Часть 1 (Предисловие)

Спецификация языка

Историческое

JavaScript – язык сценариев, разработанный Netscape Communications. Этот язык был разработан для оснащения веб- страниц функциональностью различных программ.

Спецификация

Первая версия JavaScript была разработана в Netscape Communications в 1995. По инициативе разработчика язык был стандартизирован в организации ECMA (European Computer Manufacturer's Association). С точки зрения спецификации язык называется ECMAScript, хотя его по прежнему называют JavaScript.
Рассматриваемая версия языка — ECMAScript 5th Edition.

Часть 2 (Pure JavaScript)

Pure JavaScript

Pure JavaScript (чистый JavaScript), должен быть рассмотрен отдельно от прочих средств. Это потому, что именно в таком виде JavaScript встраивается в различные приложения и (опционально) оснащается характерными им средствами.

Некоторые общие сведения

Текст сценария составляет собственно код на языке сценариев, и комментарии к нему.
Что касается комментариев — все точно так же, как в C++. Однако, таким способом они могут включаться только в код сценария, а не в код самого веб-документа.
Что касается собственно кода — код на языке JavaScript составляют его лексеммы с учетом формального синтаксиса.
Наиболее общие правила синтаксиса:
пользовательские лексеммы не должны быть омографичны языковым лексеммам;
запись языковых лексемм регистрозависима;
код сценария допускает верстку средствами текстовых редакторов (пробельными символами), но не текстовых процессоров;

Подключение сценариев к документу

JavaScript интегрирован в HTML, то есть его код можно встроить прямо в веб- документ. Так же, код сценария можно разместить и в отдельном файле. Так или иначе, подключение кода сценария осуществляется средствами HTML.
Тег <script> позволяет подключить сценарий либо путем включения его кода в контейнер этого тега, либо из внешнего файла — посредством атрибута src (в этом случае контейнер должен быть пустым). Атрибут type тега <script> принимает значение «text/javascript», атрибут language тега <script> принимает значение «JavaScript».
При подключении сценариев их загрузка и исполнение происходит в том же порядке, в котором прописаны теги <script>. Об этом принято говорить, что сценарии загружаются синхронно с документом. Чтобы сценарии загружались асинхронно с документом (то есть, сначала документ, а потом сценарии) применяют один из следующих атрибутов элемента <script>:

атрибут `defer` (не имеет значений) откладывает загрузку сценариев до завершения загрузки документа (при этом порядок следования сценариев сохраняется);
атрибут `async` (не имеет значений) откладывает загрузку сценариев до завершения загрузки документа (при этом порядок следования сценариев не обязательно сохраняется — они загружаются асинхронно как с документом, так и друг с другом);

Эти атрибуты срабатывают только при подключении сценариев из внешнего файла. Подключение сценариев из внешнего файла возможно так же при помощи элемента ссылки:

```
<a src="javascript: … ">
<!-- … это код сценария-->
```

В значении `src` прописывается псевдопротокол и текст сценаря за ним.
Подключение сценариев из кода документа возможно так же при помощи так называемых событий. События прописываются как атрибуты элементов (различным элементам доступны характерные им события), а в значениях этих атрибутов прописывается код сценария (исполняемый при наступлении события). В записи это выглядит так:

```
<ElName AttrName=" … ">
<!-- … это код сценария-->
```

Об именах

Идентификаторы

Так же, как в `C++`.

Области видимости имен

Области видимости имен в языке `JavaScript` подразделяются на глобальную область видимости (она глобальна для всего сценария), и локальные области видимости (они локализованы телами функций).
Сразу же стоит оговорить, что блоки кода в языке `JavaScript` не создают областей видимости.

Блоки кода

Блок кода в языке `JavaScript` это участок кода, ограниченный фигурными скобками. Блок кода служит для следующей цели:

он группирует заключенный в него код в блок — это оказывается важным при использовании различных средств языка (которые требуют, чтобы участок кода был заключен в блок).
В языке JavaScript обычный блок кода не обозначает собой область видимости имен — область видимости имен обозначает тело функции.

О переменных

Идентично тому, как в C++. Разве что, имеют место некоторые различия.

Объявление переменных

Будучи объявлены в некой области видимости, переменные в ней и локализуются.
В случае совпадения имен в глобальной и локальной областях видимости, в локальной области видимости работа будет происходить с локальной переменной. Об этом случае принято говорить, что локальная переменная маскирует глобальную.
В языке JavaScript переменные должны быть объявлены до их использования. Запись объявления переменных выглядит так:

```
var VarName;
```

Переменные можно объявить и списком:

```
var VarName0, … , VarNameN;
```

Если же обратиться к еще не объявленной переменной, то ситуация развивается следующим образом. При обращении по чтению (к еще не объявленной переменной) возникнет ошибка. При обращении по записи (к еще не объявленной переменной) эта переменная будет объявлена неявно, самим языком (причем, она будет объявлена в глобальной области видимости). И затем произойдет доступ к ней по записи. Далее она будет доступна для любого использования.
Переменные, объявленные при помощи оператора var принято называть долговременными. Долговременные переменные не могут быть удалены пользователем.

Инициализация переменных

При объявлении переменную можно инициализировать:

```
var VarName = … ;
```

Разумеется, инициализировать можно и после объявления:

```
VarName = … ;
```

Допустимо и множественное присваивание:
`VarName0 = … = VarNameN;`
И составное присваивание (обычной для этого записью).

Элементарные типы данных

`JavaScript` – нетипизированный (слабо типизированный язык). Поэтому в языке `JavaScript` имеются следующие особенности:
тип данных при объявлении переменных не указывается;
в выражениях не требуется явного приведения типов, как и соответствия типов;
все типы данных (и не только они) приводимы друг к другу;

Однако, это не значит, что в языке `JavaScript` типов данных нет вообще.
Язык `JavaScript` поддерживает следующие типы данных.

Числа

все числа представляются в интерпретаторе значениями с плавающей точкой двойной точности (независимо от того, в каком виде они прописаны в сценарии);
к числам относятся так же специальные численные значения:
`Infinity, -Infinity` (бесконечность, возникает при переполнении сверху и снизу, и при разрешении неопределенности «число дробь ноль»), `NaN` (не-число, несмотря на название это численное значение, возникает при разрешении прочих математических неопределенностей и при исполнении математических операторов с ошибкой);

Строки

строки считаются элементарными типами данных;
все строки представляются в интерпретаторе в кодировке Unicode;
строки предназначены для хранения секвенций символов с длиной полезной нагрузки в один символ и более (отдельного типа данных для символов нет);
строки задаются последовательностью символов в одинарных или двойных кавычках;
также строкам доступен синтаксис объектов (конструкторы и прочее), хотя они и считаются элементарными типами данных (это благодаря тому, что есть объекты- обертки строк);
строки не изменяемы (как и в `Java`);

Логические значения

логические операнды, как обычно, принимают значения `true` и `false`; и, как обычно, говорить о размере операндов этого типа не целесообразно;

`null`

это специальное значение `null`;

`undefinied`

это специальное значение `undefinied`;

Указатели

Указатели в принципе не поддерживаются языком `JavaScript`.

Составные типы данных

Составные типы данных в принципе не поддерживаются языком `JavaScript`, так как собственно структур в нем нет, массивы — это объекты `Array`, а строки (официально) считаются элементарными типами данных.

О типизации

`JavaScript` – нетипизированный язык. Поэтому все типы данных (и не только они) приводимы друг к другу.
Приведение типов производится неявно. Необходимость в этом возникает в тех случаях, когда применяются операторы, работающие в конечном счете с тем или иным типом.
Таким образом, приведение типов выполняется взависимости от потребности в том или ином типе. Или, как принято об этом говорить, «приведение типов выполняется взависимости от того, какой тип требуется в контексте исполнения кода» (или же, «в каком контексте используется операнд»).
Контекст использования операнда определяется использующим его оператором (различные операторы работают в конечном счете с определенным типом операндов). Если же оператор может работать с разными типами, то для определения контекста может быть важен порядок операндов.
Если интерпретатору не удается сразу привести тип операнда к целевому, то он пытается повторно и по другому алгоритму.

Приведение типов в соответствии с контекстом

`Числа`

при использовании в строковом контексте приводится к строке (вида `«123»`);

при использовании в логическом контексте приводится к логическому значению (значению `true` или `false`, взависимости от значения (значения `0` приводится к значению `false`, все остальные значения приводятся к `true`));
при использовании в объектном контексте приводится к объекту (`Number`);

Строки

при использовании в числовом контексте приводится к числу (если содержит литерал числа — то к его значению, если пустая — то к значению `0`, иначе — к значению `NaN`);
при использовании в логическом контексте приводится к логическому значению (значению `true` или `false`, взависимости от значения (взависимости от того, пустая эта строка или же не пустая));
при использовании в объектном контексте приводится к объекту (`String`);

Логические значения

при использовании в строковом контексте приводится к строке (строке «`true`» или «`false`», взависимости от значения);
при использовании в числовом контексте приводится к числу (числу `1` или `0`, взависимости от значения);
при использовании в объектном контексте приводится к объекту (`Boolean`);

null

при использовании в строковом контексте приводится к строке (строке «`null`»);
при использовании в числовом контексте приводится к числу (числу `0`);
при использовании в логическом контексте приводится к логическому значению (значению `false`);
при использовании в объектном контексте приводится к объекту (`Error`);

undefinied

при использовании в строковом контексте приводится к строке (строке «`undefinied`»);
при использовании в числовом контексте приводится к числу (численному значению `NaN`);
при использовании в логическом контексте приводится к логическому значению (значению `false`);
при использовании в объектном контексте приводится к объекту (объекту `Error`);

Объекты

Различные языковые и пользовательские объекты приводятся к другим типам при помощи их методов (подробности — далее в этих записях);
если объект содержит методы приведения к целевым типам, то они и вызываются при необходимости приведения к соответствующим типам (разумеется, это касается языковых объектов — о специфике пользовательских объектов реализация языка знать не может);

Операторы

Оператор, как обычно, это запись некой конечной операции. Такая запись должна завершаться точкой с запятой.

Примечание

Интерпретатор автоматически учитывает наличие точки с запятой (в конце строк) там, где она предполагается (даже если не прописана явно). Интерпретатор предполагает, что точка с запятой в конце строки уместна, если запись данной строки может трактоваться как запись оператора.
Так, если операторы прописаны в секвенции строк (и точка с запятой в их концах опущена), то интерпретатор просто учитывает наличие точки с запятой в концах этих строк. Если же операторы прописаны в одной строке (и точка с запятой в концах этих операторов опущена), то будет обнаружена ошибка.
Необходимо помнить, что интерпретатор делает предположение о необходимости точки с запятой только для конца строки, а не другого ее участка.

