между тэгами <script>.

**Связывание (linking)**

С помощью атрибута *src* указывается адрес внешнего файла. Этот вид внедрения сценариев предпочтителен и обладает преимуществами перед методом встраивания.

У элемента <script> используется атрибут type="text/javascript" для обозначения типа контента. Значение этого атрибута указывает браузеру на то, каким модулем обрабатывать этот контент (его можно встретить в таких элементах, как <style>, <object>, <embed>, <link>). Раньше использовался атрибут language, но в настоящее время использовать его не рекомендуется.

Факультатив

*Псевдопротокл javascript:*

Инструкции JavaScript также могут встречаться совместно с псевдопротоколом javascript: в значении атрибута *href* у ссылок (или в адресной строке браузера). Такие включения кода будут рассмотрены во второй части – «Клиентский JavaScript».

Факультатив

*Атрибут defer*

У элемента <script> есть специальный (одиночный для HTML-документов) атрибут *defer*, который указывает на то, что браузеру не стоит ожидать загрузки внешнего файла скрипта для начала формирования дерева документа. Причина кроется в методе *document.write()*, способном дописать содержание страницы (метод рассматривается во второй части курса). Стоит отметить, что этот метод в некоторых языках разметки, таких как (x)HTML, не выполняется (попросту запрещен).

Поддержка этого атрибута в различных браузерах отличается и требует отдельного исследования.

Факультатив

*jit компиляция*

Интерпретация кода приводит к тому, что на момент вызова отдельных участков кода начинаются «тормоза». По этой причине в некоторых браузерах реализуется технология jit (just-in-time) – компиляции кода JavaScript (пока известны шаги Mozilla в этом направлении https://wiki.mozilla.org/JavaScript:TraceMonkey). Это позволяет значительно ускорить выполнение скриптов в ряде случаев. Некоторые методы сводят на нет работу таких предкомпиляторов – о таких методах будет разговор в конце базового курса.

Факультатив

*Пробельные символы*

Под «пробельными» понимают 4 служебных символа:

| Символ | Что обозначает | Код символа |
|---|---|---|
| SPACE | Пробел | U+0020 |
| TAB | Табуляция | U+0009 |
| CARRIAGE RETURN | Возврат каретки | U+000D |
| LINE FEED | Перевод строки | U+000A |

Последние два символа формируются при нажатии на клавишу Enter и используются в разных ОС в следующих сочетаниях:

MAC – CR

UNIX – LF

Windows – CR LF

# Синтаксис

Любой код состоит из инструкций, функций и комментариев. Код записывается по строго определенным правилам (так называемый синтаксис языка).

При записи используются собственные имена, а так же зарезервированные слова и знаки препинания (операторы), которые имеют заранее определенный смысл.

## Инструкции и функции

Инструкции представляют собой выполняемое действие – законченные фразы (аналог – предложения в языке).

Инструкции могут быть простыми и составными.

Простые инструкции принято заканчивать “;”. В противном случае процессор JavaScript это делает за нас. В ряде случаев незакрытие инструкций приводит к ошибкам. Кроме того, код, написанный без соблюдения этого правила, невозможно объединить в одну строку.

Составные инструкции и функции имеют свой синтаксис и подробно рассматриваются в следующем модуле.

## Комментарии

Символы, стоящие внутри комментариев, расцениваются как служебные и не обрабатываются как исполняемый код.

С помощью комментариев можно писать служебные пометки, как для себя, так и для других программистов. Это может быть очень полезным в больших программах.

Вторая функция – комментарии могут использоваться для скрытия кода, который на данный момент не нужно выполнять.

Комментарии могут быть 2-х видов.

**Однострочные**

```
//здесь скрытый от исполнения код
```

**Многострочные**

```
/*здесь
скрытый
от исполнения
код*/
```

## Ключевые и зарезервированные слова

Для написания кода используются ключевые слова, которые имеют свой смысл.

**Список ключевых слов**

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

Помимо указанных, некоторые слова зарезервированы в стандарте ECMA для будущих расширений языка.

**Список слов, зарезервированных для будущих реализаций JavaScript**

| | | | | | |
|---|---|---|---|---|---|
| abstract | debugger | final | int | private | super |
| byte | double | float | interface | protected | synchronized |
| char | enum | goto | long | public | throws |
| class | export | implements | native | short | transient |
| const | extends | import | package | static | volatile |

Некоторые слова уже используются в текущей реализации языка при взаимодействии с HTML-документами (для названий встроенных свойств и методов, классов-конструкторов и т.д.).

**Список слов, использующихся в текущей реализации JavaScript**

| | | | | | |
|---|---|---|---|---|---|
| arguments | decodeURIComponent | eval | isNaN | parseFloat | String |
| Array | encodeURI | EvalError | Math | parseInt | SyntaxError |
| Boolean | encodeURIComponent | Function | NaN | RangeError | TypeError |
| Date | Error | Infinity | Number | ReferenceError | undefined |
| decodeURI | escape | isFinite | Object | RegExp | unescape |
| | | | | | URIError |

Все перечисленные слова либо уже имеют конкретное предназначение, либо получат его в будущем. Они используются для обозначения конкретных действий или объектов. По этой причине мы не можем использовать их для обозначения своих произвольных переменных (описано в пункте про переменные).

## Операторы

Часть зарезервированных слов, а так же многие знаки представляют собой операторы языка – элементарные команды для выполнения действия. Например, “=” – оператор присваивания.

Операторы не являются законченными выражениями и всегда применяются лишь как часть инструкций (аналог – предлоги в предложении).

Операторы имеют приоритет выполнения – порядок, в котором будут выполнены действия. Для изменения приоритета используют круглые скобки “()”. Заключенные в них действия выполняются в первую очередь.

Операторы всегда подразумевают наличие операндов – значений или переменных, по отношению к которым и выполняются действия.

Справочная информация

*Операторы JavaScript*

| P | A | Оператор | Типы операндов | Выполняемая операция |
|---|---|---|---|---|
| 15 | L | . | Объект, идентификатор | Обращение к свойству |
| | L | [] | Массив, целое число | Индексация массива |
| | L | () | Функция, аргументы | Вызов функции |
| | R | new | Вызов конструктора | Создание нового объекта |
| 14 | | ++ | Левостороннее выражение | Префиксный или постфиксный инкремент (унарный) |
| | | -- | Левостороннее выражение | Префиксный или постфиксный декремент (унарный) |
| | R | - | Число | Унарный минус (смена знака) |
| | R | + | Число | Унарный плюс (нет операции) |
| | R | ~ | Целое число | Поразрядное дополнение (унарный) |
| | R | ! | Логическое значение | Логическое дополнение (унарный) |
| | R | delete | Левостороннее значение | Аннулирование определения свойства (унарный) |
| | R | typeof | Любой | Возвращает тип данных (унарный) |
| | R | void | Любой | Возвращает неопределенное значение (унарный) |
| 13 | L | *, /, % | Числа | Умножение, деление, остаток |
| 12 | L | +, - | Числа | Сложение, вычитание |
| | L | + | Строки | Конкатенация строк |
| 11 | L | << | Целые числа | Сдвиг влево |
| | L | >> | Целые числа | Сдвиг вправо с расширением знакового разряда |
| | L | >>> | Целые числа | Сдвиг вправо с дополнением нулями |
| 10 | L | <, <= | Числа или строки | Меньше чем, меньше или равно |
| | L | >, >= | Числа или строки | Больше чем, больше или равно |
| | L | instanceof | Объект, конструктор | Проверка типа объекта |

| | | | | |
|---|---|---|---|---|
| | L | in | Строка, объект | Проверка наличия свойства |
| 9 | L | == | Любой | Проверка на равенство |
| | L | != | Любой | Проверка на неравенство |
| | L | === | Любой | Проверка на идентичность |
| | L | !== | Любой | Проверка на неидентичность |
| 8 | L | & | Целые числа | Поразрядная операция И |
| 7 | L | ^ | Целые числа | Поразрядная операция исключающего ИЛИ |
| 6 | L | \| | Целые числа | Поразрядная операция ИЛИ |
| 5 | L | && | Логические значения | Логическое И |
| 4 | L | \|\| | Логические значения | Логическое ИЛИ |
| 3 | R | ?: | Логическое значение, любое, любое | Условный трехместный оператор |
| 2 | R | = | Левостороннее значение, любое | Присваивание |
| | R | *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, \|= | Левостороннее значение, любое | Присваивание с операцией |
| 1 | L | , | Любой | Множественное вычисление |

**Пояснение к таблице**

P – приоритет выполнения. Чем выше приоритет, тем больше цифра.

A – ассоциативность оператора. L – слева направо, R – справа налево.

Операнд – значения, над которыми выполняется действие.

# Переменные и литералы

Для работы с полученными результатами и их хранения используются переменные. У переменных обязательно есть имена (идентификаторы). Фактически – это адресация к памяти, в которой эти значения и хранятся.

В качестве имен переменной категорически запрещается использовать ключевые и зарезервированные слова.

Для формирования имен можно использовать в любом количестве:

- a – z, A – Z (латинские буквы в любом регистре)
- 0 – 9 (арабские цифры)
- _ (символ подчеркивания)
- $ (знак доллара)

**Имя переменной не может начинаться с цифры**. Желательно стараться давать переменным осмысленные имена.

## Объявление и инициализация переменных

Для создания переменной применяется оператор var.

```
var x;
```

Любая переменная может иметь значение. Для этого применяется оператор присваивания.

```
var x = 5;
```

Создание переменной называется ее **объявлением**, присваивание значения – ее **инициализацией**.

То, что записывается с правой стороны от оператора “=”, называется **литералом**, попросту говоря, значением.

## Классификация типов данных, оператор typeof

Значения любых переменных (литералы) в JavaScript могут относиться к одному из заранее предопределенных типов данных.

**Типы данных в JavaScript**

| Тривиальный | Базовый | Объектный | Функция |
|---|---|---|---|
| null | String | Object | Function |
| undefined | Number | Array | |
| | Boolean | Date | |
| | | RegExp | |
| | | Error | |

Все типы данных, кроме тривиальных типов, имеют свои **классы-конструкторы**. Это – специальные встроенные функции, которые вызываются для создания экземпляров своего типа данных.

Что это такое, мы подробнее узнаем в 3-м модуле. Пока нам следует знать, что все классы-конструкторы имеют предопределенные свойства (значения) и методы (функции), а созданные ими экземпляры данных наследуют эти свойства и методы (за редким исключением).

Кроме перечисленных типов данных, всегда есть глобальный объект, который носит абстрактное имя “Global”. Это – самый важный объект, значение которого мы узнаем дальше.

Узнать, к какому типу данных принадлежит значение переменной myVar, можно узнать 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)

# Краткое описание типов данных

## Глобальный объект

Любая программная среда, использующая JavaScript (в том числе и браузер при открытии любой HTML-страницы), создает глобальный объект. Все возможности JavaScript, типы данных и т.д. связаны именно с этим объектом. В дальнейшем мы будем постоянно обращаться к этому объекту, и дополнять наши знания о нем.

Глобальный объект определяет понятие глобального контекста исполнения кода (подробнее этот вопрос рассмотрим в следующем модуле).

## Тривиальные типы данных null, undefined

Тривиальные типы данных всегда означают отсутствие:

- null – отсутствие объекта
- undefined – отсутствие значения или свойства объекта (позже мы узнаем, что это такое)

На практике нередко возникают ситуации, в которых различать эти типы данных нет необходимости.

Факультатив

### *undefined*

В списке зарезервированных слов нет “undefined”. Предполагается, что для получения неопределенного значения используется нигде не использованная переменная. По этой причине не нужно называть свои переменные этим словом.