Составление выражений

Приоритет операторов в выражении можно задать расстановкой скобок.

Оператор « , »

Оператор « , » реализован так же, как в C++.
В языке JavaScript имеется и тернарный оператор, он записывается так же, как в C++.

Оператор void

Оператор void служит для применения совместно с выражениями (и является средством для работы с выражениями).

Он работает так:
исполняет указанное выражение;
но вместо его значения возвращает `undefinied`;

Таким образом, этот оператор используется для того, чтобы указанное выражение вычислялось в определенном участке кода, но его значение не возвращалось в него.

Арифметические операторы

Арифметические операторы задаются теми же знаками, что и в языке `C++`.
При использовании арифметических операторов необходимо помнить:
оператор деления выполняет нецелочисленное деление (это логично, поскольку все числа представляются интерпретатору в нецелочисленном виде);
оператор деления по модулю применим ко всем числам — заданным в целочисленном или нецелочисленном виде (это логично, поскольку все числа представляются интерпретатору в нецелочисленном виде); этот оператор возвращает остаток от деления в целочисленном или нецелочисленном виде (для целочисленных он вычисляет остаток так, как если бы деление было целочисленным; для нецелочисленных он вычисляет остаток похожим образом — так, чтобы не потребовалось увеличивать количество знаков после запятой);

О типизации

Арифметические операторы используют свои операнды предпочтительно в числовом контексте. Исключением является оператор суммы. Если один из его операндов — строка, то он использует свои операнды предпочтительно в строковом контексте.

Логические операторы

Логические операторы задаются теми же знаками, что и в языке `C++`. Однако, некоторые из них работают иначе:

`!`	NOT
`&&`	CONDITIONAL AND
`||`	CONDITIONAL OR

Операторы `&&` и `||` имеются только в условных вариантах. Условные логические операторы (CONDITIONAL AND, CONDITIONAL OR) исполняются условно — они исполняются при условии истинности первого операнда, иначе же они просто возвращают его значение.
Для операции XOR не предусмотрено отдельного знака, эта операция задается эквивалентной последовательностью операций.

О типизации

Логические операторы используют свои операнды предпочтительно в логическом контексте.

Побитовые операторы

Побитовые операторы задаются теми же знаками, что в `Java`.

Операторы сравнения

Операторы сравнения задаются теми же знаками, что в `C++`. Вдобавок, имеется оператор идентичности:

`===` IDENTICALLY
`!==` NOT IDENTICALLY

Идентично значит равно по значению и однотипно.
При использовании операторов сравнения необходимо помнить следующее:
численное значение `NaN` не сравнимо ни с какими- либо другими значениями, ни само с собой;
специальное значение `null` сравнимо с другими значениями и само с собой;
специальное значение `undefinied` сравнимо с другими значениями и собой;
специальные значения `null` и `undefinied` сравнимы друг с другом и равны (но не идентичны);

О типизации

Операторы сравнения используют свои операнды предпочтительно в числовом контексте.

Приоритет операторов, задаваемых знаками

Как и во всех `C-` образных языках.

Операторы управления ходом вычислений

Идентично тому, как в `C++`. Разве что, имеют место некоторые различия. Всилу нестрогой типизации языка `JavaScript` эти операторы могут использовать переменные разных типов.

Оператор goto

Оператор `goto` отсутствует в языке `JavaScript`. Разработчик языка аргументирует это борьбой с макаронным кодом.

Операторы управления итерацией

Операторы `break` и `continue` реализованы идентично тому, как в `Java`.

Операторы цикла

Идентично тому, как в `Java`. Разве что, имеют место некоторые различия.
Усовершенствованный цикл `for` в записи выглядит иначе:

```
for (VarName in CollectionName) … ;
```

Переменная в условии цикла осуществляет перебор заданной коллекции (например, массива) — она последовательно принимает имена (не значения) элементов коллекции (для элементов массива это индекс). Таким образом, переменную цикла можно использовать для доступа к элементам коллекции. О перечислении элементов коллекции необходимо помнить следующее — некоторые элементы являются перечислимыми, а некоторые — нет (потому что все коллекции — это объекты, и не все их члены являются перечислимыми). Пользовательские члены объектов по умолчанию перечислимы. Языковые члены объектов по умолчанию перечислимы (члены данных). Это касается как собственных членов объектов, так и членов их прототипов (о прототипах — в соответствующей главе). Что касается перечисления элементов массивов — то усовершенствованный цикл без проблем осуществляет их перечисление. А необходимость фильтрации членов (не являющихся элементами массива) возникает только тогда, когда такие члены приписаны программистом. Что касается порядка перечисления — то он всегда остается на откуп реализации языка. Хотя, классическим считается следующий порядок — члены перечисляются в порядке их следования в коде сценария; причем, сначала идут собственные члены объектов, затем члены их прототипов; затем члены их прототипов (то есть, более высоких уровней иерархии их прототипов).
К элементу коллекции можно обратиться по имени следующей записью:

```
CollectionName[VarName]
// VarName – переменная цикла
```

Остальные операторы цикла реализованы без различий.

Операторы выбора или условия

Операторы `if` и `switch` реализованы идентично тому, как в языке `C++`. Однако, многие интерпретаторы привносят в интерпретацию оператора `switch` следующие особенности:

все его ветви должны завершаться оператором `break`, иначе же тела всех его ветвей будут исполнены (причем безусловно);
переменная цикла сравнивается со значениями из ветвей как при применении оператора `===`, а не `==`;

О типизации

Операторы управления ходом вычислений используют свои операнды предпочтительно в логическом и числовом контексте.

Оператор with

Оператор `with` применяется совместно с операторами управления ходом вычилений.
Этот оператор служит для изменения области видимости (на время исполнения целевого оператора управления ходом вычислений). Он работает так:
указанный в его операнде объект делает верхом иерархии областей видимости;
исполняет целевой оператор управления ходом вычислений;
восстанавливает область видимости до прежнего вида;

Этот оператор прописывается так:

```
with(ObjectName.NestedObjectName)
 … ; // … - целевой оператор
```

Как видно из записи, верхом иерархии областей видимости можно задать как объект, так и секвенцию вложенных объектов.
С одной стороны, этот оператор упрощает написание кода сценария (так как делает доступ к членам указанного объекта непосредственным); с другой стороны — усложняет его интерпретацию (так как требует от интерпретатора соответствующих действий). При использовании этого оператора необходимо помнить, что реализации языка `JavaScript` привносят в интерпретацию этого оператора свою специфику. Использование этого оператора не рекомендуется.

Функции

Функции в языке `JavaScript` тесно связаны с объектами, и поэтому рассматриваются в соответствующей главе.

Часть 3 (ООП)

О ссылочных типах

Говоря о языке JavaScript приходится оперировать таким понятием, как ссылочные типы.
Хотя в этом языке указателей (и в том числе ссылок) как таковых нет, в нем есть аналогичное (но не идентичное) средство — ссылочные типы.
Так называемые ссылочные типы — это объекты Object, и прочие объекты, и функции (для работы с функциями языком поставляется специальный объект — объект функции). Строки не являются ссылочными типами (хотя им и доступен объектный синтаксис, в остальном с ними нельзя работать как со ссылочными типами). Ссылочные типы называются так потому, что имена их экземпляров используются для ссылки на них.
Все ссылочные типы есть производные от объекта Object.

Классы

Язык JavaScript обладает специфическими средствами для работы с классами и объектами. Строго говоря, слово «классы» здесь употреблено лишь по традиции — в языке JavaScript классов, как таковых нет; а объекты — это не экземпляры классов, а производные их прототипов.

Объекты

Объекты это производные их прототипов (прототипы будут рассмотрены соответствующей главе).
В языке JavaScript есть различные объекты. Это объекты Object (объекты наиболее общего назначения), и прочие объекты (объекты специализированного назначения).
В языке JavaScript все находящееся в глобальной области видимости является инкапсулированным в так называемый глобальный объект. Эта инкапсуляция производится неявно, самим языком, и в нуждах языка. Глобальный объект принято называть Global — в Pure JavaScript (при этом, необходимо помнить, что его имя — специфика приложения, в которое встраивается интерпретатор), и Window – в Client- side JavaScript (это имя таково во всех браузерах).
В первую очередь стоит рассмотреть объекты Object. Остальные объекты идентичны им (разве что, привносят свою специфику). Но не стоит считать, что все объекты полностью реализуют возможности объектов Object. Несмотря на наличие прототипов и «прототипного наследования» в языке JavaScript, не

все возможности `Object` реализованы в остальных языковых объектах (такое специфическое поведение организовано самим языком). Общими (по оф. т.з.) для них являются лишь следующие возможности (лучше сказать, члены) — `valueOf()`, `toString()`, `toLocaleString()`, `toJSON()` (причем, этот член существует лишь в `Pure JavaScript`). Впрочем, даже эти члены (лучше сказать, методы) могут быть переопределены в различных объектах, и реализовать свою логику. Какие именно есть члены и в каких объектах — можно уточнить в справочниках (и далее в этих записях).

Объекты Object

Объекты `Object` создаются записью:
```
var ObjectName = new Object();
```

Можно создать и с инициализацией:
```
var ObjectName = {VarName:Value, … }
```

В этой записи имя члена объекта может быть задано и строкой, и тогда запись принимает вид:
```
var ObjectName = {«VarName»:Value, … }
```

Помимо членов объекты имеют и атрибуты. Это следующие:
`prototype` – служит ссылкой на прототип;
`class` – содержит строку с именем «класса» объекта;
`extensible` – флаг, определяющий возможность приписывания новых свойств;

Члены объектов также имеют атрибуты. Это следующие:
`writable` – задает доступ к члену объекта по записи;
`enumerable` – задает доступность члена объекта для перечисления усовершенствованным циклом `for`;
`configurable` – задает возможность удаления члена объекта и изменения атрибутов;

Доступ к атрибутам объектов и их членов осуществляется при помощи соответствующих методов. Некоторые эти методы используют дескриптор членов объекта. Дескриптор членов объекта — это специальный объект, описывающий их атрибуты. Вид дескриптора членов с данными `{value, writable, enumerable, configurable}`; дескриптор членов с методами доступа имеет вид `{get, set, enumerable, configurable}`, (о членах с методами доступа — в следующем параграфе). Когда какому- либо методу нужно передать только определенные составляющие дескриптора, то можно указывать только их (и тогда запись дескриптора примет не полный вид). Когда методу передается дескриптор, то необходимо помнить следующее: значение атрибута `configurable` члена объекта может быть изменено только на более строгое.

Остальные же атрибуты члена объекта могут менять свое значение как на более строгое, так и на менее строгое. Кроме того, необходимо помнить следующее: значение атрибута `extensible` самого объекта тоже может быть изменено только на более строгое. Значение атрибута `class` самого объекта не может быть изменено программистом ни прямо, ни косвенно. Значение атрибута `prototype` может быть изменено программистом косвенно, посредством одноименного члена объекта (члена `ObjectName.prototype`).
В дальнейшем переменная `ObjectName` может быть реинициализирована ссылкой на другой объект.
Можно и вообще разорвать связь переменной `ObjectName` с каким- либо объектом. Для этого ей нужно присвоить специальное значение `null`.

Доступ к членам объектов

Доступ к членам объекта извне этого объекта производится через оператор доступа к членам объекта (через точку). Доступ к членам объекта изнутри этого объекта обходится и без оператора доступа к членам объекта.
Доступ к членам объекта возможен и посредством индексации:

```
ObjectName[«VarName»];
```

В этой записи `«VarName»` это строка с именем члена объекта.
Доступ к членам объекта возможен и для объявления — объекту могут быть приписаны новые члены. Это делается записью:

```
ObjectName.VarName = … ;
/* как видно из записи,
это похоже на инициализацию членов,
которые уже существуют */
```

Это же возможно и при помощи индексации:

```
ObjectName[«VarName»] = … ;
/* как видно из записи,
это похоже на инициализацию членов,
которые уже существуют */
```

Объекту могут быть приписаны как члены данных, так и методы, и объекты.
Доступ к члену объекта может осуществляться и при помощи методов доступа. В языке `JavaScript` под методами доступа имеются ввиду не просто какие- либо методы, осуществляющие доступ, а методы, объявленные специфической записью, и вызываемые неявно при доступе к членам.
Если у каких- либо членов нет методов доступа, то они называются членами с данными. Если же есть — то они называются членами с методами доступа. У члена объекта могут быть предусмотрены методы доступа по чтению и записи, либо один из них.
Методы доступа могут быть любыми (в плане их тела), но они должны позволять работать корректно. Методы доступа по чтению должны возвращать значение считываемого члена, и не принимать параметров. Методы доступа по записи

должны принимать параметр — записываемое значение, и не возвращать значений. Этим и обеспечивается их корректная работа.
Методы доступа объявляются специфической записью в объявлении соответствующих членов. Можно сказать, что сами эти члены объявляются в специфической записи:

```
get VarName() { … },
set VarName( … ) { … }
// VarName – имя члена
```

Так объявляются члены, для которых предусмотрены оба метода доступа. Сами эти методы объявляются именно в такой форме.

Операции над объектами

Оператор присваивания применим к объектам. Причем, нестрогая типизация языка JavaScript позволяет присваивать переменной (служащей ссылкой на объект) значения произвольного типа.
Операторы сравнения также применимы к объектам (хотя, для них имеют смысл только проверки на равенство\неравенство и идентичность\неидентичность).

Оператор delete

Оператор delete служит для удаления членов объекта.
Он записывается так:

```
delete ObjectName.VarName;
```

При использовании такой записи будет удален член самого объекта (а не прототипа).
Для удаления члена прототипа служит запись:

```
delete ObjectName.prototype.VarName;
```

Удалены могут быть только пользовательские члены объектов и прототипов; языковые члены удалены быть не могут, как и долговременные переменные.
Этот оператор возвращает значение true, в случае:
если удаление прошло успешно;
если запрошено удаление несуществующего свойства;
если в операнде прописано выражение, которое не указывает никакого члена;

Иначе же, этот оператор возвращает значение false.

Оператор in

Оператор in служит для проверки наличия указанного члена. То есть, лексемма in может применяться не только в усовершенствованном цикле for.
Оператор in записываеся так:

```
«VarName» in ObjectName
```

```
// именно так
```

Этот оператор возвращает значение `true`, если указанный член имеется (и не важно, у самого объекта, или его прототипа); иначе же он возвращает `false`.

Оператор instanceof

Оператор `instanceof` служит для проверки того, является ли указанный объект объектом указанного типа. Он записывается так:

```
ObjectName instanceof Object; // Object, etc
```

Этот оператор возвращает значение `true`, если указанный объект является объектом указанного типа (необходимо помнить, что все объекты есть производные от объекта `Object`, так что принадлежность к нему безусловно подтверждается), иначе же — `false`.

Оператор typeof

Оператор `typeof` служит для определения типа его операнда. Его запись:

```
typeof VarName
/* альтернативная запись: typeof (VarName) */
```

Этот оператор возвращает строку:

«undefinied»	для операнда типа `undefinied`;
«object»	для операнда типа `null`;
«boolean»	для операнда логического типа;
«number»	для операнда численного типа;
«string»	для операнда строкового типа;
«function»	для операнда — функции;
«object»	для операнда — объекта;
etc	для операнда — объекта (сп. сред. исп.);

Для объекта, специфичного среде исполнения (то есть, приложению, в которое встраивается интерпретатор) этот оператор может возвращать строку другого содержания (то есть, не одного из вышеописанных вариантов).
Для специального значения `null` этот оператор возвращает строку «object», хотя это значение и не является объектом. Подробнее о специальных значениях `null` и `undefinied` – в следующем параграфе.

О специальных значениях null, undefinied

Специальные значения `null` и `undefinied` трактуются специфически.
Оба эти значения считаются значениями элементарных типов данных (не стоит обманываться тем, что оператор `typeof` вовращает для значений `null` строку

«object» – это происходит исключительно по причине специфики интерпретации этого значения). Эти значения действительно являются значениями элементарных типов данных; и более того, соответствующих объектов- оберток не имеют.
Специальное значение `undefinied` используется языком для того, чтобы обозначить отсутствие у какой- либо переменной значения инициализации. В том числе, при возврате из функции ничтоже, при определении значения не переданных аргументов, при считывании пустого элемента массива, и при считывании не приписанного объекту члена.
Специальное значение `null` используется языком для того, чтобы обозначить отсутствие переменной вообще. В том числе, при разрыве связи ссылки на объект с объектом.

О конструкторах

Конструкторы — это средство языка, предназначенное для создания объектов. Именно объектов, так как классов (как таковых) в этом языке нет.
В языке `JavaScript` конструкторы обладают следующей спецификой. Они не создают объектов, а просто инициализируют собственные члены объектов (обычной для этого записью: `this.VarName = … ,`). Объекты же создаются оператором `new`. Таким образом, конструкторы и не должны проделывать всю работу по созданию объектов. Поэтому в качестве конструкторов могут применяться любые функции (и даже имя их может быть любым).
Конструкторы объектов, объекты, и их прототипы (о них — в соответствующей главе) взаимосвязаны — они ссылаются друг на друга следующим образом. Объект ссылается на свой прототип (`ObjectName.prototype`). Прототип ссылается на функцию- конструктор (`PrototypeName.constructor`). Функция конструктор ссылается на прототип инициализируемых ей объектов (`FuncName.prototype`).
Итак, конструкторы могут быть как языковыми, так и пользовательскими. Пользовательские конструкторы (с официальной точки зрения) должны применяться так:
создается прототип пользовательских объектов;
создается пользовательская функция, которую предполагается применять как конструктор пользовательских объектов;
ей приписывается ссылка на объект- прототип;
и тогда объект- прототип автоматически получит ссылку на эту функцию (`PrototypeName.constructor`);

Необходимо помнить, что хотя прототипом пользовательского объекта можно назначить целевой объект (тем самым, косвенно изменив атрибут `prototype` этого объекта), но атрибут `class` этого объекта программист вообще не может изменить — ни прямо, ни косвенно. Поэтому, все пользовательские объекты — это, по большому счету, объекты `Object`.

О прототипах

Прототип — это какой-либо объект, используемый как прототип для каких- либо объектов. Он используется интерпретатором для конструирования объектов — объекты конструируются по их прототипам (это следует понимать так — прототипом объекта назначается соответствующий объект).
Прототип используется для хранения общей (для всех производных от него объектов) информации. В этом он похож на статический класс в `C++`.
Программист имеет доступ к прототипу. Для доступа к прототипу служит соответствующий член объекта — это член `ObjectName.prototype`.
Прототип объектов `Object` прототипа не имеет. Однако, прототипы других объектов (не `Object`) имеют прототип — им служит прототип объектов `Object`.

Прототипы и доступ к членам объектов

Доступ к членам объекта происходит следующим образом:
этот член ищется в объекте — если он в нем есть, то происходит доступ;
если нет, то он ищется в прототипе этого объекта — если он в нем есть, то происходит доступ;
если нет — то он ищется в прототипе прототипа (в случае его наличия) — и так итеративно, до нахождения целевого члена, либо прототипа без прототипа (и тогда член ищется в нем);
если целевой член есть — то происходит доступ;
если нет — то происходит возврат специального значения — это при попытке доступа не по записи, или же происходит объявление члена в этом объекте (причем, не в прототипе) — это при попытке доступа по записи;

Маскировка членов прототипа

В том случае, когда к члену объекта обращаются по объявлению, и при этом он есть в прототипе, происходит следующее:
этот член объявляется в объекте (причем, не в прототипе);
в дальнейшем, доступ происходит к члену самого объекта — об этом принято говорить, что член объекта маскирует член прототипа;

Таким образом, прототипы позволяют эффективно использовать основную память компьютера. Если при доступе к члену объекта достаточно информации (общей для всех таких объектов), то доступ происходит к члену самого прототипа — это организуется неявно, самим языком. Если же при доступе к члену объекта требуется информация (специфичная для конкретного объекта), то только тогда происходит доступ к члену самого объекта. То есть, члены объектов до обращения к ним по записи на самом деле хранятся в прототипе. И лишь после обращения к ним по записи они заносятся и в объекты (и тогда член

объекта маскирует член прототипа). Разумеется, речь идет о членах, которые не были приписаны объекту (его конструктором, либо программистом вручную).