В старых версиях JavaScript (1.3) для некоторых браузеров (IE <= 5) требовалось объявлять такую переменную самостоятельно во избежание проблем:

```
var undefined;
```

## Базовый тип String

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";
```

Особое значение имеет символ “\” (обратная косая черта или обратный «слеш»). В совокупности с некоторыми другими символами он образует так называемые «управляющие последовательности». Эти последовательности могут обозначать непечатаемые символы и знаки.

Далеко не все управляющие последовательности нужны для отображения на мониторе. Некоторые требуются для управления выводом на других типах устройств, например, на принтере.

**Список управляющих последовательностей**

| Последовательность | Что означает |
|---|---|
| \0 | Символ NULL (\u0000) |
| \b | «Забой» (\u0008) |
| \t | Горизонтальная табуляция (\u0009) |
| \n | Перевод строки (\u000A) |
| \v | Вертикальная табуляция (\u000B) |
| \f | Перевод страницы (\u000C) |
| \r | Возврат каретки (\u000D) |
| \” | Двойная кавычка (\u0022) |
| \’ | Одинарная кавычка (\u0027) |
| \\ | Обратная косая черта или обратный «слеш» (\u005C) |
| \xXX | Символ Latin-1, заданный двумя шестнадцатеричными цифрами |
| \uXXXX | Символ Unicode, заданный четырьмя шестнадцатеричными цифрами |

С помощью управляющих последовательностей можно внутри строки, ограниченной двойными кавычками, поставить двойную кавычку \”. Этот прием называется «экранированием».

**Нумерация** символов в строке **начинается с 0**. Этот принцип нумерации справедлив для многих областей JavaScript.

*Кодировки, \xXX и отличие от \uXXXX*

**Кодировка ASCII**

Эта кодировка включает 128 символов – латинские буквы, арабские цифры, знаки препинания, служебные символы. Является основой в OEM, ANSI кодировках. Именно она используется для написания кода программ (кроме комментариев, значений строк и свойств объектов).

**\x[2 символа в 16-й системе счисления]**

Кодировка Latin-1, или ISO 8859-1 – западноевропейская (относится к ANSI). Зарегистрирована в 1992 году и в HTML используется по умолчанию. Возможные варианты названий:

- SO_8859-1:1987
- ISO_8859-1
- ISO-8859-1
- iso-ir-100
- sISOLatin1
- latin1
- l1
- IBM819
- CP819

Именно коды этой кодировки должны использоваться при записи в строке \xXX. Однако большинство браузеров используют при отображении символов коды из windows-1252, что приводит к путанице для символов с кодом 128 – 255. По факту именно коды windows-1252 служат ориентиром для отображения символов во всех современных браузерах. Это закреплено в стандарте HTML 5.

**\u[4 символа в 16-й системе счисления]**

В XML (в xHTML и других языках разметки) используется UTF-8 (UTF-16). Первые 256 кодовых позиций в Unicode совпадают с ASCII. Для задания кода символа используются 4 цифры в 16-й системе счисления.

Получить 4 цифры для кодирования любого знака можно с помощью таблицы символов («Стандартные» -> «Служебные» -> «Таблица символов»). Открыть ее можно проще – «Выполнить» / charmap.

Наибольший выбор знаков предоставляет шрифт Arial Unicode MS.

Содержание строк в JavaScript всегда предполагает использование Unicode.



*Синтаксис записи в описании методов*

При чтении описания методов следует понимать, что ряд аргументов являются необязательными.

| Запись аргументов | Как читать |
|---|---|
| methodName(x) | *x* – обязательный атрибут |
| methodName([x]) | *x* – необязательный атрибут |
| methodName(x[, y, z]) | *x* – обязательный атрибут, *y* и *z* – необязательные атрибуты |

## Свойства и методы типа данных String

Каждый экземпляр String наследует практически все свойства и методы своего класса-конструктора, кроме одного. Такие методы называются статическими и вызываются с использованием названия класса-конструктора.

**Свойства типа данных String**

| Название | Что означает |
|---|---|
| length | Длина строки (количество символов) |

**Статические методы класса-конструктора String**

| Название | Что означает | |
|---|---|---|
| String.fromCharCode(n1[, …]) | *n1*, … – коды символов Unicode | Создает и возвращает строку из переданных кодов<br>Работает в паре с *charCodeAt()* |

**Методы типа данных String**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| concat(text1[, …]) | *text1*, … – строки для объединения с существующей строкой | Возвращает строку – объединение исходной строки и переданных значений<br>На практике используют оператор сложения “+” |
| charAt(n) | *n* – позиция символа | Возвращает символ строки на указанной позиции |
| charCodeAt(n) | *n* – позиция символа | Возвращает код Unicode символа строки на указанной позиции (0 … 65 535)<br>Эти значения используются в *String.fromCharCode()* |
| indexOf(text[, start]) | *text* – строка, которую нужно найти<br>*start* – индекс стартовой позиции поиска | Позицию первого вхождения указанного текста в строке (>=0) или -1, если ничего не найдено<br>Ищет с начала строки<br>Если стартовая позиция не указана, поиск идет с первого символа |
| lastIndexOf(text[, start]) | *text* – строка, которую нужно найти<br>*start* – индекс стартовой позиции поиска | Позицию первого вхождения указанного текста в строке (>=0) или -1, если ничего не найдено<br>Ищет с конца строки к ее началу<br>Если стартовая позиция не указана, поиск идет с последнего символа |
| replace(text1, text2) | *text1* – строка или регулярное выражение, которые ищем<br>*text2* – строка, на которую заменяем | Возвращает строку, в которой произведена замена первого аргумента на второй<br>Если первым аргументом передается строка, первое ее вхождение заменяется вторым аргументом |
| slice(start[, end]) | *start* – индекс стартовой позиции | Возвращает строку, которая начинается с символа исходной строки со стартовым |

| | | |
|---|---|---|
| | *end* – индекс конечной позиции | индексом и заканчивается символом исходной строки с конечным индексом<br>Символ с конечным индексом не включается<br>Если второй аргумент отсутствует, он считается равным [длина строки – 1]<br>Отрицательные индексы означают отсчет с конца строки<br>Если стартовая позиция больше конечной, вернется пустая строка |
| substring(start[, end]) | *start* – индекс стартовой позиции<br>*end* – индекс конечной позиции | Возвращает строку, которая начинается с символа исходной строки со стартовым индексом и заканчивается символом исходной строки с конечным индексом<br>Символ с конечным индексом не включается<br>Если второй аргумент отсутствует, он считается равным [длина строки – 1]<br>Отрицательных индексов быть не может<br>Если стартовая позиция больше конечной, метод автоматически меняет их местами |
| split(text[, length]) | *text* – разделитель (строка или регулярное выражение), по которому производится разбиение<br>*length* – максимальное количество элементов | Разбивает исходную строку на отдельные строки согласно разделителю<br>Возвращает массив полученных элементов<br>Если указано максимальное количество элементов, длина массива не превышает этого числа<br>Работа с регулярным выражением в качестве разделителя рассмотрена в модуле 4 |
| toLowerCase() | – | Возвращает исходную строку, в которой все символы находятся в нижнем регистре |
| toUpperCase() | – | Возвращает исходную строку, в которой все символы находятся в верхнем регистре |
| match(reg) | *reg* – шаблон регулярного выражения | Поиск в строке по шаблону |
| search(reg) | *reg* – шаблон регулярного выражения | Поиск в строке по шаблону |

Важно – ни один из методов экземпляров String не меняет исходную строку. Все методы возвращают новый элемент (строку или массив), который можно присвоить новой переменной.

Строки можно складывать с помощью оператора сложения “+”.

### *Методы, определенные в другом месте*

Некоторые методы являются общими для всех типов данных, имеющих свой класс-конструктор (все, кроме тривиальных типов). Откуда и как они появляются, мы узнаем при рассмотрении типа данных *Object*. Всего их 6.

Часть этих общих методов используются в служебных целях и не будут интересны нам. В редких случаях некоторые из этих методов могут быть полезны.

Другие общие методы очень специфичны и их предназначение мы рассмотрим позже.

А пока вот пара из общих методов, которые нужны для явного преобразования типов данных.

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| toString() | – | Возвращает строковое значение<br>Для строки метод не представляет ценности |
| valueOf() | – | Возвращает элементарное значение<br>То же самое для строки делает toString() |

## Базовый тип Number

Все числа в JavaScript условно делятся на вещественные и целые. Особенность внутреннего хранения предполагает разные диапазоны для работы с числами:

- Вещественные от ±1.8e308 до ±5e-324
- Целые от -2e53 до 2e53

Несмотря на теоретически бесконечное количество вещественных чисел, это количество ограничено (грубо что-то чуть менее 2e64). Тем не менее, этого хватает для вычислений.

Фактически, все числа хранятся как вещественные и используют для каждого числа 8 байт (*float*).

### Способы создания экземпляров Number

С помощью вызова класса-конструктора

```
var n = new Number(5);
var n = Number(5);
```

С помощью числового литерала

```
var n = 5;
```

### Операторы для работы с числами

| Оператор | Как называется | Как применяется | Результат |
|---|---|---|---|
| + | оператор сложения | 8 + 7 | 15 |
| – | оператор вычитания | 11 – 6 | 5 |
| / | оператор деления | 21 / 7 | 3 |
| * | оператор умножения | 4 * 6 | 24 |
| % | оператор остатка от деления | 5 % 3 | 2 |
| ++ | инкремент (применяется к переменной) | myVar++, ++myVar | myVar + 1 |
| -- | декремент (применяется к переменной) | myVar--, --myVar | myVar – 1 |

Знак “–” (минус), записанный перед числом, делает его отрицательным (смена знака).

Знак “+” (минус), записанный перед именем или значением переменной, пытается превратить значение переменной в число.

Если в результате операции требуется прибавить к имеющемуся числу другое число (отнять и т.д.), то можно применить сокращенную форму записи выражения (составной оператор).

### Сокращенная запись операторов

| Полная форма записи | Сокращенная форма записи |
|---|---|
| x = x + y | x += y |
| x = x – y | x –= y |
| x = x / y | x /= y |
| x = x * y | x *= y |
| x = x % y | x %= y |

*Порядок применения инкремента и декремента*

В зависимости от места указания инкремента и декремента в инструкции, результат выполнения именно этой инструкции может быть различным.

Постфиксное написание

```
var x = 7;
var y = x++;
alert(x); //покажет 8 - инкремент сработал
alert(y); //покажет 7 - в y попало значение x, потом значение x увеличилось
```

**Префиксное** написание

```
var x = 7;
var y = ++x;
alert(x); //покажет 8 - инкремент сработал
alert(y); //покажет 8 - значение x увеличилось, потом в y попало значение x
```

Из примеров видно, что по отношению к операнду инкремент и декремент работают одинаково независимо от формы записи. Разница видна в результатах операции присвоения.

## Свойства и методы типа данных Number

Каждый экземпляр Number наследует все методы своего класса-конструктора. Однако свойства класса-конструктора Number не наследуются – они статические.

**Статические свойства**

| Название | Что означает |
|---|---|
| Number.MIN_VALUE | Наименьшее представимое число |
| Number.MAX_VALUE | Наибольшее представимое число |
| Number.NEGATIVE_INFINITY | Отрицательная бесконечность – возвращается при переполнении<br>Эквивалентно свойству глобального объекта -Infinity |
| Number.POSITIVE_INFINITY | Положительная бесконечность – возвращается при переполнении<br>Эквивалентно свойству глобального объекта Infinity |
| Number.NaN | «Нечисло» – возвращается при ошибке арифметической операции<br>Эквивалентно свойству глобального объекта NaN |

На практике последними тремя статическими свойствами не пользуются – вместо них используются свойства глобального объекта.

**Методы**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| toFixed(n) | *n* – количество цифр после десятичной точки<br>от 0 до 20 | Форматирует число в нотации с фиксированной точкой<br>Возвращает строку – представление числа |
| toExponential(n) | *n* – количество цифр после десятичной точки<br>от 0 до 20 | Форматирует число в экспоненциальной нотации<br>Возвращает строку – представление числа |
| toPrecision(n) | *n* – количество значащих цифр<br>от 1 до 21 | Форматирует число в подходящей для результата нотации<br>Возвращает строку – представление числа<br>Без указания количества равносилен вызову toString() |

**Методы, определенные в другом месте**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| toString([n]) | *n* – система счисления<br>от 2 до 36 включительно | Преобразует число в строку и возвращает результат в зависимости от указанной системы счисления<br>По умолчанию это – 10 |
| valueOf() | – | Возвращает для числа его же значение |

Метод *toString()* для чисел уникален – он позволяет получать строковое представление числа в других системах счисления. Обратный ему метод – функция глобального объекта *parseInt()*.

## Свойства и методы глобального объекта для работы с числами

**Свойства глобального объекта**

| Название | Что означает |
|---|---|
| -Infinity | Отрицательная бесконечность – возвращается при переполнении |
| Infinity | Положительная бесконечность – возвращается при переполнении |
| NaN | «Нечисло» – возвращается при ошибке арифметической операции |

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

Именно эти свойства глобального объекта используются вместо статических свойств класса-конструктора Number.

**Методы глобального объекта**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| isNaN(n) | n – проверяемый результат | Проверяет результат на «нечисло» и в случае «нечисла» возвращает true |
| isFinite(n) | n – проверяемый результат | Проверяет результат на «бесконечность» и «нечисло»<br>Для конечных чисел возвращает true |
| parseFloat(n) | n – строка, из которой пытаемся извлечь число | Если строка начинается с цифр, извлекает вещественное число и возвращает его<br>Иначе возвращает NaN |
| parseInt(n[, s]) | n – строка, из которой пытаемся извлечь число<br>s – основание системы счисления (по умолчанию 10) | Если строка начинается с символов, допустимых для системы счисления, извлекает число в этой системе и возвращает его<br>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.

Факультатив

## *Системы счисления и синтаксис записи*

Кроме 10-ричной системы счисления можно записывать числа в 2-х других системах.

**8-ричная система счисления**

Первая цифра – 0, остальные от 0 до 7. Например, 077 – это 63 в десятеричной системе (7*8 + 7).

Не все реализации JavaScript поддерживают это представление (например, Chrome отличается от других в части работы с 8-й системой счисления), поэтому не стоит придерживаться его в дальнейшем. Кроме того, стандарт ECMA не предписывает такую поддержку.

**16-ричная система счисления**

Начинается с последовательности “0x”, далее цифры от 0 до 9 и буквы от “a” до “f” (или от “A” до “F”). Например, “0xFF” – это 255 в десятеричной системе (15*16 + 15).

Факультатив

## *Особенности работы parseInt() с системами счисления*

Если второй аргумент в методе *parseInt()* не задан, то его определение происходит по схеме:

Первый аргумент начинается с “0x”, второй устанавливается в 16

Первый аргумент начинается с “0”, второй устанавливается в 8 (кроме Chrome)

В остальных случаях второй аргумент устанавливается в 10

Именно по этой причине во всех браузерах, кроме Chrome, результат работы следующего кода будет неожиданным