Члены объектов Object

Конструктор объектов Object

`Object()`

конструктор объектов `Object`; параметров либо не принимает, либо принимает значение элементарных типов данных (имеющих соответствующие им обертки); создает либо объект `Object`, либо объект- обертку; возвращает объект `Object`, либо объект- обертку;

Методы объектов Object

`toString()`

используется для получения строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержащую `[object Object]`);

`toLocaleString()`

используется для получения строкового представления своего объекта, причем в локализованном (согласно пользовательским региональным настройкам) виде; однако, в действительности это просто синоним вышеописанного метода;

`valueOf()`

используется для возврата значения своего объекта; аргументов не принимает; возвращает значение своего объекта (сам свой объект); возвращает `Object`;

`create()`

используется для создания объекта по указанному прототипу; принимает параметры — объект (указатель на прототип), и объект инкапсулирующий переменные целевого объекта (они инициализируются целевым значением их дескриптора) — запись передачи параметров имеет вид `(ObjectName`,

`{VarName: { … }, … })`; создает объект указанным образом; возвращает объект (это созданный объект);
этот метод «статический»;

`preventExtensions()`

используется для предотвращения изменения указанного объекта; принимает параметр — указанный объект; делает указанный объект нерасширяемым; возвращает объект (это указанный объект, претерпевший изменения);
этот метод «статический»;

`isExtensible()`

используется для проверки указанного объекта на расширяемость; принимает параметр — указанный объект; проверяет указанный объект на расширяемость; возвращает логическое значение (true – если расширяем, false – если нет);
этот метод «статический»;

`seal()`

используется «опечатывания» указанного объекта; принимает параметр — указанный объект; делает указанный объект нерасширяемым, а его собственные члены ненастраиваемыми; возвращает объект (это указанный объект, претерпевший изменения);
этот метод «статический»;

`isSealed()`

используется для проверки «опечатанности» указанного объекта; принимает параметр — указанный объект; проверяет, нерасширяем ли указанный объект, и ненастраиваемы ли его собственные члены; возвращает логическое значение (`true` – если «опечатан», `false` – если нет);
этот метод «статический»;

`freeze()`

используется для «заморозки» указанного объекта; принимает параметр — указанный объект; делает указанный объект нерасширяемым, все его собственные члены ненастраиваемыми, и нереинициализируемыми (последнее касается членов с данными); возвращает объект (это указанный объект, претерпевший изменения);
этот метод «статический»;

`isFrozen()`

используется для проверки указанного объекта на «замороженность»; принимает параметр — указанный объект; проверяет, нерасширяем ли указанный объект, и приэтом ненастраиваемы ли его собственные члены, и нереинициализируемы ли его собственные члены с данными; возвращает логическое значение (true – если «заморожен», false – если нет);
этот метод «статический»;

`getPrototypeOf()`

используется для возврата прототипа указанного объекта; принимает параметр — указанный объект; возвращает прототип указанного объекта; возвращает объект (это прототип указанного объекта);
этот метод «статический»;

`isPrototypeOf()`

используется для проверки, является ли свой объект прототипом указанного объекта; принимает параметр — объект; проверяет, является ли свой объект прототипом указанного объекта (прототипом из иерархии прототипов, и необязательно непосредственным); возвращает логическое значение (`true` – если да, `false` – если нет);

`defineProperty()`

используется для определения собственного члена указанного объекта; принимает параметры — объект (указатель на целевой объект), строку (имя целевого члена), и объект (дескриптор целевого члена); определяет (то есть, либо приписывает новый, либо изменяет существующий член) целевой собственный член объекта указанным образом; возвращает объект (это указанный объект, претерпевший изменения);
этот метод «статический»;

`defineProperties()`

используется для определения ряда собственных членов объекта; принимает параметры — объект (указатель на целевой объект), и объект который инкапсулирует переменные целевого объекта (они инициализируются целевым значением их дескриптора) — запись передачи параметров имеет вид `(ObjectName, {VarName: { … }, … })`; определяет (то есть, либо приписывает новые, либо изменяет существующие члены) ряд собственных

членов указанным образом; возвращает объект (это указанный объект, претерпевший изменения);
этот метод «статический»;

`getOwnPropertyDescriptor()`

используется для возврата дескриптора собственного члена объекта (дескриптор члена — это специальный объект, описывающий его; дескриптор членов объекта имеет характерный ему вид; принимает аргументы — объект (указатель на целевой объект), и строку (имя целевого члена объекта); возвращает дескриптор целевого члена объекта; возвращает объект (запрошенный дескриптор);
этот метод «статический»;

`getOwnPropertyNames()`

используется для возврата имен собственных членов указанного объекта (включая имена неперечислимых); принимает параметр — указанный объект; возвращает имена собственных членов указанного объекта (включая имена неперечислимых); возвращает массив строк (с именами членов);
этот метод «статический»;

`keys()`

используется для возврата имен собственных членов указанного объекта (исключая имена неперечислимых); принимает параметр — указанный объект; возвращает имена собственных членов указанного объекта (исключая имена неперечислимых); возвращает массив строк (с именами членов);
этот метод «статический»;

`hasOwnProperty()`

используется для проверки наличия у объекта собственного члена; принимает параметр — строку с именем члена; проверяет наличие у своего объекта указанного члена (как собственного, не унаследованного члена); возвращает логическое значение (`true` – если есть, `false` – если нет);
`propertyIsEnumerable()`

используется для проверки собственного члена на перечислимость; принимает параметр — строку с именем члена; проверяет указанный член своего объекта на перечислимость (проверка исполняется только для собственных, неунаследованных членов); возвращает логическое значение (`true` – если это собственный член и он перечислим, `false` – если нет);

Переменные объектов Object

`prototype`

используется для ссылки на прототип; содержит член — `prototype.constructor` (используется для ссылки на конструктор);

Объекты- обертки

Объекты- обертки — это объекты, поддерживаемые языком для инкапсулляции значений э.т.д. Что, в свою очередь, служит для возможности объектного представления значений элементарных типов данных. В целях беспрепятственного использования этой возможности языком языком поддерживается автоматическое преобразование между обертками и соответствующими им значениями (непосредственными значениями и переменными) элементарных типов данных. Это преобразование происходит в выражениях в случае необходимости, и производится неявно, самим языком.
Языком поддерживаются следующие обертки:

```
Boolean для логических значений
Number  для численных значений
String  для строковых значений
```

Ввиду применения объектов- оберток обнаруживается следующая особенность — для создания объекта- обертки нет необходимости создавать его явно. Они и так создаются при необходимости в этом.

Члены объектов Boolean

Конструктор объектов Boolean

`Boolean()`

конструктор объектов `Boolean`; принимает параметр — значение (этим значением будет инициализирован созданный объект); создает объект `Boolean`; возвращает объект `Boolean`;

Методы объектов Boolean

`toString()`

используется для возврата строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта

(лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержащую `true` или `false`);

`valueOf()`

используется для возврата значения своего объекта; аргументов не принимает; возвращает значение своего объекта (значение элементарного типа данных, которое инкапсуллирует); возвращает логическое значение;

Переменные объектов Boolean

Их нет.

Члены объектов Number

Конструктор объектов Number

`Number()`

конструктор объектов `Number`; принимает параметр — значение (этим значением будет инициализирован созданный объект); создает объект `Number`; возвращает объект `Number`;

Методы объектов Number

`toString()`

используется для возврата строкового представления своего объекта; параметров либо не принимает, либо принимает число (используется как основание системы счисления, в которой будет представлен результат); возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержащую инкапсуллируемое значение);

`toLocaleString()`

используется для получения строкового представления своего объекта, причем в локализованном (согласно пользовательским региональным настройкам) виде; однако, в действительности это просто синоним вышеописанного метода;

`valueOf()`

используется для возврата значения своего объекта; аргументов не принимает; возвращает значение своего объекта (значение элементарного типа данных, которое инкапсуллирует); возвращает число (значение элементарного типа);

`toFixed()`

используется для возврата инкапсуллируемого числа в виде с фиксированной точкой; параметров либо не принимает, либо принимает число (используется как количество разрядов в дробной части, если этот параметр не передан то он считается равным нулю); возвращает строковое представление инкапсуллируемого числа в соответствующем виде; возвращает строку (содержащую инкапсуллируемое число);

`toExponential()`

используется для возврата инкапсуллируемого числа в виде с плавающей точкой; параметров либо не принимает, либо принимает число (используется как количество разрядов в дробной части, если этот параметр не передан то он считается равным нулю); возвращает строковое представление инкапсуллируемого числа в виде с плавающей точкой (и с указанным количеством разрядов дробной части); возвращает строку (содержащую инкапсуллируемое число);

`toPrecesion()`

используется для возврата инкапсуллируемого числа с заданным количеством значащих разрядов; параметров либо не принимает, либо принимает число (используется как количество значащих разрядов, если этот параметр не передан то просто происходит вызов метода `toString()`); возвращает строковое представление инкапсуллируемого числа с заданным количеством значащих разрядов (причем, если заданное количество разрядов позволяет вместить все число — то оно представляется в виде с фиксированной точкой, если же не позволяет вместить все число — то оно представляется в виде с плавающей точкой); возвращает строку (содержащую инкапсуллируемое число);

Переменные объектов Number

`MIN_VALUE`

константа; содержит минимальное представимое в интерпретаторе число;

`MAX_VALUE`

константа; содержит максимальное представимое в интерпретаторе число;

`NEGATIVE_INFINITY`

константа; содержит специальное численное значение — минус бесконечность;

`POSITIVE_INFINITY`

константа; содержит специальное численное значение — плюс бесконечность;

`NaN`

константа; содержит специальное численное значение — не- число;

Члены объектов String

Конструктор объектов String

`String()`

конструктор объектов `String`; принимает параметр — значение (значение инициализации); создает объект `String`; возвращает объект `String`;

Методы объектов String

`toString()`

используется для возврата строкового представления своего объекта; параметров либо не принимает, либо принимает число (используется как основание системы счисления, в которой будет представлен результат); возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (значение элементарного типа);

`valueOf()`

используется для возврата значения своего объекта; аргументов не принимает; возвращает значение своего объекта (значение элементарного типа данных, которое инкапсуллирует); возвращает строку (значение элементарного типа);

`search()`

используется для поиска; принимает операнд — регулярное выражение; осуществляет поиск указанного регулярного выражения в своей строке; возвращает численное значение (позицию вхождения регулярного выражения в строку, если вхождений нет — то `-1`);
этот метод игнорирует флаг `g` регулярного выражения;

`match()`

используется для поиска; принимает операнд — регулярное выражение; осуществляет поиск указанного регулярного выражения в своей строке; возвращает массив (позиций вхождения — если флаг `g` выставлен; или же такой массив (нулевой элемент — строка (соответствующая регулярному выражению), последующие элементы — строки (соответствующие подвыражениям регулярного выражения)) - если флаг `g` сброшен);
этот метод учитывает флаг `g` регулярного выражения;

`replace()`

используется для поика с заменой; принимает операнды — регулярное выражение, и строку замены; осуществляет поиск указанного регулярного выражения в своей строке, осуществляет замену найденного строкой замены;
этот метод учитывает флаг `g` регулярного выражения

`localeCompare()`

используется для сравнения своей строки с указанной строкой (с учетом локализации); принимает параметр — указанная строка; сравнивает свою строку с указанной строкой (с учетом порядка локализации); возвращает число (если своя строка меньше указанной — отрицательное число, если своя строка равна указанной — число `0`, если своя строка больше указанной — положительное число);

`indexOf()`

используется для поиска указанной подстроки в своей строке; принимает параметры — строку (это указанная подстрока) и число (стартовая позиция поиска, этот аргумент можно опустить); возвращает число — позицию первого вхождения указанной подстроки в свою строку (если вхождения нет, то число `-1`); возвращает число;

`lastIndexOf()`

используется для поиска указанной подстроки в своей строке, ведя поиск от конца строки к началу; принимает параметры — строку (это указанная подстрока) и число (стартовая позиция поиска, этот аргумент можно опустить); возвращает число — позицию первого вхождения указанной подстроки в свою строку (если вхождения нет, то число `-1`), ведя поиск от конца строки к началу; возвращает число;

`charAt()`

используется для возврата символа своей строки, располагающегося по указанной позиции; принимает параметр — число (это указанная позиция); возвращает символ своей строки по указанной позиции; возвращает строку;

`charCodeAt()`

используется для возврата кода символа своей строки, располагающегося по указанной позиции; принимает параметр — число (это указанная позиция); возвращает код символа своей строки по указанной позиции; возвращает число;

`substr()`

используется для возврата подстроки своей строки; принимает параметры — число (стартовая позиция возврата подстроки, если задана отрицательным — то отсчет от конца строки), и число (количество символов подстроки, если не задано — то до конца строки); возвращает подстроку своей строки, руководствуясь переданными параметрами, свою строку не изменяет; возвращает строку (это подстрока);

`substring()`

используется для возврата подстроки своей строки (со стартовой позиции включительно по финальную позицию не включительно; причем, если стартовая позиция равна финальной — то исправно вернется пустая строка, если же стартовая позиция больше финальной — то эти позиции рокируются); принимает параметры — число (стартовая позиция), и число (финальная позиция); возвращает подстроку своей строки, руководствуясь переданными параметрами, свою строку не изменяет; возвращает строку (это подстрока);

`slice()`

используется для возврата подстроки из своей строки (со стартовой позиции включительно, по финальную позицию не включительно); принимает параметры — число (стартовая позиция, если задана отрицательным — то отсчитывается от конца строки), и число (финальная позиция, если задана отрицательным — то отсчитывается от конца строки, если не задана — то ей считается конец строки); возвращает подстроку своей строки, руководствуясь переданными параметрами, свою строку не изменяет; возвращает строку (это подстрока);

`split()`

используется для дробления своей строки на подстроки; принимает параметры — строку/ регулярное выражение (используется как разделитель; по позиции этого разделителя в строке и будет происходить ее дробление; разделитель может быть и не указан — тогда дробление даст в результате исходную строку, или указан пустым — тогда дробление будет через каждый символ строки), и число (лимит количества результирующих подстрок, этот лимит может быть и не задан); разбивает свою строку на подстроки по указанному разделителю (причем, сам разделитель исключается из результатов) и учитывая лимит дробления, этот метод создает массив и сохраняет в нем результат работы но свою строку не изменяет); возвращает объект `Array` (массив результирующих подстрок);

`fromCharCode()`

используется для создания строки по заданным кодам символов; принимает параметры — ряд значений (это значения кодов символов); создает строку по заданным кодам символов; возвращает строку (это созданная строка);
этот метод «статический»;

`concat()`

используется для конкатенации своей строки с рядом заданных значений; принимает параметры — ряд значений; осуществляет конкатенацию своей строки с рядом заданных значений; возвращает строку;

`toLowerCase()`

используется для возврата копии своей строки в нижнем регистре символов; параметров не принимает; возвращает копию своей строки в нижнем регистре символов; возвращает строку (копию своей строки в нижнем регистре символов);

`toLocaleLowerCase()`

используется для возврата копии своей строки в нижнем регистре символов (с учетом локализации); параметров не принимает; возвращает копию своей строки в нижнем регистре символов (с учетом локализации); возвращает строку (копию своей строки в нижнем регистре символов);

`toUpperCase()`

используется для возврата копии своей строки в верхнем регистре символов; параметров не принимает; возвращает копию своей строки в верхнем регистре символов; возвращает строку (копию своей строки в нижнем регистре символов);

`toLocaleUpperCase`

используется для возврата копии своей строки в верхнем регистре символов (с учетом локализации); параметров не принимает; возвращает копию своей строки в верхнем регистре символов (с учетом локализации); возвращает строку (копию своей строки в нижнем регистре символов);

Переменные объектов String

`length`

константа; содержит длину строки в количестве символов;

Объекты Array

Все массивы являются объектами `Array`. Им доступен синтаксис массивов и объектов. В силу нестрогой типизации языка `JavaScript` массивы могут содержать разнотипные элементы. Массивы создаются следующей записью:

```
var VarName = [ … ];
/* … это список значений;сэтот список может быть пустым;
или содержать пустые элементы, тогда он примет вид:
Value0, , Value2, …
пропущенные элементы не существуют,
но занимают место в адресном пространстве индексов */
```

Как видно из записи, запись «инициализатора массива» в языке `JavaScript` имеет специфический вид. Альтернативная запись:

```
var VarName = new Array( … );
/* … это переданные параметры,
 о параметрах — далее в этой главе */
```

Доступ к элементам массива, как обычно происходит посредством индексации. Приэтом, индексы могут быть указаны как непосредственно, так и в кавычках (это означает, что элемент массива — это член объекта- массива с именем своего индекса). Приэтом, если индекс указан не натуральным числом (и не важно прописан ли он без кавычек, или в кавычках), то он интерпретируется как имя члена объекта- массива (не являющегося элементом массива), и происходит попытка доступа к нему.
Массиву, как и любому другому объекту, после создания могут быть приписаны новые члены. Члены, являющиеся элементами массива, также могут быть приписаны. Приэтом, они могут быть приписаны не только в порядке счета индексов — тогда массив будет разрежен. Массив может быть разрежен также удалением элементов массива.
Массивы могут содержать элементы любого типа, в том числе и другие массивы. Такие массивы создаются обычной (в языке `JavaScript`) для этого записью. И доступ к элементам такого массива осуществляется обычной для этого записью:
`VarName[ … ] … [ … ]`

Члены объектов Array

Конструктор объектов Array

`Array()`

конструктор объектов `Array`; принимает параметры — либо ничего, либо значение (это количество элементов массива), либо ряд значений (это значения элементов массива); создает объект `Array`; возвращает объект `Array`;

Методы объектов Array

`toString()`

используется для возврата строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту), метод действет так — вызывает метод `toString()` для каждого элемента массива и выполняет конкатенацию результирующих строк (для разделителя используя запятую); возвращает строку;

`toLocaleString()`

используется для получения строкового представления своего объекта, причем в локализованном (согласно пользовательским региональным настройкам) виде; параметров не принимает; метод действет так — вызывает метод

`tolocaleString()` для каждого элемента массива и выполняет конкатенацию результирующих строк (для разделителя используя символ, согласно локализации);

`indexOf()`

используется для возврата индекса указанного элемента своего массива; принимает параметры — либо значение (значение целевого элемента), либо значение (значение целевого элемента) и значение (стартовая позиция поиска); возвращает индекс целевого элемента (если он найден, иначе — число `-1`), причем для поиск целевого элемента происходит как при использовании оператора `===`; возвращает число;

`lastIndexOf()`

используется для возврата индекса указанного элемента своего массива (поиск ведется от конца массива); принимает параметры — либо значение (значение целевого элемента), либо значение (значение целевого элемента) и значение (стартовая позиция поиска); возвращает индекс целевого элемента своего массива (если он найден, иначе — число `-1`), причем для поиск целевого элемента происходит как при использовании оператора `===` (поиск ведется от конца массива); возвращает число;

`push()`

используется для заталкивания указанных значений в свой массив; принимает параметры — ряд значений (произвольной длины ряд указанных значений); заталкивает ряд указанных значений в массив; возвращает число (это результирующая длина массива (значение члена `length` объекта- массива));

`pop()`

используется для выталкивания последнего элемента из своего массива; параметров не принимает; выталкивает последний элемент из массива; возвращает значение (это значение вытолкнутого элемента, если же массив изначально был пуст — то значение `undefinied`);

`shift()`

используется для сдвига своего массива на один элемент; параметров не принимает; сдвигает свой массив на один элемент (к началу массива); возвращает значение (это значение элемента, утерянного в результате сдвига);

`unshift()`

используется для сдвига своего массива и заполнения указанными значениями; принимает параметры — ряд указанных значений; сдвигает свой массив на количество элементов, соответствующих количеству переданных параметров, (к концу массива); возвращает число (это результирующая длина массива (значение члена `length` объекта- массива));

`reverse()`

используется для инвертирования порядка следования элементов в своем массиве; параметров не принимает; инвертирует порядок следования элементов в своем массиве; возвращает объект (это ссылка на свой массив);

`sort()`

используется для упорядочивания элементов своего массива; параметров либо не принимает (тогда порядок упорядочивания — по умолчанию), либо принимает функцию (пользовательская функция сравнения, задающая пользовательский порядок упорядочивания); осуществляет упорядочивание элементов своего массива, руководствуясь переданными параметрами (если порядок упорядочивания по умолчанию — то элементы упорядочиваются в алфавитном порядке (для этого при необходимости приводятся к строке), если же порядок упорядочивания задается пользовательской функцией сравнения, то эта функция должна быть прописана соответствующим образом — она должна принимать два аргумента, сравнивать их по своей логике и по результатам сравнения возвращать число (отрицательное число будет значить что — 1й аргумент следует считать меньше второго, число ноль будет значить — 1й аргумент следует считать равным второму, положительное число будет значить — 1й рагумент следует считать больше второго), в результате метод `sort()` будет упорядочивать элементы массива по прежнему в алфавитном порядке — просто порядок старшинства элементов будет определяться пользовательской функцией сравнения, само тело пользовательской функции сравнения может быть тривиальным — оно должно лишь принимать два аргумента и возвращать число — а корректное применение этой функции по своему массиву остается на откуп интерпретатору); возвращает объект (это ссылка на свой массив);
пользовательская функция сравнения применяется интерпретатором следующим образом — функция принимает два параметра, исполняет свое тело и возвращает значение сравнения; два принимаемых ей параметра — это элементы массива, участвующие в упорядочивании (лучше сказать, сортировке); алгоритм сортировки, применяемый интерпретатором, остается на откуп интерпретатора (но обычно это быстрая сортировка);
необходимо помнить, что интерпретатор не вызывает эту пользовательскую функцию по несуществующим элементам;