```
var x = '09';
var y = parseInt(x); //система счисления установилась в 8
alert(y); //вернет 0 - в 8-й системе символа 9 нет
```

Факультатив

## *Работа со значением цвета в 16-й системе счисления*

Получение строки в 16-й системе счисления из числа (цвет в канале RGB)

```
var x = 255;
var y = x.toString(16); //вернет 'ff'
```

Обратная операция

```
var x = 'ff';
var y = parseInt(x, 16); //вернет 255
```

## Объект Math

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

**Свойства объекта Math (константы)**

| Название | Что означает |
|---|---|
| Math.E | **e** – основание натуральных логарифмов |
| Math.LN10 | Натуральный логарифм 10 |
| Math.LN2 | Натуральный логарифм 2 |
| Math.LOG10E | Десятичный логарифм **e** |
| Math. LOG2E | Логарифм **e** по основанию 2 |
| Math.PI | Константа **π** |
| Math.SQRT1_2 | Единица, деленная на корень квадратный из 2 |
| Math. SQRT2 | Квадратный корень из 2 |

**Методы объекта Math**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| Math.abs(n) | *n* – любое число | Возвращает абсолютное значение числа |
| Math.acos(n) | *n* – число от -1.0 до 1.0 | Возвращает арккосинус числа в радианах<br>Результат в интервале от 0 до **π** |
| Math.asin(n) | *n* – число от -1.0 до 1.0 | Возвращает арксинус числа в радианах<br>Результат в интервале от -**π**/2 до **π**/2 |
| Math.atan(n) | *n* – любое число | Возвращает арктангенс числа в радианах<br>Результат в интервале от -**π**/2 до **π**/2 |
| Math.atan2(x, y) | *x* – координата по оси Y<br>*y* – координата по оси X | Возвращает угол между направлением на точку и положительной осью X в радианах (против часовой стрелки)<br>Результат в интервале от -**π** до **π** |
| Math.ceil(n) | *n* – любое число | Возвращает ближайшее целое, большее или равное переданному числу |
| Math.cos(n) | *n* – количество радиан в угле | Возвращает косинус переданного угла<br>Результат в интервале от -1.0 до 1.0 |
| Math.exp(n) | *n* – любое число | Возвращает **e** в степени переданного числа |
| Math.floor(n) | *n* – любое число | Возвращает ближайшее целое, меньшее или равное переданному числу |
| Math.log(n) | *n* – любое положительное число | Возвращает натуральный логарифм числа |
| Math.max(n[, …]) | *n*, … – перечень любого количества чисел | Возвращает максимальное значение из переданных чисел |
| Math.min(n[, …]) | *n*, … – перечень любого количества чисел | Возвращает минимальное значение из переданных чисел |

| | | |
|---|---|---|
| Math.pow(x, y) | x – любое число<br>y – значение степени | Возвращает число, возведенное в степень |
| Math.random() | – | Возвращает псевдослучайное число в диапазоне между 0 и 1.0 |
| Math.round(n) | *n* – любое число | Возвращает целое, ближайшее к переданному числу |
| Math.sin(n) | *n* – количество радиан в угле | Возвращает синус переданного угла<br>Результат в интервале от -1.0 до 1.0 |
| Math.sqrt(n) | *n* – любое положительное число | Возвращает квадратный корень из переданного числа |
| Math.tan(n) | *n* – количество радиан в угле | Возвращает тангенс переданного угла |

Факультатив

*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);
```

Логарифм числа по основанию 10

```
var x = 87;
var y = Math.log(x) * Math.LOG10E;
```

Логарифм числа по основанию 2

```
var x = 145;
var y = Math.log(x) * Math.LOG2E;
```

## Базовый тип Boolean

Экземпляры логического типа данных Boolean имеют два возможных значения – true и false.

Этот тип данных используется там, где есть проверка на соответствие условию. JJavaScript легко преобразовывает типы данных из одного в другой, что следует учитывать в таких операциях.

**Способы создания экземпляров Boolean**

С помощью вызова класса-конструктора

```
var b = new Boolean (true);
var b = Boolean (true);
```

С помощью логического (булевского) литерала

```
var b = true;
```

**Операторы сравнения**

| Оператор | Называется | Что делает |
|---|---|---|
| == | Равенство | Проверяет равенство значений двух объектов<br>Если равны – возвращает true, нет – false |
| === | Идентичность | В отличие от сравнения сначала проверяет соответствие типов данных двух объектов<br>Если объекты идентичны (тип одинаков и значения после преобразования равны) – возвращает true, нет – false |
| != | Неравенство | Проверяет, не равны ли два объекта<br>Если не равны – возвращает true, нет – false |
| !== | Неидентичность | Проверяет, не идентичны ли два объекта<br>Если не идентичны – возвращает true, нет – false |
| > | «Больше» | Если левый объект строго больше правого, возвращает true, нет – false |
| >= | «Больше или равно» | Если левый объект больше или равен правому, возвращает true, нет – false |
| < | «Меньше» | Если левый объект строго меньше правого, возвращает true, нет – false |
| <= | «Меньше или равно» | Если левый объект меньше или равен правому, возвращает true, нет – false |

**Логические операторы**

| Оператор | Называется | Что делает |
|---|---|---|
| ! | Логическое «НЕ» | Возвращает обратное значение (инвертирует его) |
| && | Логическое «И» | Пытается преобразовать значение каждого операнда в логическое значение<br>Прекращает работу, когда значение операнда преобразуется в false<br>Возвращает значение последнего операнда или того, на котором прекратилось выполнение оператора<br>Перебирает все операнды, пока не встретится значение, которое |

| | | |
|---|---|---|
| | | нельзя преобразовать в true |
| \|\| | Логическое «ИЛИ» | Пытается превратить значение каждого операнда в логическое значение<br>Возвращает значение первого операнда, значение которого удалось преобразовать в true или значение последнего операнда<br>Перебирает все операнды, пока не встретится значение, которое нельзя преобразовать в true |

Никаких собственных свойств и методов у класса-конструктора Boolean нет.

В дальнейшем мы увидим, что при сравнении и ряде других операций происходит перевод данных из одного типа в другой. Именно для точной проверки равенства (не только значения, но и типа данных) используется оператор идентичности “===”.

**Преобразования типов данных**

Преобразования значений в JavaScript производятся очень часто.

Следует понимать, что при таких фоновых преобразованиях значение, хранящееся в переменной, не меняется – создается временное значение с типом, который нужен для спешного выполнения операции.

Например, сравниваются строка и число – эти значения приводятся к одному типу данных, сравнение возвращает логическое значение, преобразованные временные значения удаляются.

Все преобразования зависят от контекста – операторов, которые выполняют над ними действия.

| Тип значения | Контекст, в котором используется значение | | | |
|---|---|---|---|---|
| | Строковый | Числовой | Логический | Объектный |
| Неопределенное значение | 'undefined' | NaN | false | Error |
| null | 'null' | 0 | false | Error |
| Пустая строка | Как есть | 0 | false | Объект String |
| Непустая строка | Как есть | Числовое значение строки или NaN | true | Объект String |
| 0 | '0' | Как есть | false | Объект Number |
| NaN | 'NaN' | Как есть | false | Объект Number |
| Infinity | 'Infinity' | Как есть | true | Объект Number |
| -Infinity | '-Infinity' | Как есть | true | Объект Number |
| Любое другое число | Строковое представление числа | Как есть | true | Объект Number |
| true | 'true' | 1 | Как есть | Объект Boolean |
| false | 'false' | 0 | Как есть | Объект Boolean |
| Объект, функция | toString() | valueOf(), toString() или NaN | true | Как есть |

*Преобразования типов данных*

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

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

Получение строки из любого объекта

```
var x = 15;
var y = x.toString();
var y = x + ''; //более короткий способ
```

Получение числа из строкового представления в 10-й системе счисления

```
var x = '15';
var y = x * 1; //числовой контекст - оператор умножения
var y = + x; //унарный плюс
var y = parseInt(x, 10); //если строка не в 10-й системе счисления
```



*Логические “И”, “ИЛИ”*

При работе логических “И”, “ИЛИ” происходит преобразование значений операндов в логические, но возвращают эти операторы исходное значение. Эти операторы будут впоследствии использоваться для написания кроссбраузерного кода.

**Логическое “ИЛИ” – ||**

Для завершения работы этого оператора достаточно хотя бы одного значения операнда, преобразованного в true. Как только такой операнд будет найден, будет возвращено его исходное значение.

Если такого операнда найдено не будет, вернется значение последнего операнда.

```
var x = 5;
var y = 'abc';
var z = 0;
var result;
result = x || y || z; //вернет 5
result = z || y || x; //вернет 'abc'
result = '' || z; //вернет 0
```

**Логическое “И” – &&**

Для завершения работы этого оператора необходимо, чтобы значения всех операндов могли быть преобразованы в true. Если значения всех операндов могут быть преобразованы в true, будет возвращено исходное значение последнего операнда.

Если значение хотя бы одного операнда будет преобразовано в false, вернется его исходное значение.

```
var x = 5;
var y = 'abc';
var z = 0;
var result;
result = x && y && z; //вернет 0
result = z && y && x; //вернет 0
result = 7 || x; //вернет 5
```

# Инструкции и функции

## Инструкции

### Понятие инструкции

Инструкция – выражение, в результате которого должны выполниться действия (например, объявление или инициализация переменной и т.д.).

Инструкции могут быть простыми и составными – составные мы рассмотрим ниже. Далее мы будем говорить «инструкция», имея в виду простую инструкцию.

Каждая простая инструкция заканчивается “;”. Если этого не делать, браузер проставляет их за нас. Ориентируется он в этом случае на переносы строки.

Составные инструкции заранее определены в языке и имеют свой синтаксис:

- Условные инструкции (if / else if / else)
- Инструкция переключения (switch / case / default)
- Инструкции цикла (while, do / while, for, for in)
- Инструкция перехвата и обработки исключения (try / catch / finally)
- Инструкция определения контекста исполнения (with) – не изучается

### Объединение инструкций, оператор ","

Иногда возникает необходимость объединить объявления или инициализации нескольких переменных. Чуть дальше мы рассмотрим такие ситуации. Для этого достаточно один раз применить оператор var и далее использовать оператор “,” (запятая) для разделения переменных.

Эти инструкции:

```
var x = 5;
var y = "text";
var z;
```

объединяются в одну:

```
var x = 5, y = "text", z;
```

### Блок инструкций {}

Когда нужно объединить несколько инструкций в исполняемый блок кода, применяют фигурные скобки. Такое объединение необходимо в составных инструкциях и функциях, которые будут рассмотрены в этом модуле.

Нужно понимать, что блок инструкций будет выполняться лишь тогда, когда программа получит указание выполнить его. Как это происходит, рассмотрим ниже.

# Условные инструкции (if, if/else, if/else if/else)

Часто возникает необходимость выполнить какие-то действия только при соблюдении условий.

```
if (проверка условия) {
  блок инструкций
}
```

«Проверка условия» представляет собой операцию, в результате которой возвращается значение логического типа *Boolean*. Это может 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 (проверка условия) {
  блок инструкций
} else {
  блок инструкций
}
```

Ключевые слова if и else могут быть употреблены в ходе одной проверки только один раз. Причем, if встречается обязательно, а употребление else является необязательным.

Для более сложных проверок между этими двумя ключевыми словами может быть применена дополнительная составная инструкция “else if”. Ее можно употреблять сколько угодно раз.