`reduce()`

используется для возврата значения редуцирования (лучше сказать, свертки) массива; принимает параметры — либо функцию (это пользовательская функция свертки), либо функцию (это пользовательская функция свертки) и значение (это начальное значение, если этот аргумент передан — то метод `reduce()` будет считать его добавленным в начало массива); применяет пользовательскую функцию свертки по своему массиву (эта функция должна быть прописана соответствующим образом — она должна принимать два параметра, выполнять их свертку по своей логике и возвращать значение их свертки, само тело пользовательской функции свертки может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему массиву остается на откуп интерпретатору); возвращает значение (это единственное значение, полученное применением интерпретатором пользовательской функции свертки);
пользовательская функция свертки применяется интерпретатором так — функция принимает два параметра, исполняет свое тело, и возвращает значение свертки; два принимаемых ей параметра — это начальное значение свертки (если не задано, то им считается первый элемент массива) и первый элемент массива (если начальное значение свертки не задано, то это второй элемент массива); интерпретатор применяет пользовательскую функцию свертки итеративно и сдвигаясь за итерацию на один элемент массива; в результате этих итераций функция свертки вернет единственное значение (поэтому она и называется функцией свертки массива);

`reduceRight()`

используется для возврата значения редуцирования (лучше сказать, свертки) массива; синонимичен вышеописанному методу за тем исключением, что выполняет свертку не с начала массива, а с конца;

`filter()`

применяется для фильтрации элементов своего массива при помощи пользовательской функции; принимает параметры — либо функцию (пользовательская функция, определяющая отфильтровывать ли элемент из массива, поэтому ее принято называть предикативной (для простоты - предикат)), либо функцию (пользовательская функция, определяющая отфильтровывать ли элемент из массива, поэтому ее принято называть предикативной (для простоты - предикат)) и объект (в его контексте будет работать предикативная функция); осуществляет фильтрацию элементов своего массива, руководствуясь переданными параметрами (передаваемая в параметрах предикативная функция должна быть прописана соответствующим образом — принимать параметр, определять (по своей логике) фильтровать ли элемент, и возвращать логическое значение (`true` – фильтровать, `false` – нет), само тело

предикативной функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему массиву остается на откуп интерпретатору), свой массив не изменяет, отфильтрованные элементы возвращает в новом массиве; возвращает массив (это массив отфильтрованных элементов);
предикативная функция применяется интерпретатором так — функция принимает параметр, исполняет свое тело, и возвращает логическое значение; принимаемый параметр — это элемент массива; интерпретатор применяет предикативную функцию по своему массиву итеративно, сдвигаясь за итерацию на один элемент массива; в результате этих итераций предикативная функция выполнит фильтрацию элементов массива;
необходимо помнить, что интерпретатор не вызывает эту пользовательскую функцию по несуществующим элементам;

`every()`

используется для проверки элементов своего массива при помощи пользовательской функции; принимает параметры — функцию (пользовательская функция, используемая для проверки элементов своего массива, поэтому ее принято называть предикативной (для простоты - предикат)) и объект (в его контексте и будет исполняться пользовательская функция, этот аргумент может быть и не передан); последовательно вызывает предикативную функцию от каждого из элементов своего массива (передаваемая в параметрах предикативная функция должна быть прописана соответствующим образом — принимать параметр, осуществлять прописанные программистом операции, и возвращать логическое значение (свидетельствующее о результате), само тело пользовательской функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему массиву остается на откуп интерпретатору); возвращает логическое значение (`true` – если предикативная функция вернет `true` для каждого элемента, `false` – иначе);
необходимо помнить, что интерпретатор не вызывает эту пользовательскую функцию по несуществующим элементам;

`some()`

используется для проверки элементов своего массива при помощи пользовательской функции; синонимичен вышеописанному методу за тем исключением, что возвращает значение `true` – если предикативная функция вернет `true` для хотя бы одного элемента;

`forEach()`

используется для вызова указанной пользовательской функции от каждого из элементов своего массива; принимает параметры — функцию (это

пользовательская функция), и объект (в его контексте и будет исполняться пользовательская функция, этот аргумент может быть и не передан); последовательно вызывает пользовательскую функцию от каждого из элементов своего масива (передаваемая в параметрах пользовательская функция должна быть прописана соответствующим образом — принимать параметр, осуществлять прописанные программистом операции, само тело пользовательской функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему массиву остается на откуп интерпретатору); возвращает ничтоже;
необходимо помнить, что интерпретатор не вызывает эту пользовательскую функцию по несуществующим элементам;

`map()`

используется для отображения элементов своего массива на элементы нового массива при помощи пользовательской функции; принимает параметры — функцию (это пользовательская функция) и объект (в его контексте и будет исполняться пользовательская функция, этот аргумент может быть и не передан); последовательно вызывает пользовательскую функцию от каждого из элементов своего масива (передаваемая в параметрах пользовательская функция должна быть прописана соответствующим образом — принимать параметр, осуществлять прописанные программистом операции, и возвращать значение (результат отображения текущего элемента), само тело пользовательской функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему массиву остается на откуп интерпретатору), свой массив не изменяет, результат отображения возвращает в новом массиве; возвращает массив (это новый массив состоящий из результатов отображения);
необходимо помнить, что интерпретатор не вызывает эту пользовательскую функцию по несуществующим элементам;

`concat()`

используется для конкатенации своего массива с рядом указанных значений; принимает параметры — ряд указанных значений; осуществляет конкатенацию своего массива с рядом указанных значений, свой массив не изменяет, резельтат возвращает в новом массиве; возвращает массив (это новый массив);

`join()`

используется для конкатенации элементов своего массива; принимает параметр — строку (используется как разделитель элементов в результате конкатенации, этот аргумент может быть не передан); осуществляет конкатенацию элементов своего массива, руководствуясь переданными параметрами (если параметр опущен, то используется разделитель по умолчанию — запятая), свой массив не

изменяет, результат возвращает в созданной строке; возвращает строку (результат конкатенации);

`slice()`

используется для возврата подмассива своего массива (со стартовой позиции включительно, по финальную позицию не включительно); принимает параметры — число (стартовая позиция, если задана отрицательным — то отсчитывается от конца массива), и число (финальная позиция, если задана отрицательным — то отсчитывается от конца массива, если не задана — то ей считается конец массива); возвращает подмассив своего массива, руководствуясь переданными параметрами, свой массив не изменяет, результат возвращает в новом массиве; возвращает массив (это подмассив);

`splice()`

используется для изменения своего массива; принимает параметры — значение (стартовая позиция, с которой начинается изменение), значение (количество изменяемых элементов, этот аргумент может быть значением `0` — тогда изменение массива будет проистекать не затиранием его элементов рядом указанных значений а вставкой (вклиниванием) в него ряда указанных значений), и ряд значений (этими значениями и будет осуществляться изменение массива, эти аргументы могут быть не переданы); осуществляет изменение своего массива, руководствуясь переданными параметрами (причем, в переданных параметрах количество изменяемых элементов может быть не равно длине ряда значений — тогда массив будет изменяться в большую или меньшую сторону чтобы вместить ряд указанных значений (вместить без разрежения)), этот метод изменяет свой массив; возвращает массив (это массив из оригинальных значений элементов, что были изменены, если же никакие значения не были изменены — то вернет пустой массив);

Переменные объектов Array

`length`

переменная; содержит число элементов массива (число элементов массива определяется исходя из занятого адресного пространства индексов — это число, как обычно, больше последнего занятого индекса на один);

Объекты RegExp

Регулярные выражения — это объекты `RegExp`. Они служат для поиска в строках. Для этого они заключают в себе полезную нагрузку — шаблон, по

которому производится поиск. Шаблон может включать в себя обычные символы (которые составляют полезную нагрузку строки) и специальные символы — так называемые «метасимволы» (которые интерпретируются специфически), в том числе символы эскейп- последовательностей. Рег. выражения создаются так:

```
var VarName = new RegExp(« … »);
// « … » - строка с полезной нагрузкой
```

Или же следующей записью:

```
var VarName = / … /; // … - сама полезная нагрузка
```

В этой записи могут быть указаны так называемые флаги регулярных выражений. Флаг представляет собой литеру, прописываемую непосредственно за регулярным выражением (то есть, непосредственно за `/ … /`). Есть флаги:

`i` – означает что поиск должен быть регистронезависимым;

`g` – означает что поиск должен быть глобальным (требуются все найденные соответствия);

`m` – означает что поиск должен быть в многострочном режиме;

Флаги могут использоваться вместе, и в любом порядке.

Специальные символы

`[ … ]`	перечисление символов …
`[^ … ]`	исключение перечисления символов …
`[a-zA-Z0-9]`	указанные диапазоны символов
`{n}`	предыдущий символ повторяется `n` раз
`{n,}`	предыдущий символ повторяется `n` раз и более
`{n,m}`	предыдущий символ повторяется `n … m` раз
`?`	предыдущий символ повторяется до `1` раза
`+`	предыдущий символ повторяется от `1` раза
`*`	предыдущий символ повторяется от `0` раз
`.`	любой обычный символ
`a|b|c`	`a` или `b` или `c`
`( … )`	регулярное выражение (целиком или его часть)
`~ …`	… с начала строки
`… $`	… в конце строки

Специальные символы включаются в полезную нагрузку регулярного выражения именно в таком виде, в каком они и указаны выше.

Поиск регулярных выражений

Поиск регулярного выражения осуществляется при помощи соответствующих методов объектов `RegExp` и `String`.

Члены регулярных выражений

Конструктор регулярных выражений

`RegExp()`

конструктор регулярных выражений; принимает аргументы — строку (полезная нагрузка регулярного выражения), или две строки (полезная нагрузка и флаги регулярного выражения); создает объект `RegExp`; возвращает объект `RegExp`;

Методы регулярных выражений

`toString()`

используется для получения строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержащую `/ … /` – литерал регулярного выражения);

`exec()`

используется для проверки на соответствие регулярному выражению; принимает аргумент — строку; проверяет указанную строку на соответствие своему регулярному выражению; возвращает `null` (если нет соответствия), или массив (если есть соответствия; нулевой элемент — строка (соответствующая рег. выражению), последующие элементы — строки (соответствующие подвыражениям регулярного выражения)); игнорирует флаг `g`;

`test()`

используется для проверки на соответствие регулярному выражению; принимает аргумент — строку; проверяет указанную строку на соответствие своему регулярному выражению; возвращает логическое значение (`true` – если соответствует; `false` – если нет); игнорирует флаг `g` ;

Переменные регулярных выражений

`source`

константа; содержит полезную нагрузку регулярного выражения;

`global`

константа; содержит флаг `g` регулярного выражения;

`ignoreCase`

константа; содержит флаг `i` регулярного выражения;

`multiline`

константа; содержит флаг `m` регулярного выражения;

`lastIdex`

переменная; используется при наличии флага `g` регулярного выражения, и содержит последнюю позицию поиска; эта переменная используется в нуждах языка;

Объекты Date

Это объекты, служащие для работы с датами.

Члены объектов Date

Конструктор объектов Date

`Date()`

конструктор объектов `Date`; параметров либо не принимает (и тогда создает объект `Date`, содержащий текущую дату и время), либо принимает число (и тогда интерпретирует его как указанное значение даты и времени в миллисекундах), либо принимает строку (и тогда интерпретирует ее как указанное значение даты и времени), либо принимает ряд значений (и тогда интерпретирует их как составляющие указанного значения даты и времени, из этог ряда значений год и месяц обязательны); создает объект `Date`; возвращает объект `Date`;

Методы объектов Date

`toString()`

используется для возврата строкового представления своего объекта в рамках текущего часового пояса; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту) в рамках текущего часового пояса; возвращает строку (содержащую `[ … ]`, где … всегда различается в разных реализациях языка);

`toLocaleString()`

используется для возврата строкового представления своего объекта с учетом локализации и в рамках текущего часового пояса; параметров не принимает; возвращает строковое представление своего объекта с учетом локализации (лучше сказать, строку сопоставленную тому или иному языковому объекту) и в рамках текущего часового пояса; возвращает строку (содержащую `[ … ]`, где … всегда различается в разных реализациях языка);

`toTimeString()`

используется для возврата строкового представления времени из своего объекта; параметров не принимает; возвращает строковое представление времени из своего объекта, зависящее от реализации языка; возвращает строку;

`toLocaleTimeString()`

используется для возврата строкового представления времени из своего объекта с учетом локализации; параметров не принимает; возвращает строковое представление времени из своего объекта, зависящее от реализации языка с учетом локализации; возвращает строку;

`toDateString()`

используется для возврата строкового представления даты из своего объекта; параметров не принимает; возвращает строковое представление даты из своего объекта, зависящее от реализации языка; возвращает строку;

`toLocaleDateString()`

используется для возврата строкового представления даты из своего объекта с учетом локализации; параметров не принимает; возвращает строковое

представление даты из своего объекта, зависящее от реализации языка с учетом локализации; возвращает строку;

`toGMTString()`

используется для получения строкового представления своего объекта (его полезная нагрузка представляется в формате GMT, это представление считается устаревшим); возвращает строковое представление своего объекта (полезная нагрузка будет в указанном формате); возвращает строку;

`toUTCString()`

используется для получения строкового представления своего объекта (его полезная нагрузка представляется в формате UTC, этот формат зависит от реализации языка); возвращает строковое представление своего объекта (полезная нагрузка будет в указанном формате); возвращает строку;

`toISOString()`

используется для получения строкового представления своего объекта (его полезная нагрузка представляется в формате ISO, то есть в следующем формате: `yyyy-mm-ddThh:mm:ss:sssZ`, где `Z` – это часовой пояс); возвращает строковое представление своего объекта (полезная нагрузка будет в указанном формате); возвращает строку;

`toJSON()`

используется для получения представления своего объекта в формате `JSON`; принимает параметр — значение; вызывает метод `JSON.stringify` и передает ему свой аргумент; возвращает строку (к-рую и возвращает вызванный метод);

`valueOf()`

используется для возврата значения своего объекта; параметров не принимает; возвращает значение своего объекта; возвращает значение (значение своего объекта в миллисекундах);

`getFullYear()`

используется для возврата поля года из своего объекта; параметров не принимает; возвращает поле года (в виде `yyyy`); возвращает число;

`setFullYear()`

используется для установки поля года, месяца и дня своего объекта; принимает параметры — число(год, в виде `yyyy`), число (месяц, [`0 … 11`], этот аргумент может быть опущен), и число (день [`1 … 31`], этот аргумент может быть опущен); устанавливает поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getYear()`

используется для возврата поля года из своего объекта; параметров не принимает; возвращает поле года своего объекта (значение этого поля минус `1900`); возвращает число; этот метод устарел;

`setYear()`

используется для установки поля года в своем объекте; принимает параметр — число (значение года [`0 … 99`]); устанавливает значение поля года в своем объекте; возвращает обновленное значение в мс; этот метод устарел;

`getMonth()`

используется для возврата поля месяца из своего объекта; параметров не принимает; возвращает значение поля месяца; возвращает число ([`0 … 11`]);

`setMonth()`

используется для установки поля месяца и числа своего объекта; принимает параметры — число (значение месяца, `[0 … 11]`), и число (поле дня, [1 … 31, этот аргумент может быть опущен]); устанавливает значение полей своего объекта руководствуясь переданными параметрами; возвращает обносвленное значение в миллисекундах;

`getDate()`

используется для возврата поля дня из своего объекта; параметров не принимает; возвращает текущий день месяца своего объекта; возвращает число;

`setDate()`

используется для установки поля дня своего объекта; принимает параметр - число; устанавливает поле дня своего объекта; возвращает обновленное значение в милисекундах;

`getDay()`

используется для возврата поля дня недели своего объекта; параметров не принимает; возвращает поле дня недели; возвращает число (`[0 … 6]`);

`getHours()`

используется для возврата поля часа своего объекта; параметров не принимает; возвращает поле часа своего объекта; возвращает число (`[0 … 23]`);

`setHours()`

используется для установки поля часа, минуты, секунды и миллисекунды своего объекта; принимает параметры — число (часы, `[0 … 23]`), число (минуты, `[0 … 59]`, этот параметр может быть опущен), число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает обозначенные поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getMinutes()`

используется для возврата поля минуты своего объекта; параметров не принимает; возвращает поле минуты своего объекта; возвращаетчисло (минуты, `[0 … 59]`);

`setMinutes()`

используется для установки поля минуты, секунды, и миллисекунды своего объекта; принимает параметры - число (минуты, `[0 … 59]`), число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает поля минуты, секунды, и миллисекунды своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getSeconds()`

используется для возврата поля секунды своего объекта; параметров не принимает; возвращает поле секунды своего объекта; возвращает число (секунды, `[0 … 59]`);

`setSeconds()`

используется для установки полей секунды и миллисекунды своего объекта; принимает параметры - число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает поля секунды, и миллисекунды своего объекта руководствуясь переданными параметрами; возвращает обн. значение в мс;

`getMilliseconds()`

используется для возврата поля миллисекунды своего объекта; параметров не принимает; возвращает поле миллисекунды; возвращает число (`[0 … 999]`);

`setMilliseconds()`

используется для установки поля миллисекунд своего объекта; принимает параметр — число (миллисекунды, `[0 … 999]`); устанавливает поле миллисекунд своего объекта; возвращает число (`[0 … 999]`);

`getTime()`

используется для возврата представления полезной нагрузки своего объекта в миллисекундах; параметров не принимает; возвращает представление полезной нагрузки своего объекта в миллисекундах; возвращает число;

`setTime()`

используется для установки полезной нагрузки своего объекта; принимает параметр — число (значение инициализации полезной нагрузки своего объекта в миллисекундах); устанавливает полезную нагрузку своего объекта; возвращает число (это переданный аргумент);

`getTimezoneOffset()`

используется для возврата разницы между местным временем и временем по Гринвичу; параметров не принимает; возвращает эту разницу (в минутах); возвращает число (в минутах);

`getUTCFullYear()`

используется для возврата поля года своего объекта; параметров не принимает; возвращает поле года своего объекта; возвращает число (`yyyy`);

`setUTCFullYear()`

используется для установки поля года, месяца и дня своего объекта; принимает параметры — число(год, в виде `yyyy`), число (месяц, [`0 … 11`], этот аргумент может быть опущен), и число (день [`1 … 31`], этот аргумент может быть опущен); устанавливает поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getUTCMonth()`

используется для возврата поля месяца из своего объекта; параметров не принимает; возвращает значение поля месяца; возвращает число ([`0 … 11`]);

`setUTCMonth()`

используется для установки поля месяца и числа своего объекта; принимает параметры — число (значение месяца, `[0 … 11]`), и число (поле дня, [1 … 31, этот аргумент может быть опущен]); устанавливает значение полей своего объекта руководствуясь переданными параметрами; возвращает обносвленное значение в миллисекундах;

`getUTCDate()`

используется для возврата поля дня из своего объекта; параметров не принимает; возвращает текущий день месяца; возвращает число;

`setUTCDate()`

используется для установки поля дня своего объекта; принимает параметр - число; устанавливает поле дня своего объекта; возвращает обновленное значение в милисекундах;

`getUTCDay()`

используется для возврата поля дня недели своего объекта; параметров не принимает; возвращает поле дня недели; возвращает число (`[0 … 6]`);

`getUTCHours()`

используется для возврата поля часа своего объекта; параметров не принимает; возвращает поле часа своего объекта; возвращает число (`[0 … 23]`);

`setUTCHours()`

используется для установки поля часа, минуты, секунды и миллисекунды своего объекта; принимает параметры — число (часы, `[0 … 23]`), число (минуты, `[0 … 59]`, этот параметр может быть опущен), число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает обозначенные поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getUTCMinutes()`

используется для возврата поля минуты своего объекта; параметров не принимает; возвращает поле минуты; возвращает число (минуты, `[0 … 59]`);

`setUTCMinutes()`

используется для установки поля минуты, секунды, и миллисекунды своего объекта; принимает параметры - число (минуты, `[0 … 59]`), число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает обозначенные поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getUTCSeconds()`

используется для возврата поля секунды своего объекта; параметров не принимает; возвращает поле секунды; возвращает число (`[0 … 59]`);

`setUTCSeconds()`

используется для установки полей секунды и миллисекунды своего объекта; принимает параметры - число (секунды, `[0 … 59]`, этот параметр может быть опущен), и число (миллисекунды, `[0 … 999]`, этот параметр может быть опущен); устанавливает обозначенные поля своего объекта руководствуясь переданными параметрами; возвращает обновленное значение в миллисекундах;