```
if (проверка 1-го условия) {
  блок инструкций
} else if (проверка 2-го условия) {
  блок инструкций
} else {
  блок инструкций
}
```

Важно еще раз подчеркнуть, что выполниться может один, и только один из блоков инструкций.

## Тернарный оператор ?:

Инструкция вида if/else может быть представлена в виде так называемого условного или тернарного оператора

проверка_условия ? значение_1 : значение_2

Следующая инструкция

```
if (x > y) {
  var z = 'x больше y';
} else {
  var z = 'x меньше y';
}
```

Может быть заменена на оператор

```
var z = x > y ? 'x больше y' : 'x меньше y';
```

Этот оператор состоит из 3-х частей:

- Проверка условия
- Значение, возвращаемое оператором в случае успешной проверки (true)
- Значение, возвращаемое оператором в противном случае (false)

Значением может быть выражение, а не просто переменная, но в любом случае это будет единственная инструкция, обязательно возвращающая значение.

Факультатив

### *Тернарный оператор ?:*

С помощью этого оператора можно записать вложенную структуру проверок.

Следующий код

```
if (выражение_1) {
  z = 'значение_1';
} else {
  if (выражение_2) {
    z = 'значение_2';
  } else {
    z = 'значение_3';
  }
}
```

Можно заменить на

```
z = выражение_1 ? 'значение_1' : выражение_2 ? 'значение_2' : 'значение_3';
```

# Инструкция переключения (switch/case/default)

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 значение_выражения_1:
    блок инструкций
    break;
  case значение_выражения_2:
    блок инструкций
    break;
  default:
    блок инструкций
}
```

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*).

## Инструкция break в инструкции переключения

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 бок инструкций, соответствующий условию, а в инструкции переключения определяется только точка входа в исполняемый код.

Инструкцию *break* мы встретим при рассмотрении инструкций цикла.

*Тонкости работы инструкции switch / case / default*

Очень важно понимать, что инструкция переключения задает только начало исполняемого кода – это не переключатель в чистом его виде. В частности, инструкции, стоящие после case, не объединены в блоки, а ключевые слова *case* представляют собой *метки*.

```
var x = 7, y;
switch (x) {
  case 1:
    y = 'товар';
    break;
  case 2:
  case 3:
  case 4:
    y = 'товара';
    break;
  default:
    y = 'товаров';
}
```

Для значений 2 и 3 выполнится код, стоящий за case 4 – инструкции break для этих меток нет.

Для сравнения выражения в круглых скобках и значений, стоящих за словом case, необязательно использовать после case значения базовых типов – там могут стоять выражения, которые могут вернуть значения этих типов. В этом JavaScript отличается от остальных языков.

Сравнение значений производится **без приведения типов**. Это значит, что 3 не будет равно '3'.

```
var x = 3, y;
switch (x) {
  case '3':
    alert('Hooray!'); //не будет выполнено
}
```

# Инструкции цикла (while, do/while, for, for/in)

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.

## Инструкция цикла while

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

## Инструкция цикла do/while

```
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 цикла for

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

Основное отличие от инструкции while – 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.

```
for (var i = 0, x, y = 0; i < 10; i++, y--) {
  блок инструкций
}
```

## Инструкция цикла for/in

```
for (var prop in myObject) {
  блок инструкций
}
```

Эта инструкция цикла сильно отличается от рассмотренных выше. Во-первых, мы нигде не инициализируем переменную цикла и нигде ее не изменяем. Во-вторых, *myObject* (в примере) – некий объект. Что же происходит в цикле *for/in*?

Происходит автоматический перебор названий свойств и методов объекта – на каждом шаге цикла переменной prop 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).

Эту инструкцию мы рассмотрим подробнее на следующем занятии при обсуждении массивов и объектов. Она очень полезна в случае, когда набор свойств и методов объекта нам неизвестен.

**Оператор in**

В инструкции цикла *for/in* применяется оператор *in*. Он позволяет понять, является ли строка названием свойства или метода объекта. Мы рассмотрим на следующем занятии этот оператор подробнее.

Факультатив

*Отличие инструкций for от while*

Кроме синтаксиса, у цикла *for* (*for / in*) есть и еще одно существенное отличие. О нем разговор пойдет при обсуждении инструкции *continue*.

## Инструкции break и continue в инструкциях цикла

### 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 ее, 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, for/in

  Перед проверкой условия производится автоматическое изменение переменной цикла. Поэтому происходит переход к следующему шагу проверки.
- while, do/while

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

### *Инструкции break, continue и метки*

Можно пометить блок инструкций с помощью метки. В этом случае внутри этого блока можно прекратить его выполнение с помощью указания метки после инструкции *break*.

```
var x = 5;
innerloop:
{
  break innerloop;
  x = 7;
}
alert(x); //покажет 5
```

Инструкция *break* в этом случае может применяться не только внутри инструкций переключения и циклов. Метка должна быть внешней по отношению к инструкции *break*. Фактически это означает прекращение работы блока, в котором встретился этот вызов.

Все инструкции, стоящие за таким вызовом, игнорируются, и исполнение кода перемещается в точку за блоком с меткой.

Понять применение такого вызова можно на примере.

```
outerloop:
for (var i = 0; i < 10; i++) {
  for (var j = 0; j < 10; j++) {
    //if (j == 2) break; //Выход из внутреннего цикла
    if (j == 2) break outerloop; //Выход из внешнего цикла
  }
}
alert(i + ' :: ' + j); //вернет '0 :: 2'
```

Блоком инструкций называется как набор инструкций, заключенных в фигурные скобки, так и составные инструкции.

В отличие от инструкции *break*, инструкция *continue* как с меткой, так и без нее может стоять только внутри инструкций цикла. Применение *continue* с меткой еще более виртуально.

Применение меток в JavaScript считается признаком плохого структурирования кода (непрозрачной логики) и не приветствуется.

# Инструкция перехвата и обработки исключения (try/catch/finally)

В тех случаях, когда мы не уверены в корректности некоторых действий по разным причинам, мы используем так называемый перехват исключений (или попросту – ошибок).

```
try {
  блок инструкций //что-то пытаемся сделать
} catch (e) {
  блок инструкций //если не получилось, перехватываем ошибку
} finally {
  блок инструкций //выполняется всегда
}
```

Если при выполнении инструкций внутри блока *try* возникла ошибка (исключение), то выполняется ее перехват и исполнение инструкций внутри блока *catch*. Обратить внимание на переменную (e) – это ссылка на саму ошибку (тип данных Error).

Для выполнения некоего кода независимо от того, произошла ошибка в блоке *try* или нет, применяется ключевое слово *finally*.

Следует отметить, что *catch* и *finally* могут употребляться, могут не употребляться, но хотя бы одно из них должно быть обязательно.

## Тип данных Error

Что такое (e) после ключевого слова catch? Не что иное, как ссылка на ошибку – тип данных Error. Название этой переменной не имеет значения.

Кроме получения ошибки внутри инструкции *try / catch / finally*, ее можно создать самостоятельно с помощью вызова класса-конструктора:

```
var e = new Error('текст ошибки');
var e = Error('текст ошибки');
```

Однако такой созданный объект не имеет для нас значения без «запуска» этой ошибки. Запускается ошибка следующим образом:

```
throw new Error('текст ошибки');
throw Error('текст ошибки');
throw 'текст ошибки';
```

Запуск без вызова класса-конструктора (просто строка сообщения в последнем варианте) приведет к возникновению неопознанной ошибки, и сообщение будет проигнорировано.

Самостоятельное создание ошибки если и используется, то невероятно редко. А вот перехват ошибки с помощью *try / catch / finally* применяется часто.

У объектов типа данных Error есть только одно универсальное свойство *message*, которое содержит текст ошибки.

*Список составных инструкций*

| Инструкция | Синтаксис | Назначение |
|---|---|---|
| if / else if / else | if (выражение_1) { блок инструкций }<br>[else if (выражение_n) { блок инструкций }]<br>[else { блок инструкций }] | Условное исполнение фрагмента программы |
| switch / case | switch (выражение) {<br>[case выражение_n: блок инструкций]<br>[default: блок инструкций]<br>} | Многопозиционное ветвление для инструкций, помеченных метками case и default |
| while | while (выражение) { блок инструкций } | Базовая конструкция для цикла |
| do / while | do { блок инструкций }<br>while (выражение); | Альтернатива циклу while |
| for | for (инициализация; проверка; инкремент) { блок инструкций } | Простой в использовании цикл |
| for / in | for (переменная in объект)<br>{блок инструкций} | Цикл по свойствам объекта |
| try / catch / finally | try { блок инструкций }<br>catch (идентификатор) { блок инструкций }<br>finally { блок инструкций } | Перехват исключения |
| function | function имя_функции([арг1][,аргn])<br>{ блок инструкций } | Объявление функции |
| with | with (объект) { блок инструкций } | Расширение цепочки областей видимости (*не рекомендуется к применению*) |

В этой таблице функция отнесена в разряд составных инструкций исключительно по формальным признакам.

На самом деле между инструкциями и функциями есть ряд существенных различий.

- Функции не выполняются немедленно, в отличие от инструкций.
- Анализ деклараций функций происходит всегда раньше, чем исполнение инструкций.
- Функции определяют область видимости переменных, чего не могут делать инструкции
- Методы объектов реализованы именно как функции

Кроме перечисленных, существует множество более тонких различий.

**Список простых инструкций**

| Инструкция | Синтаксис | Назначение |
|---|---|---|
| break | break;<br>break имя_метки; | Выход из самого внутреннего цикла инструкции switch или инструкции с именем *имя_метки* |
| case | case выражение: { блок инструкций } | Метка для инструкции внутри конструкции switch |
| continue | continue;<br>continue имя_метки; | Перезапуск самого внутреннего цикла или цикла, помеченного меткой имя_метки |
| default | default: { блок инструкций } | Отметка инструкции по умолчанию внутри инструкции switch |
| Пустая инструкция | ; | Ничего не делает |
| Метка | идентификатор: блок инструкций; | Присваивание блоку инструкций имени идентификатор |
| return | return [выражение]; | Возврат из функции и задание возвращаемого функцией значения (равного *выражению*) |
| throw | throw выражение; | Генерация исключения |
| var | var имя_1 [= значение_1]<br>[, имя_n [= значение_n]]; | Объявление и инициализация переменных |

Некоторые инструкции и методы работы с ними по разным причинам не были рассмотрены в основном курсе (читать в факультативах).

Метки инструкций сложны в понимании, однако впоследствии могут использоваться в рабочих программах. Практика показывает, что подобное усложнение не требуется и метки крайне редко можно встретить в программах.

Составная инструкция with сложна сточки зрения процессора и может приводить к странным и трудным для понимания результатам. Использовать эту инструкцию просто не рекомендуется, тем более что ограничение области работы с объектом достигается гораздо более простыми путями.

Эта инструкция описана в разделе, посвященном работе методов объекта.

# Функции

## Имена функций, их создание и вызов

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.

### Передача аргументов

При вызове функции ей можно передать любое количество аргументов, что делает функции незаменимым инструментом программирования. Передача аргументов позволяет получать различные результаты в зависимости от переданных значений. Для передачи аргументов в момент определения функции в круглых скобках записывают имена. При вызове функции вместо этих имен подставляют значения.

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

### Вызов функций

Вызываются функции просто – имя функции и оператор "()". Например, так:

```
f(10, 5);
```

### Свойства функций

| Название | Что означает |
|---|---|
| length | количество ожидаемых аргументов |

*Различия при создании функций*

**Function Declaration**

Перед тем, как приступить к исполнению инструкций, интерпретатор JavaScript сначала анализирует все декларации функций. Таким образом, при выполнении первой же инструкции все объявленные функции уже доступны.

```
f(); //такая функция уже есть
function f() {
  блок инструкций;
}
```

**Function Expression**

А вот инструкция, используемая для создания функции, выполняется в одном ряду с другими.

```
f(); //такой функции еще нет - она создается ниже
var f = function() {
  блок инструкций;
};
```

То же самое касается и создания функций с помощью вызова класса-конструктора. Этот способ используется крайне редко, когда решаются специфические задачи, связанные с контекстом создаваемой функции (*scope*).

## Область видимости переменных

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 ее 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 (ее 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.

```
var g = 5; //g - глобальная переменная
function outer_function(x) { //x - локальная переменная (аргумент)
  var y = 3; //y - локальная переменная
  function inner_function() {
    var z = 8; //z - локальная переменная
    alert(z); //найдена среди локальных переменных функции inner_function
    alert(y); //найдена среди локальных переменных функции outer_function
    alert(g); //найдена среди глобальных переменных
  }
  inner_function();
}
outer_function();
```

## Аргументы функций, объект arguments

Все аргументы функции являются локальными переменными этой функции.

Внутри функции переданные значения доступны под именами переменных, написанных при определении функции.

```
function type_arg(x) { //x - локальная переменная
  alert(x); //будет выведено 7
}
type_arg(7);
```

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. Для работы с ними мы будем использовать методы работы с массивами. Что такое массив, мы узнаем на следующем занятии, но уже сейчас можно сказать, что он позволяет узнать его длину (количество аргументов) и пройтись по всем аргументам с помощью инструкции цикла.

```
function type_sum() {
  for (var i = 0, sum = 0; arguments[i]; i++) {
    sum += arguments[i];
  }
  alert(sum); //будет выведена сумма переданных аргументов
}
type_sum(7, 6, 3, 56, 9);
```

**Свойства объекта arguments**

| Название | Что означает |
|---|---|
| length | количество переданных аргументов |
| callee | ссылка на саму функцию |

С помощью свойства *arguments.callee.length* у нас есть возможность узнать количество аргументов, которое функция ожидает получить (количество переменных, использованных при определении функции в качестве аргументов).

Если, например, у нас есть функция, ожидающая передачи строго трех аргументов, а ей при вызове передается два, то произойдет ошибка. Чтобы избежать ее, можно перед исполнением кода сравнить эти значения.

```
function f() {
  if (arguments.length == arguments.callee.length) {
    здесь помещаем код функции
  }
}
```

## Инструкция return

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.

## Рекурсивный вызов функции

Функцию можно вызвать внутри ее самой с помощью ее же имени, или с помощью ссылки arguments.callee. Такой вызов называется рекурсией и напоминает по работе любой цикл.

При этом справедливы требования, определенные для инструкций цикла – нужна некоторая переменная, которая будет инициализирована при первом запуске функции, будет меняться при каждом повторном запуске и проверка условия для предотвращения бесконечного цикла.

```
function factorial(n) {
  if (n == 1) {
    return n;
  }
  //return n * factorial(n - 1); //функцию внутри себя можно вызвать по имени
  return n * arguments.callee(n - 1); //функцию можно вызвать не по имени
}
alert(factorial(5));
```

*Замыкания или еще раз про область видимости переменных – читать осторожно☺*

Каким же образом внутри функции мы имеем доступ к тем же глобальным переменным или переменным, объявленным во внешней функции?

Каждая функция момент создания образует так называемый «лексический контекст» – специальный объект *scope*, в котором хранятся все локальные переменные этой функции. В этот контекст попадают все переменные ближайшего внешнего контекста и так далее по цепочке вплоть до глобального контекста. В результате мы получаем так называемую *scope chains* – цепочку объектов scope.

```
function f() {
  var z = 7;
  return function() {
    alert(z); //в возвращаемую функцию попало значение внешней переменной z
  }
}
var x = f();
x(); //выведет 7
```

В момент создания анонимной возвращаемой функции в ее контекст попало значение внешней переменной *z*. Теперь мы можем вызывать эту функцию в любой момент времени – значение переменной *z* будет доступно при каждом вызове.

Этот процесс, при котором переменные внешнего контекста попадают в контекст создаваемой функции, и называется *замыканием (closure)*. По определению *замыкание* – это функция, находящаяся внутри создаваемой функции, которая и передает значения внешних переменных в ее лексический контекст. Мы замыкаем значения переменных из внешних контекстов на саму функцию. Эти значения определяются на момент завершения работы ближайшего внешнего контекста.

Мы еще поговорим впоследствии про замыкания после изучения прототипов.

А пока - еще один часто публикуемый пример, вызывающий сложности с пониманием.

```
for (var i = 0, x = []; i < 3; i++) {
  x[i] = function() {
    alert(i);
  }
}
```

Для функций, записываемых в массив, внешним контекстом является тот, в котором выполняется цикл for. После завершения работы цикла переменная i этого контекста равна 3. Именно это значение и попадет в контекст создаваемых функций.

# Массивы и объекты

## Тип данных Array

Массив – это коллекция пронумерованных элементов.

**Способы создания массивов**

С помощью вызова класса конструктора (используется крайне редко)

```
var a = new Array(1, 5, 'text');
var a = Array(1, 5, 'text');
```

С помощью литерала массива

```
var a = [1, 5, 'text'];
```

Если не передать никаких значений в класс-конструктор или ничего не писать внутри квадратных скобок, будет создан пустой массив.

Элементы массива имеют так называемые индексы – его номер по порядку. Нумерация в массиве начинается с нуля. Доступ к элементу массива осуществляется с помощью квадратных скобок и его индекса в массиве.

```
a[1]; //второй элемент массива
```

Создать в массиве новый элемент или переопределить значение уже существующего можно с помощью оператора присваивания.

```
a[10] = 9; //значение элемента массива с индексом 11 теперь равно 9
```

Значениями элементов массива могут быть любые типы данных.

Факультатив

*Особенности работы класса-конструктора Array()*

Создание массива с помощью вызова класса-конструктора имеет особенность. Если в круглых скобках будет стоять единственное число, то будет создан массив с указанным количеством пустых элементов.

```
var a = [5]; //массив a состоит из единственного элемента и он равен 5
var b = new Array(5); //массив a состоит из 5 пустых элементов со значением undefined
```

Именно по этой причине вызов класса-конструктора используется крайне редко.

## Свойства и методы типа данных Array

Каждый экземпляр Array наследует все свойства и методы своего класса-конструктора. Важно отметить, что часть методов изменяют исходный массив. Это означает, что после их применения массив изменяется.

**Свойства экземпляров Array**

| Название | Что означает |
|---|---|
| length | Индекс последнего элемента + 1 |

**Методы экземпляров Array**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| sort([fn]) | *fn* – функция для сортировки<br>`function(a, b){`<br>`  //код функции`<br>`  return n;`<br>`}` | Сортирует массив посимвольно (в алфавитном порядке)<br>Если передаем функцию, то она принимает 2 аргумента, для сравнения которых можно использовать свои правила<br>1-й элемент будет стоять первым, если вернется положительное число, вторым, если отрицательное<br>Если не хотим сортировать, нужно вернуть 0<br>Исходный массив меняется |
| reverse() | – | Переставляет элементы в обратном порядке<br>Исходный массив меняется |
| join([n]) | *n* – строка | Соединяет все элементы массива в строку, используя в качестве соединителя аргумент<br>Если аргумент отсутствует, в качестве соединителя будет использована "," (запятая) |
| slice(start[, end]) | *start* – индекс стартовой позиции<br>end – индекс конечной позиции | Возвращает часть массива, которая начинается с элемента со стартовым индексом и заканчивается элементом с конечным индексом<br>Элемент с конечным индексом не включается<br>Если второй аргумент отсутствует, он считается равным [длина массива – 1]<br>Отрицательные индексы означают отсчет с конца массива<br>Если стартовая позиция больше конечной, вернется пустой массив |
| concat(n1[, ...]) | *n1*, … – любые значения, разделенные запятой | Возвращает новый массив, в который добавляются переданные значения в качестве новых элементов<br>Если любое из значений – массив, то |

| | | |
|---|---|---|
| | | добавляется каждый его элемент |
| pop() | – | Удаляет и возвращает последний элемент массива<br>Исходный массив меняется |
| push(n1[, ...]) | *n1*, … – любые значения, разделенные запятой | Добавляет передаваемые значения в конец массива<br>Исходный массив меняется |
| shift() | – | Удаляет и возвращает первый элемент массива<br>Исходный массив меняется |
| unshift(n1[, ...]) | *n1*, … – любые значения, разделенные запятой | Добавляет передаваемые значения в начало массива<br>Исходный массив меняется |
| splice(start[, length, n1, ...]) | *start* – стартовая позиция элемента<br>*length* – количество элементов<br>*n1*, … – любые значения, разделенные запятой | Универсальный метод<br>Вырезает и возвращает указанное количество элементов, начиная со стартовой позиции<br>Если количество элементов не передается, вырезается все элементы до конца массива, начиная со стартовой позиции<br>Если передаются значения, то они вставляются вместо вырезанных элементов, начиная со стартовой позиции<br>Если в качестве значений для вставки передается массив, то вставляется не массив, а его элементы каждый по отдельности<br>Исходный массив меняется |

Факультатив

### *Пользовательская сортировка массива*

Поскольку сравнение элементов в методе *sort()* работает с элементами, как со строками, для сортировки массива, состоящего из чисел, применяется пользовательская функция.

```
var x = [3, 11, 5];
x.sort(); //теперь x равен [11, 3, 5]
x.sort(function(a, b) {
  return a - b;
}); //теперь x равен [3, 5, 11]
```

Факультатив

### *Универсальный метод splice()*

Последний метод может заменить любой из методов для работы с концом и началом массива.

```
a.splice(0, 1) //замена shift()
a.splice(0, 0, n1, ...) //замена unshift()
a.splice(a.length - 1, 1) //замена pop()
a.splice(a.length, 0, n1, ...) //замена push()
```

Факультатив

### *Добавление элементов и объединение массивов*

Для вставки значений в массив «как есть» применяется метод *push()*

```
var x = [1, 2];
var y = [3, 4];
x.push(y); //x теперь равен [1, 2, [3, 4]]
```

Однако если требуется «сложить» 2 и более массивов, применяется метод *concat()*

```
var x = [1, 2];
var y = [3, 4];
var z = x.concat(y); //z равен [1, 2, 3, 4]
```

«Разложение» добавляемого массива по элементам работает только на первом уровне.

```
var x = [1, 2];
var y = [[3, 4], [5, 6]];
var z = x.concat(y); //z равен [1, 2, [3, 4], [5, 6]]
```

# Тип данных Object

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

```
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.

### 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
}
```

Последний способ имеет преимущество при создании объектов с одним и тем же методом и позволяет экономить ресурсы – в каждом объекте хранится не код функции целиком, а только лишь ссылка на внешнюю функцию.

В дальнейшем мы рассмотрим способ еще больше сократить расход ресурсов при создании объектов одного типа данных с одинаковыми методами (*prototype*).

### Удаление свойств и методов

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

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

## Свойства и методы типа данных Object

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

Встроенных свойств у этого типа данных нет. Методы этого типа данных наследуют все остальные встроенные в JavaScript типы данных.

**Методы экземпляров Object**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| valueOf() | – | Возвращает элементарное значение |
| toString() | – | Возвращает строковое значение |
| toLocaleString() | – | Возвращает строковое значение в местном (локальном) формате |
| hasOwnProperty(s) | *s* – название свойства или метода | Возвращает логическое значение<br>*true*, если свойство или метод не унаследованы<br>*false*, если свойства или метода с таким именем нет, или они были унаследованы |
| propertyIsEnumerable(s) | *s* – название свойства или метода | Возвращает логическое значение<br>*true*, если свойство или метод может быть перечислен с помощью инструкции *for/in*<br>*false*, если свойства или метода с таким именем нет, или они были унаследованы<br>Метод практически идентичен *propertyIsEnumerable* |
| isPrototypeOf(o) | *o* – ссылка на объект | Возвращает логическое значение<br>*true*, если объект является прототипом для указанного объекта o<br>*false*, если это не так |

Эти методы наследуются всеми классами-конструкторами, и присутствуют у любого экземпляра любого типа данных.

Методы *valueOf()* и *toString()* малоинформативны (кроме ситуации с *Number*). Вызываются эти методы автоматически при работе с разными типами данных в зависимости от контекста.

Например, в случаях применения операторов из разного контекста (числового и строчного)

```
alert('7' + 5); //покажет '75', неявно вызван toString() для 5
alert('7' - 5); //покажет 2, неявно вызван valueOf() для '7'
```

Метод *toLocaleString()* полезен только для вывода данных, использующих локальные настройки, например, даты.

Методы *hasOwnProperty()* и *propertyIsEnumerable()* интересны для решения специфических задач (описаны ниже).

Метод *isPrototypeOf()* используется крайне редко для определения факта принадлежности объекта к прототипу другого объекта (прототипы рассматриваются ниже). На практике применяется проверка типа данных с помощью оператора *instanceOf* или свойства *constructor*.

*Название свойства/метода в виде строки*

Такой синтаксис необходим тогда, когда название свойства формируется динамически (например, название передается в качестве значения переменной в функцию).

```
var o = {
  a: 5
}
f(o, 'a');
function f(e, name) { //получаем название метода в виде строки: name = 'a'
  alert(e[name]); //e.name не имеет смысла - это имя переменной, а не свойства
}
```

# Пользовательские объекты

## Понятие класса-конструктора, свойство constructor

Все типы данных в JavaScript, кроме тривиальных типов (null, undefined) имеют собственные **классы-конструкторы**. Можно было заметить, что при их вызове используется оператор вызова функций “()”. Что же представляют собой классы-конструкторы?

Классы-конструкторы – функции. Вызывая их для создания экземпляра типа данных, мы передаем им в качестве аргументов некоторые значения, ровно так же, как поступаем и при вызове функций. После работы класса-конструктора мы получаем от него экземпляр типа данных, который обладает унаследованными свойствами и методами, характерными для своего типа данных.

Как же можно узнать, какой класс-конструктор был использован при создании значения той или иной переменной? С помощью свойства любой переменной constructor.

```
var x = 5;
alert(x.constructor);
```

Для этого примера мы получим:

```
function Number() {
  [native code]
}
```

*[native code]* выводится вместо кода в теле функции – просто его много и он не представляет для пользователей интереса.

Меняя значение переменной на любой тип данных, можно получить название того класса конструктора, который был использован при создании ее значения.

## Оператор instanceof

Проверить принадлежность типа данных можно и с помощью оператора *instanseof*.

```
var x = []; //создаем пустой массив
alert(x instanceof Array); //выводит true
```

Однако, этот оператор для базовых типов данных, созданных с помощью литералов, вернет *false*. Это связано с особенностью реализации базовых типов данных.

```
var x = 5; //создаем экземпляр Number с помощью литерала
alert(x instanceof Number); //вернет false
var x = new Number(5); //создаем экземпляр Number с помощью вызова класса-конструктора
alert(x instanceof Number); //вернет true
```

## Создание пользовательских объектов

Помимо встроенных классов-конструкторов, в JavaScript предусмотрено создание пользовательских классов-конструкторов. Что это такое и для чего это нужно?

Предположим, что для каталогизации библиотеки нужно создать много объектов с уникальными, но похожими свойствами – названием, артикулом, ценой, количеством страниц, авторами и т.д. Кроме того, мы хотим создать для каждого объекта схожие методы, которые, например, выводили бы часть информации, форматируя ее нужным образом.

В таких случаях используется так называемый пользовательский объект. Иначе говоря, мы создаем свой класс-конструктор, который создавал бы новый экземпляр объекта с заданными свойствами. Для этого мы используем функцию, которой даем название.

```
function MyBook(x) {
  this.title = x;
  this.show = function () {
    alert(this.title);
  }
}
var book1 = new MyBook('Основы программирования');
book1.show();
```

Внутри нашего класса-конструктора ключевое слово *this* ссылается на новый создаваемый экземпляр.

Если проверить свойство *constructor* у созданной переменной (book1.constructor), то оно покажет

```
function MyBook(x) {
  код нашей функции целиком
}
```

Таким образом мы написали свой класс-конструктор, который будет создавать наши пользовательские объекты.

К какому типу данных будет принадлежать пользовательский объект? Несомненно, к типу данных Object. В то же время, он будет относиться и к своему классу-конструктору.

```
alert(book1 instanceof Object); //выводит true
alert(book1 instanceof MyBook); //выводит true
```

## *Общение двух разных глобальных объектов*

Попытка проверить с помощью оператора *instanceof* объекта, созданного в другом глобальном контексте (например, при работе с дочерними окнами – *iframe*), приведет к ошибке.

Дело в том, что для каждого глобального объекта реализованы свои классы-конструкторы. По этой причине единственным надежным средством проверки является свойство *constructor*.



## *Object и остальные классы-конструкторы*

Тип данных *Object* является прародителем для абсолютно всех типов данных в JavaScript.

Все встроенные и пользовательские классы-конструкторы наследуют методы этого типа. Фактически, вся структура классов в JavaScript двухуровневая – во главе стоит единственный класс *Object*, все остальные классы – его непосредственные потомки. К сожалению, невозможно создать подкласс, например, типа данных Array.

## Механизм наследования, свойство prototype функций

Мы отмечали, что все классы-конструкторы наследуют свойства и методы от *Object*, а экземпляры этих типов данных наследуют свойства и методы от своих классов-конструкторов. Как же происходит наследование в JavaScript?

У любой функции определено свойство *prototype* и вступает оно в действие при использовании этой функции с оператором *new*, то есть при использовании ее в качестве класса-конструктора. Это свойство ссылается на так называемый «объект-прототип» - виртуальный предок всех экземпляров этого класса-конструктора.

С помощью этого свойства можно создать любое свойство или метод, который будет наследоваться всеми экземплярами этого класса-конструктора. Метод наследования в чем-то аналогичен поведению программы при поиске значения переменной в функции. Если запрашивается некое свойство или вызывается некий метод у переменной, программа пытается найти их у самой переменной. Если поиск не дал результатов, программа ищет их в виртуальном объекте-прототипе. Если и этот поиск не дал результатов, программа ищет их в объекте-прототипе Object (только если сам экземпляр не был создан с помощью этого класса-конструктора). На это цепочка поиска заканчивается.

Таким образом, создание свойств и методов с использованием *prototype* автоматически делает их видимыми даже для всех ранее созданных экземпляров.

С помощью этого механизма наследования можно сократить расход машинных ресурсов для универсальных методов:

```
function MyBook(x) {
  this.title = x;
}
MyBook.prototype.show = function () {
  alert(this.title);
}
var book1 = new MyBook('Основы программирования');
book1.show();
```

В этом случае универсальный метод *show* будет определен только для прототипа класса-конструктора, и при его вызове у любого экземпляра, созданного с помощью этого конструктора, программа найдет его именно в прототипе.

Кроме того, можно определить методы и свойства у любого экземпляра типа данных.

## *Наследование в типах данных JavaScript*

Все типы данных, кроме тривиальных, наследуют свойства и методы своих классов-конструкторов и класса-конструктора *Object* с помощью свойства *prototype*.

Можно определить свойства и методы для всех экземпляров любого типа данных с помощью прототипа класса-конструктора этого типа.

```
Array.prototype.sortAsNumber = function() {
  this.sort(function(a, b){
    return a-b;
  });
}
var x = [2, 11, 3];
x.sortAsNumber();
alert(x);
```



## *Использование for / in для обхода массивов и объектов*

**Массивы**

Предположим, что мы создали массив с «прогалинами» – часть элементов пропущена.

```
var x = [5];
x[99] = 56; // придется бегать по пустым элементам
```

Что делать для проверки всех элементов такого массива?
Можно проверять значения элементов. Вроде этого:

```
for (var i = 0, k = 0; i < x.length; i++, k++) {
  if (x[i]) {
    //тут что-то делаем
  }
}
alert(k); //получим 100
```

Куда как практичнее применить цикл *for/in*

```
for (var i in x) {
  //тут что-то делаем
  k++;
}
alert(k); //получим 2
```

**Объекты**

Цикл for / in используется для обхода свойств и методов объекта, про который нам ничего неизвестно. Особенность этого цикла в том, что можно узнать только те свойства и методы, которые определены именно для этого объекта «вручную». Все наследуемые с помощью *prototype* свойства и методы не могут быть показаны в этом цикле.

Мы можем узнать для объекта только те свойства и методы, которые определены только для него.

```
var x = {
  a: 5
};
Object.prototype.b = 9;
for (var i in x) {
  alert(i + ': ' + x[i]); //выведет 'a: 5' и 'b: 9'
}
```

Никаких общих методов вроде *valueOf()* мы не получим. Только то, что установили в коде самостоятельно (включая прототип).

Факультатив

*Методы propertyIsEnumerable() и hasOwnProperty()*

Эти методы предназначены для проверки возможности перечисления свойств и методов объекта. Унаследованные методы и свойства не могут быть перечислены – только то, что установлено «в лоб». Невозможность перечисления относится и к свойствам и методам прототипа.

```
var x = {
  a: 5
};
Object.prototype.b = 9;
alert(x.propertyIsEnumerable('a')); //выведет true
alert(x.hasOwnProperty('a')); //выведет true
alert(x.propertyIsEnumerable('b')); //выведет false
alert(x.hasOwnProperty('b')); //выведет false
```

Все отличие от возможностей *for / in* в том, что для свойств и методов, установленных в прототипе *propertyIsEnumerable()* и *hasOwnProperty()* выведут false. Так можно при перечислении в цикле понять, что установлено в объекте, что в прототипе.

Факультатив

*Метод isPrototypeOf()*

Применяется крайне редко для проверки - является ли некий объект прототипом другого объекта.

```
var x = [];
alert(Object.prototype.isPrototypeOf(x)); //покажет true
alert(x instanceof Object); //покажет true
```

### Оператор new для классов-конструкторов

Наверняка, уже было отмечено, что вызов встроенных классов-конструкторов был возможен как с оператором new, так и без оного.

Например, результат работы этих инструкций идентичен

```
var x = new Array(1, 4);
var x = Array(1, 4);
```

Это не относится к пользовательским классам-конструкторам. Их можно вызывать для создания новых объектом исключительно с оператором *new*.

Кроме того, тип данных *Date* также не стоит создавать без вызова оператора *new*, иначе результат будет иметь сильно ограниченные рамки и не соответствовать ожиданиям.

### Вызов класса-конструктора Object()

Поскольку тип данных Object является «прародителем» для всех других, можно вызывать его для создания любого встроенного типа данных.

Результат выполнения следующих инструкций будет идентичен

```
var x = new Array(1, 4);
var x = new Object([1, 4]); //в качестве значения использован литерал массива
var x = Object([1, 4]); //результат тот же
```

Такой способ создания подходит только для объектов, имеющих представление в виде литерала, и не подходит для встроенных типов данных, не имеющих литералов (*Error* и *Date*). Для создания новых объектов этих типов следует вызывать собственный класс-конструктор.

Тем не менее, для создания любых объектов класс-конструктор *Object* всегда используется на заднем плане.

```
function f() {}
var x = new f();
alert(x instanceof f); //вернет true
alert(x instanceof Object); //вернет true
```

### Удаление встроенных свойств и методов

При вызове оператора *delete* мы можем удалить только пользовательские свойства и методы объектов (то, что определили для объектов самостоятельно). Поскольку все встроенные свойства и методы находятся в прототипах своих типов данных, то удалить их можно только в самих прототипах (категорически не стоит этого делать).

```
var x = [5, 3];
delete Array.prototype.sort;
x.sort(); //ошибка - такого метода больше нет
```

### Массивы

В массивах удаление элемента не приводит к изменению длины – просто в значении удаленного элемента теперь содержится *undefined*.

```
var x = [5, 3];
delete x[0];
alert(x[0]); //вернет undefined
alert(x.length); //осталось 2 элемента
```

### Переменные, созданные без оператора var

Создавая переменную без оператора var, мы фактически создаем свойство глобального объекта. Отличие от глобальной переменной будет состоять в том, что такое свойство удалить можно, а вот глобальную переменную – нет.

```
var x = 5;
delete x;
alert(x); //как было 5, так и осталось
```

Свойство же глобального объекта удаляется без вопросов

```
x = 5;
delete x;
alert(x); //ошибка - такой переменной нет
```

# Даты и углубление понятия объекта

## Углубление понятия объекта в JavaScript

Как было отмечено ранее, в JavaScript всё является объектами со своими особенностями реализации (например, как базовые типы данных). Это означает, что для любой переменной можно создать свои свойства и методы.

У классов-конструкторов всех типов данных есть унаследованные и встроенные методы и свойства. От кого же наследуются эти свойства и методы? От типа данных Object. Именно этот тип данных является «родителем» всех классов-конструкторов.

Однако следует отметить, что базовые типы данных ведут себя немного иначе, чем объектные типы и функции. Говорят, что базовые типы данных передаются «по значению», а объектные типы и функции «по ссылке».

Продемонстрируем разницу в работе с типами данных.

**Базовый тип**

```
var x = 5; //инициализировали первую переменную – число
var y = x; //присвоили второй переменной значение первой
y = 3; //изменили вторую переменную
alert(x); //увидим 5 - первая переменная не изменилась
```

**Объектный тип**

```
var x = [5]; //инициализировали первую переменную - массив
var y = x; //присвоили второй переменной ссылку на первую
y[0] = 3; //изменили вторую переменную
alert(x[0]); // увидим 3 - первая переменная изменилась
```

Это означает, что для базовых типов (строка, число, логическое значение) в момент создания второй переменной "var y = x;" происходит копирование значения первой переменной. В дальнейшем обе переменных независимы, и меняя значения одной из них, мы не трогаем вторую.

Для объектных типов данных и функций при выполнении этой же операции копирования не происходит, и вторая переменная просто ссылается на тот же объект или функцию. При изменении любой их переменных изменения будут видны и в другой – мы меняем один и тот же объект, на который ссылаются обе переменных.

Именно поэтому говорят, что базовые типы хранятся «по значению», а объектные и функции – «по ссылке».

## Контекст выполнения функции

С контекстом выполнения функций связано значение ключевого слова *this* внутри тела функции. Это ключевое слово ссылается на объект, в контексте которого она выполняется. Это означает, что для следующего кода внутри функции *test* ключевое слово *this* будет ссылаться на объект *obj*.

```
obj.show = test;
function test() {
  здесь ключевое слово this ссылается на obj
}
```

Функция в JavaScript всегда выполняется в качестве метода объекта. Если такой объект невозможно определить, им назначается глобальный. «Безхозных» функций в JavaScript не бывает.

Важно

### *Контекст выполнения функции*

Любая функция исполняется в контексте того объекта, в качестве метода которого была вызвана.

Когда мы вызываем функцию из примера выше вот так

```
test();
```

эта функция вызывается в контексте глобального объекта, и ключевое слово this ссылается на глобальный объект.

Если же исполнить код этой же функции, вызвав его с помощью

```
obj.show();
```

то ключевое слово this будет ссылаться на объект obj.

Факультатив

### *Составная инструкция with*

Эту инструкцию мы не рассматривали, поскольку она сложна в оптимизации кода и при реализации может тратить больше ресурсов, чем альтернативные способы.

Смысл этой инструкции заключается в определении контекста исполнения инструкций в ее теле.

Например, для объекта

```
var x = {
  a: {
    m: '7'
  }
};
```

Можно записать обращение к свойству “m” внутреннего объекта “a”

```
with(x.a) {
  alert(m);
};
```

## Методы функций call(), apply()

Для выполнения функции в качестве метода объекта необязательно регистрировать создавать этот метод, особенно если ее нужно выполнить только один раз. Для этого есть 2 метода у функций.

**Методы функций для выполнения в контексте объекта**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| call(o, n1, …) | o – объект<br>n1, … – аргументы функции | Выполняет функцию в качестве метода передаваемого объекта **o** |
| apply(o, [n1, …]) | o – объект<br>[n1, …] – аргументы функции | Выполняет функцию в качестве метода передаваемого объекта **o** |

Предположим, что уже существуют и объект, и функция.

```
function f(){}
var o = {};
```

Выполнение кода:

```
f.call(o, 1, 2);
```

Равносильно следующему коду:

```
o.m = f;
o.m(1, 2);
delete o.m;
```

Метод *apply()* равносилен *call()* за исключением того, что аргументы передаются в виде массива.

*Прототипы функций и замыкания – читать осторожно☺*

Часто возникает необходимость вызвать метод объекта таким образом, чтобы ключевое слово *this* в этом методе ссылалось на этот объект.

При уже знакомом нам вызове кода метода объекта он выполнится в его контексте.

```
var x = {
  a: 3,
  b: function() {
    alert(this.a);
  }
};
x.b(); //вернет 3 - код функции выполнен в контексте объекта "x"
```

Если у нас появляется новый объект "y", в контексте которого нужно выполнить код объекта "x", то мы уже знаем способ достижения этой цели.

```
var y = {
  a: 7
};
x.b.apply(y);
```

Однако во второй части курса мы увидим, что у глобального объекта появятся методы, выполняющие код в его же контексте. Например, отложенный вызов исполняемого кода *setTimeout()*. Этот метод выполняется в контексте глобального объекта, и все слова *this* в коде вызываемой функции будут ссылаться на глобальный объект.

```
var a = 'global';
setTimeout(x.b, 0); //вернет 'global' - код метода "x.b" выполнен в контексте глобального объекта
```

И вот тогда мы создадим метод в прототипе функций, возвращающий код самой же функции, гарантированно выполняющийся в контексте указанного объекта.

```
Function.prototype.bind = function(e) {
  var o = this;
  return function() {
    o.apply(e, o.arguments);
  };
}
```

Теперь отложенный вызов запустит код в контексте нужного объекта

```
setTimeout(x.b.bind(y), 0); //вернет 7
```

# Тип данных Date

Этот тип данных предназначен для работы с датами и временем.

### Способы создания дат

Экземпляры *Date* всегда создаются с помощью вызова класса конструктора – у этого типа данных нет литерала. В зависимости от типа и количества аргументов создаются даты с различным значением.

### Текущая дата

Если не передавать никаких аргументов, создается объект *Date* с текущим временем и датой.

```
var d = new Date();
var d = Date();
```

Вызывать класс-конструктор *Date* без оператора *new* можно только при создании текущей даты. Все остальные способы создания заданной даты предполагают обязательное использование оператора *new*.

### Заданная дата из миллисекунд

Единственное число в качестве аргумента представляется в виде миллисекунд и создается объект Date по количеству миллисекунд, прошедших с начала 1970 года.

```
var d = new Date(миллисекунды);
```

### Заданная дата из списка чисел

Перечень чисел:

- год – число из 4-х цифр (для числа от 0 до 99 автоматически добавляется 1900)
- номер месяца (нумерация с 0 до 11)
- дата (от 1 до 31)
- часы (от 0 до 23)
- минуты (от 0 до 59)
- секунды (от 0 до 59)
- миллисекунды (от 0 до 999)

Год и месяц – обязательные числа (если будет указано одно число – сработает предыдущий способ для миллисекунд).

```
var d = new Date(2009, 6, 7, 0, 25, 32, 245);
```

Очень важным моментом является то, что при передаче некорректных цифр в класс-конструктор, он исправляет их автоматически.

```
var d = new Date(2009, 12); //будет создана дата 1 января 2010 года
```

### Заданная дата из строки

Единственную строку в качестве аргумента класс-конструктор пытается представить в формате, который может быть понят статическим (не наследуемым) методом Date.parse(). Количество миллисекунд при использовании этого способа указать нельзя.

```
var d = new Date('Tue Jul 07 2009 00:25:32 GMT+0400');
```

Формат строки допускает различные разделители – пробел, запятую, "/", "–" (с некоторыми ограничениями).

```
var d = new Date('Jul/07/2009/00:25:32-GMT+0400');
```

Порядок следования может быть любым – необходимо лишь, чтобы эту строку понял и смог разобрать специальный метод *parse()* класса-конструктора *Date* (описан ниже).

Обязательным из всего этого списка является только месяц, число и год.

```
var d = new Date('Jul/07/2009');
var d = new Date('Jul/07/2009/00');
var d = new Date('Jul/07/2009/00:25');
var d = new Date('Jul/07/2009/00:25:32');
```

Факультатив

### *Системы времени*

При создании дат с помощью описанных вызовов класса-конструктора мы используем так называемое локальное или местное время. Оно зависит от настроек операционной системы и для такого времени указывается смещение относительно Гринвичского меридиана, определяемое часовым поясом.

GMT (Greenwich Mean Time) – система времени, основанная на астрономических наблюдениях и привязанная к Земле. Точка отсчета – Гринвичский меридиан, что и закреплено в названии. Эта система неудобна тем, что в различных точках Земли смещение может быть продиктовано не только геофизическим расположением, но и переходом на летнее/зимнее время. Кроме того, неравномерность времени в GMT (из-за смещения географических полюсов) делает ее некорректной при решении навигационных задач.

Более удобной в этом смысле является система универсального времени UTC (Coordinated Universal Time) . Она была введена в 1964 г. И привязана к равномерной шкале атомного времени. Точка отсчета для удобства осталась на Гринвичском меридиане, но время в этой системе одинаково в любой момент времени в любой точке Земли. При появлении смещения UTC относительно GMT+0 более, чем на 0.9 с происходит коррекция (30 июня или 31 декабря). Первое такое смещение было произведено 30 июня 1972 г.

Именно в этой системе передается время по телевидению, радио, интернет и в навигационных системах.

## Свойства и методы экземпляров Date

У экземпляров Date нет свойств. Вместо них для чтения и записи информации используются методы.

Для работы с локальными, универсальными датами и временами используются наборы очень похожих методов. По этой причине такие методы сгруппированы.

Статические методы не создают экземпляр Date – они просто возвращают количество миллисекунд, прошедших с 1 января 1970 года.

**Статические методы класса-конструктора Date**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| Date.UTC (year, month[, date, hours, minutes, seconds, ms]) | *year* – год<br>*month* – номер месяца<br>*date* – дата<br>*hours* – часы<br>*minutes* – минуты<br>*seconds* – секунды<br>*ms* – миллисекунды | Метод сходен с работой вызова класса-конструктора Date() и передачей набора чисел<br>Основное различие – работа идет с универсальным временем, а не с локальным<br>Возвращает количество миллисекунд с 1 января 1970 года |
| Date.parse(s) | *s* – строка, которая должна быть в формате даты | Метод неявно вызывается, когда в класс-конструктор Date() передается строка<br>Возвращает количество миллисекунд с 1 января 1970 года |

**Методы экземпляров Date – получение информации**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| getYear() | – | Возвращает год<br>Для года, меньшего, чем 2000-й, может вернуть только последние две цифры<br>Метод признан устаревшим |
| getFullYear()<br>getUTCFullYear() | – | Возвращает год (из 4-х цифр) |
| getMonth()<br>getUTCMonth() | – | Возвращает номер месяца (цифра от 0 до 11) |
| getDate()<br>getUTCDate() | – | Возвращает дату (цифра от 1 до 31) |
| getHours()<br>getUTCHours() | – | Возвращает количество часов (цифра от 0 до 23) |
| getMinutes()<br>getUTCMinutes() | – | Возвращает количество минут (цифра от 0 до 59) |
| getSeconds()<br>getUTCSeconds() | – | Возвращает количество секунд (цифра от 0 до 59) |
| getMilliseconds()<br>getUTCMilliseconds() | – | Возвращает количество миллисекунд (цифра от 0 до 999) |

| | | |
|---|---|---|
| getDay()<br>getUTCDay() | – | Возвращает номер дня недели (цифра от 0 до 6)<br>Нумерация начинается с воскресенья |
| getTime() | – | Возвращает количество миллисекунд с 1 января 1970 года<br>Это количество не зависит от часового пояса |
| valueOf() | – | Метод полностью аналогичен методу getTime() |
| getTimezoneOffset() | – | Возвращает разницу с гринвичским временем в минутах<br>Зависит от переходов на летнее/зимнее время |
| toString()<br>toUTCString()<br>toGMTString() | – | Методы выводят данные об экземпляре в строковом формате (дата и время) |
| toLocaleString() | – | Делает то же, что и toString(), но с учетом настроек операционной системы (язык, форматы и т.д.) |
| toTimeString() | – | Выводит только время экземпляра в строковом формате |
| toLocaleTimeString() | – | Делает то же, что и toTimeString(), но с учетом настроек операционной системы (язык, форматы и т.д.) |
| toDateString() | – | Выводит только дату экземпляра в строковом формате |
| toLocaleDateString() | – | Делает то же, что и toDateString(), но с учетом настроек операционной системы (язык, форматы и т.д.) |

**Методы экземпляров Date – запись информации**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| setYear(year) | *year* – год | Устанавливает новый год<br>Для года, меньшего, чем 2000-й, может быть использовано число от 0 до 99<br>Метод признан устаревшим |
| setFullYear(year)<br>setUTCFullYear(year) | *year* – год | Устанавливает новый год (число из 4-х цифр) |
| setMonth(month)<br>setUTCMonth(month) | *month* – номер месяца | Устанавливает новый номер месяца (цифра от 0 до 11) |
| setDate(date)<br>setUTCDate(date) | *date* – дата | Устанавливает новую дату (цифра от 1 до 31) |
| setHours(hours)<br>setUTCHours(hours) | *hours* – часы | Устанавливает новое количество часов (цифра от 0 до 23) |
| setMinutes(minutes)<br>setUTCMinutes(minutes) | *minutes* – минуты | Устанавливает новое количество минут (цифра от 0 до 59) |
| setSeconds(seconds)<br>setUTCSeconds(seconds) | *seconds* – секунды | Устанавливает новое количество секунд (цифра от 0 до 59) |
| setMilliseconds(ms)<br>setUTCMilliseconds(ms) | *ms* – миллисекунды | Устанавливает новое количество миллисекунд (цифра от 0 до 999) |
| setTime(ms) | *ms* – миллисекунды | Устанавливает новое время и дату (полностью меняя экземпляр), принимая количество миллисекунд с 1 января 1970 года<br>Это количество не зависит от часового пояса |

# Методы глобального объекта

В заключение курса следует сказать о нескольких методах глобального объекта.

**Методы глобального объекта**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| eval(c) | *c* – исполняемый код в виде строки | Выполняет код, переданный в качестве строки |
| escape(s) | s – строка для кодирования | Возвращает закодированную копию строки, в которой все символы заменены на соответствующие шестнадцатеричные управляющие последовательности вида %xx или %uxxxx в зависимости от кода символа<br>Не кодируются:<br>a-z A-Z 0-9 - _ . * @ + /<br>Метод признан устаревшим |
| unescape(cs) | *cs* – строка для раскодирования | Возвращает раскодированную копию строки, в которой все шестнадцатеричные управляющие последовательности заменены на соответствующие символы<br>Метод признан устаревшим |
| encodeURI(s) | *s* – строка для кодирования | Возвращает закодированную копию строки, в которой все символы заменены на соответствующие шестнадцатеричные управляющие последовательности вида %xx<br>Символ может быть закодирован одной или несколькими последовательностями в зависимости от его кода<br>Не кодируются:<br>a-z A-Z 0-9 - _ . ! ~ * ' ( )<br>В отличие от encodeURIComponent() также не кодируются:<br>; / ? : @ & = + $ , # |
| decodeURI(cs) | *cs* – строка для раскодирования | Возвращает раскодированную копию строки, в которой все шестнадцатеричные управляющие последовательности заменены на соответствующие символы |
| encodeURIComponent(s) | *s* – строка для кодирования | Возвращает закодированную копию строки, в которой все символы заменены на соответствующие шестнадцатеричные управляющие последовательности вида %xx<br>Символ может быть закодирован одной или несколькими последовательностями в зависимости от его кода<br>Не кодируются:<br>a-z A-Z 0-9 - _ . ! ~ * ' ( )<br>Рекомендуется к использованию |
| decodeURIComponent(cs) | *cs* – строка для раскодирования | Возвращает раскодированную копию строки, в которой все шестнадцатеричные управляющие |

| | |
|---|---|
| | последовательности заменены на соответствующие символы |
| | Рекомендуется к использованию |

Это – 3 пары кодирования/раскодирования текста и метод, позволяющий выполнить строку как исполняемый код.

Для чего нужны операции с кодированием текста? Это связано с кодировкой символов, использующихся в операционной среде. Обмен с сервером информацией, которая содержит не ASCII символы, может привести к потере части этой информации. Когда-нибудь в будущем, возможно, будет введена полная поддержка любых символов различных кодировок.

Работу одного из методов encodeURIComponent() можно увидеть, введя русские слова для поиска в любой поисковой системе.

Помимо обмена информацией эти методы используются для предварительной подготовки информации перед записью в cookie.

Кроме этих пар в списке присутствует метод, позволяющий запустить код, записанный в виде строки. В процессе рассмотрения обмена информацией с сервером мы увидим, что получаем в ответ строку, представляющую собой код JavaScript. Сделать эту строку кодом позволяет метод eval(). Помимо этой операции, мы можем использовать его в некоторых методах, которые рассмотрим во второй части курса.

# Факультатив

# Регулярные выражения

## Тип данных RegExp

Этот тип данных предназначен для выполнения сложного поиска в строках. Пришел в JavaScript из серверного языка программирования Perl.

**Способы создания регулярных выражений**

С помощью вызова класса конструктора

```
var a = new RegExp('регулярное_выражение', 'флаги');
```

С помощью литерала регулярного выражения

```
var a = /регулярное_выражение/флаги;
```

Регулярное выражение составляется с помощью символов и спецсимволов, значение которых рассматривается ниже. Кроме этого, для выражения может быть применен любой из трех флагов.

У способов создания выражения есть основное отличие – при вызове класса-конструктора нужно экранировать обратный слеш, то есть заменить '\' на '\\'.

### Синтаксис регулярных выражений

Все алфавитные символы и цифры ищутся с помощью регулярных выражений ровно так, как они написаны (все символы и именно в таком порядке).

```
var r = /abc/; //будет найдено совпадение с текстом строки 'abc'
```

Этот вид поиска не отличается от уже рассмотренных методов строки indexOf() и lastIndexOf().

## Спецсимволы, неалфавитные символы (управляющие последовательности)

Некоторые символы с обратным слешем обозначают внутри регулярных выражений неалфавитные символы, аналогичные управляющим последовательностям в строках.

**Список управляющих последовательностей**

| Последовательность | Что означает |
|---|---|
| \0 | Символ NULL (\u0000) |
| \b | «Забой» (\u0008) |
| \t | Горизонтальная табуляция (\u0009) |
| \n | Перевод строки (\u000A) |
| \v | Вертикальная табуляция (\u000B) |
| \f | Перевод страницы (\u000C) |
| \r | Возврат каретки (\u000D) |
| \\ | Обратная косая черта или обратный «слеш» (\u005C) |
| \xXX | Символ Latin-1, заданный двумя шестнадцатеричными цифрами |
| \uXXXX | Символ Unicode, заданный четырьмя шестнадцатеричными цифрами |
| \cX | Управляющий символ ^X (например, \cJ соответствует переводу строки \n) |

Кроме этого, некоторые символы играют в регулярных выражениях особый смысл:
^ $ . * + ? = ! : | \ / ( ) [ ] { }

Для поиска таких символов «как есть» используют обратный слеш "\" (экранирование символа).

## Классы символов

Для нахождения любого из целого набора символов применяются классы символов. Класс символов записывается с помощью квадратных скобок “[]”.

```
var r = /[abc]/; //будет найдено совпадение с любым из символов 'a', 'b', 'c'
```

Класс символов может быть применен с отрицанием – в этом случае ищется любой из символов, кроме тех, которые стоят внутри скобок.

```
var r = /[^abc]/; //будет найдено совпадение с любым символов, кроме 'a', 'b', 'c'
```

В классах символов можно указывать диапазон.

```
var r = /[a-c]/; //будет найдено совпадение с любым из символов 'a', 'b', 'c'
```

Некоторые классы используются достаточно часто и для них были придуманы управляющие последовательности.

**Список классов символов**

| Символ | Что означает |
|---|---|
| […] | Один из символов, указанных в скобках |
| [^…] | Один из символов, кроме указанных в скобках |
| . | Любой из символов, кроме перевода строки (или другого разделителя строк в Unicode) |
| \w | Любой символ ASCII. Эквивалентно [a-zA-Z0-9_] |
| \W | Любой символ, кроме символа ASCII. Эквивалентно [^a-zA-Z0-9_] |
| \d | Любая цифра ASCII. Эквивалентно [0-9] |
| \D | Любой символ, кроме цифр ASCII. Эквивалентно [^0-9] |
| \s | Любой символ-разделитель Unicode |
| \S | Любой символ, кроме символа-разделителя Unicode |

## Повторение

Для поиска заданного количества повторений символов существуют специальная запись. Примененная после символов для поиска, эта запись позволяет найти требуемое количество повторов.

Некоторые виды повторений используются часто и для них были придуманы соответствующие обозначения.

**Символы повторения**

| Символ | Что означает |
|---|---|
| {n, m} | Шаблон повторяется не менее n, но и не более m раз |
| {n,} | Шаблон повторяется не менее n, но и не более m раз |
| {n} | Шаблон повторяется точно n раз |
| ? | Шаблон может быть найден 1 раз или не найден вовсе. Эквивалентно {0, 1} |
| + | Шаблон может быть найден 1 раз или более. Эквивалентно {1,} |
| * | Шаблон может быть найден сколько угодно раз или не найден вовсе. Эквивалентно {0,} |

**Нежадное повторение**

Указанное количество повторений при поиске дает максимально возможное количество повторяющихся символов.

```
var r = /a+/; //в строке 'aaa' будет найдена строка целиком
```

Чтобы ограничить повторения нужно после любого из символов повторения поставить “?”.

```
var r = /a+?/; //в строке 'aaa' будет найден первый символ
```

## Альтернатива

Для поиска альтернативных символов или групп символов применяется знак “|”.

```
var r = /ab|cd/; //в строке будет найдены или 'ab' или 'cd'
```

Программа ищет альтернативы, начиная с левой части. Поэтому в случае нахождения первой альтернативы правые части искаться не будут. Символов альтернативы можно применять сколько угодно раз.

## Группировка и ссылки (внутри и вне шаблона)

Символы можно группировать для того, чтобы применять к этой группе символы повторения, альтернативы и т.д. Такая группировка называется подмаской.

```
var r = /ab(cd)?/; //в строке будет найдено 'ab' и необязательное 'cd'
```

Кроме того, подмаски позволяют сослаться на результат поиска этой подмаски дальше внутри шаблона регулярного выражения. Подмасок внутри шаблона может быть сколько угодно.

Ссылка на результат поиска любой подмаски записывается с помощью символа "\" и ее номера. Номер подмаски определяется позицией ее левой скобки (первая скобка соответствует цифре 1). Подмаски могут находиться внутри других.

Например, требуется найти символы внутри одинарных или двойных кавычек, но обязательно одинаковых.

Очевидно, что следующее выражение не даст требуемый результат:

```
var r = /['"][^'"]*['"]/; //кавычки могут оказаться разными
```

Для этой задачи применяется подмаски

```
var r = /(['"])[^'"]*\1/; //ссылка \1 заменяется на первую найденную кавычку
```

Что еще более важно, подмаски позволяют использовать ссылки на найденные в них вхождения не только внутри шаблонов, но и вне шаблонов при использовании различных методов поиска и замены.

Если мы не хотим создавать ссылку для сгруппированных символов, следует применить (?:…). Этот способ группировки не создает ссылки на результат поиска подмаски ни внутри, ни вне шаблона.

**Символы группировки, альтернативы и ссылок**

| Символ | Что означает |
|---|---|
| \| | Альтернатива. Поиск будет осуществляться слева направо по указанным подвыражениям. |
| (…) | Группировка. К группе могут применяться символы повторения. Создает ссылку на результат поиска, который можно использовать как внутри, так и вне шаблона. |
| (?:…) | Только группировка. Ссылки на результат не создается. |
| \n | Соответствует символам, которые были найдены в результате поиска группы (подмаски) с номером n. |

## Позиции соответствия

В отличие от символов и классов символов в регулярных выражениях есть элементов, которые соответствуют позициям в тексте, а не символам строки.

Для этих элементов существуют обозначения. Некоторые из этих обозначений представляют собой управляющие последовательности.

**Символы позиций**

| Символ | Что означает |
|---|---|
| ^ | Поиск с начала строки |
| $ | Поиск до конца строки |
| \b | Позиция между символом ASCII и не символом ASCII (граница слова) |
| \B | Позиция между двумя символами ASCII (не граница слова) |
| (?=p) | Требование соответствия последующих символов шаблону **p**, но не включает эти символы в результат поиска |
| (?!p) | Требование несоответствия последующих символов шаблону **p** |

```
var r = /^abc/; //символы 'abc' ищутся только с начала строки

var r = /abc$/; //символы 'abc' ищутся только в конце строки

var r = /\bjava\b/; //ищется слово 'java', окруженное символами разделителей или переносов строки

var r = /\bjava\B/; //ищутся символы 'java', являющееся первой частью другого слова

var r = /java(?=\:)/; //ищутся символы 'java', после которых стоит ":"

var r = /java(?!\:)/; //ищутся символы 'java', после которых не стоит ":"
```

## Флаги

В регулярных выражениях существуют 3 флага, которые указывают на различные условия поиска.

### i (ignoreCase)

Указывает на нечувствительность поиска к регистру. Вместо перечисления символов в верхнем и нижнем регистре используется флаг i.

```
var r = /a/i; //ищутся символы 'a' и 'A'
```

### g (global)

При поиске ищутся все вхождения, а не только первое. Вместо организации циклов поиска по строке используется флаг i.

```
var r = /a/g; //ищутся все символы 'a'
```

### m (multiline)

Нужен при поиске по многострочным данным (например, по значению элемента формы textarea). В этом случае конец строки $ будет соответствовать концу последней строки.

Флаги могут использоваться в любом сочетании и порядке.

```
var r = /a/gim; //ищутся все символы 'a' и 'A' во всех строках
```

## Свойства экземпляров RegExp

У любого экземпляра RegExp есть 5 свойств, из которых только одно доступно для записи и чтения. Значения остальных свойств доступны только для чтения.

**Символы группировки, альтернативы и ссылок**

| Символ | Что означает |
|---|---|
| source | Текст регулярного выражения |
| ignoreCase | Наличие флага i – логическое значение |
| global | Наличие флага g – логическое значение |
| multiline | Наличие флага m – логическое значение |
| lastIndex | Номер позиции в строке, с которой начнется следующий поиск |

## Методы экземпляров RegExp и String для поиска с использованием шаблонов

Для поиска по шаблонам регулярных выражений существуют методы строк и самих экземпляров RegExp.

**Методы экземпляров String**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| replace(text1, text2) | *text1* – строка или шаблон регулярного выражения<br>*text2* – строка, на которую заменяем | Возвращает строку, в которой произведена замена первого аргумента на второй<br>В зависимости от флага регулярного выражения, первое вхождение шаблона, либо все они заменяются вторым аргументом-строкой<br>Во втором аргументе можно использовать ссылки на подмаски (группы) регулярного выражения с помощью строки '$n' (n – номер подмаски)<br>Поскольку метод возвращает строку, можно использовать подряд несколько вызовов метода |
| search(reg) | *reg* – шаблон регулярного выражения | Возвращает позицию первого символа найденного вхождения или -1 при отрицательном результате поиска<br>Игнорирует флаг *g* и выполняет поиск первого вхождения |
| match(reg) | *reg* – шаблон регулярного выражения | Если в шаблоне указан флаг *g*, то возвращается массив всех найденных вхождений<br>Если флага *g* нет, возвращается массив, в котором первый элемент – все найденное выражение, а в последующих элементах – подмаски выражения<br>Для отрицательного результата поиска в обоих случаях возвращается null |

| | | |
|---|---|---|
| split(reg) | *reg* – строка или шаблон регулярного выражения | Разбивает строку на массив подстрок, используя шаблон в качестве разделителя<br>Флаг *g* не используется |

У метода *match()* есть особенность, связанная с наличием у шаблона флага *g*.

Без флага *g* у возвращаемого массива есть свойства:

*index* – номер позиции первого символа вхождения

*input* – копия строки, в которой выполнялся поиск

**Методы экземпляров RegExp**

| Название | Аргументы | Что делает и возвращает |
|---|---|---|
| exec(s) | *s* – строка, в которой производится поиск | Возвращает массив из единственного вхождения (первого), но предоставляет о нем полную информацию<br>Для отрицательного результата поиска возвращается null |
| test(s) | *s* – строка, в которой производится поиск | Возвращает true в случае успешного поиска и false в обратном случае |

Оба метода регулярных выражений ведут себя отлично от методов строки *match()*, *replace()* и *search()*. Отличие состоит в текущем значении свойства шаблона *lastIndex* и зависит от наличия флага *g*.

Если флага *g* нет, *exec()* и *test()* сбрасывают после поиска это свойство в значение 0.

Если флаг *g* есть, то свойство выражения *lastIndex* равно номеру позиции символа, следующего за найденным вхождением.

Если одно и то же выражение с флагом *g* используется много раз для разных строк, нужно самостоятельно сбрасывать перед его применением *lastIndex* в 0.