`getUTCMilliseconds()`

используется для возврата поля миллисекунды своего объекта; параметров не принимает; возвращает поле миллисекунды своего объекта; возвращает число (миллисекунды, `[0 … 999]`);

`setUTCMilliseconds()`

используется для установки поля миллисекунд своего объекта; принимает параметр — число (миллисекунды, `[0 … 999]`); устанавливает поле миллисекунд своего объекта; возвращает число (`[0 … 999]`);

`UTC()`

используется для преобразования ряда указанных значений (соответствующих полям объекта `Date`) к значению его полезной нагрузки в виде миллисекунд; принимает параметры — ряд чисел, соответствующих году, месяцу, числу, часу, минуте, секунде и миллисекунде; преобразует ряд указанных значений к значению полезной нагрузки объекта `Date` в виде миллисекунд; возвращает число (миллисекунды);
этот метод «статический»;

`now()`

используется для возврата текущего времени (в составе полной полезной нагрузки объекта `Date`) в виде миллисекунд; параметров не принимает; возвращает текущее время (в составе полной полезной нагрузки объекта `Date`) в виде миллисекунд; возвращает число (миллисекунды);
этот метод «статический»

`parse()`

используется для получения полезной нагрузки объекта `Date` в виде миллисекунд из указанной строки; принимает параметр — строку (указанная строка, предположительно содержащая полезную нагрузку объекта `Date`); интерпретирует указанную строку как содержащую полезную нагрузку объекта `Date`, и преобразует еек миллисекундному виду; возвращает число (миллисекунды);
этот метод «статический»;

Объекты Function

В языке `JavaScript` исполняемый код сценария может храниться как в объектах (в них он хранится в функциях — методах объектов), так и вне объектов (вне объектов он хранится в обычных функциях). В любом случае для каждой функции языком поставляется служебный объект — объект функции (объект `Function`). Это делается в нуждах языка. Благодаря чему функции имеют объектное представление, и им доступен объектный синтаксис.

В выражениях функции могут быть использованы точно так же, как и любые ссылочные типы. Но, необходимо помнить, что к переменной из тела функции нельзя обратиться просто так: `FuncName.VarName`. Переменные из тела функции хранятся в ней специфическим способом (этот способ всегда остается на откуп реализации языка). Так что обращаться к ним можно только как это положено — в самом теле функции. К параметрам функции тоже нельзя обратиться так: `FuncName.VarName`. Параметры хранятся в массиве аргументов. Так что обращаться к ним можно только как это положено — в самом теле функции (непосредственно, или индексируя массив аргументов). Чтобы к члену функции можно было обратиться как к члену функции, то это должен быть член функции — а не просто переменные, которыми оперирует ее тело. Действительно, тело функции — это исполняемый код, и хранится в памяти как исполняемый код. То есть, искать в нем такие привычные понятия, как члены объектов, просто иррационально. Такого не приходится ожидать даже от такого языка, как `JavaScript`.

В языке `JavaScript` объявления функций должны быть полными (кратких не предусмотрено). Функции объявляются так:

```
function FuncName (VarName0, … )
{ … }
/* return опционален;
его формы записи:
return ( … );
return … ;
return; */
```

Функцию можно разместить в переменной:

```
VarName = FuncName;
```

Это возможно и при объявлении:

```
VarName = function FuncName ( … ) { … };
```

Тогда эта переменная будет служить ссылкой на соответствующий объект функции (и вызов функции будет возможен в следующей записи `VarName( … );`). Для вызова функции будет использоваться имя переменной, а имя функции будет теперь доступно только в области видимости самой функции (конечно, если функция будет размещена в переменной при объявлении, иначе она по прежнему будет доступна и под своим именем).

Таким образом, при размещении функции в переменной, ее изначальное имя оказывается избыточными сведениями. Поэтому, при размещении функции в переменной, имя функции можно и не указывать:

```
VarName = function ( … ) { … };
```

При использовании этой записи переменная может быть затем реинициализирована. Но, при использовании другой формы записи (где имя функции указано), переменная более не может быть реинициализирована (конечно, это касается только тех записей, где функция сохраняется в

переменной при объявлении, иначе переменная по прежнему может быть реинициализирована).
Объекты функций содаются интерпретатором еще на этапе загрузки сценария, и поэтому функции доступны в исходном коде еще до их объявления. Однако, если функция сохранена в переменной, то она будет доступна только после исполнения этого выражения.
Функции могут быть вложенными. Но, функции могут быть вложены лишь в функции (и объекты) — ни в какие другие структуры кода они не могут быть вложены. Исключение — тот случай, когда при объявлении функция сохраняется в переменной — в этом случае объявление функции может быть вложенным в прочие структуры кода.

Примечание

Объявление функции в разных формах записи принято называть по разному. Это должно быть оговорено.Форма записи:
```
function FuncName( … ) { … }
```

называется, собственно, объявлением (или, лучше сказать определением) функции. Форма записи:
```
var VarName = function FuncName( … ) { … };
```

называется объявлением именованной функции. Форма записи:
```
var VarName = function( … ) { … };
```

называется просто функциональным выражением.

Примечание к примечанию

Функция, объявленная просто функциональным выражением, может быть вызвана непосредственно в этом выражении. Для этого прибегают к специфической записи:
```
var VarName = (function( … ) { … }( … ));
```
Как видно из этой записи, для вызова функции функциональное выражение завершается записью `( … )`, где функции передаются ее фактические аргументы. Кроме того, функциональное выражение при этом заключается в скобки, и принимает следующий вид: `(function( … ) { … }( … ))`.
Необходимо помнить, что только функция, объявленная просто функциональным выражением, может быть вызвана так.
В языке `JavaScript` имеется также следующая специфическая возможность вызова функций. Когда вызов функции возвращает объект, то из него можно вызвать метод (причем, в том же самом выражении, что и вызов функции). Это делается следующей записью:
```
FuncName( … ).MethodName( … );
```

Если и вызваный метод возвращает объект, то из него тоже может быть вызван его метод, и тогда запись принимает вид:
`FuncName( … ). … .MethodName( … );`

Таким образом, вызовы методов могут происходить по цепочке (причем, длина этой цепочки может быть любой).

Рекурсия

Идентично тому, как в `C++`. И форма записи тоже.

Ключевое слово this

Хотя в языке `JavaScript` нет указателей, все функции могут ссылаться на инкапсулирующий их объект посредством ключевого слова `this`.
Необходимо помнить, что в случае вложенных функций инкапсулирующая функция никогда не является доступной по ключевому слову `this`.
Также необходимо помнить, что если функция вызывается как просто функция, то инкапсулирующим объектом считается глобальный объект.

О перегрузке функций

В языке `JavaScript` нет перегрузки функций. Строго говоря, в языке `JavaScript` нет переопределения функций именно перегрузкой. Но, переопределение функций собственно переопределением функций есть. Собственно переопределение функций производится так — к уже имеющейся (у объекта) функции обращаются по записи (приписывая ей новый контент); таким образом, переопределение функций производится обращением к ним (по записи) как к членам их объектов.

О параметрах функций. Передача параметров

Элементарные типы данных передаются по значению, а ссылочные типы передаются по ссылке.

О параметрах функций. Количество и тип

При вызове функции ей можно передать параметры, любые по типу, и даже по количеству (это не приведет к ошибке).
Если функции передано меньше параметров, чем у нее есть — то отсутствующие параметры будут инициализированы специальным значением `undefinied`. Если функции передано больше параметров, чем у нее есть — то избыточные параметры все равно будут приняты. Все параметры функции хранятся в объекте

`Arguments` (член объекта вызова); для ссылки на него служит член функции `FuncName.arguments`; избыточные параметры не доступны по имени, но они доступны посредством индексации (`arguments[ … ]` , все параметры индексируются с нуля); впрочем, все параметры функции доступны по индексации. Если функция должна вести себя по разному (взависимости от типов и количества переданных параметров), то это достигается включением в тело функции соответствующих условных операторов и проверок.

Члены объектов- функций

Конструктор объектов- функций

`Function()`

конструктор объектов- функций; принимает параметры — секвенцию строк (причем, последняя из них трактуется как содержащая тело функции — даже если это и есть единственная переданная конструктору строка, остальные же трактуются как содержащие объявления параметров); создает объект `Function`; возвращает объект `Function`;
этот конструктор не принимает в параметрах имя функции — таким образом, применение этого конструктора объявляет функцию просто функциональным выражением;
этот конструктор приводит к объявлению функции в глобальной области видимости (и не важно, где он применен — в глобальной области видимости, или же в области видимости какой- либо функции);

Методы объектов- функций

`toString()`

используется для возврата строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержащую `[ … ]`, где … всегда различается в разных реализациях языка);

`call()`

ипользуется для вызова своего объекта- функции как метода указанного объекта (то есть, в контексте указанного объекта); принимает параметры — ряд различных параметров (первый из них трактуется как указатель на объект, остальные — как параметры, передаваемые вызываемой функции); вызывает

функцию в контексте указанного объекта; возвращает то, что вернет вызванная функция (свой объект- функция);

`apply()`

синоним вышеописанного; разве что аргументы принимает в виде массива;

`bind()`

используется для «связывания» своего объекта- функции с указанным объектом; принимает параметры — указанный объект, и ряд значений (передаются опционально, и используются как параметры, которые своя функция получит, если будет вызвана вдальнейшем (причем, если при вызове ей передать еще параметры, то они просто дополнят их)); «связывает» свой объект- функцию с указанным объектом (если вдальнейшем эта «связанная» функция будет вызвана, то она будет вызвана как метод указанного объекта (то есть, в контексте указанного объекта)); возвращает объект- функцию (это свой объект-функция, претерпевший связывание);

Переменные объектов- функций

`prototype`

используется для ссылки на прототип; содержит член — `prototype.constructor` (используется для ссылки на конструктор);

`arguments`

эта переменная была рассмотрена выше; также можно дополнить, что этот член инкапсулирует в себе переменную `arguments.lenght` – эта переменная используется языком для того, чтобы объект `FuncName.arguments` можно было индексировать как массив;

`length`

в данном случае речь идет о члене самого объекта- функции (а конкретно `FuncName.lenght`); эта переменная используется языком для хранения количества объявленных параметров;

calee

используется для обеспечения совместимости с устаревшим кодом — в современном коде его использовать не рекомендуется; указывает на функцию, исполняемую в данный момент;

Объекты Error

Как обычно, выброс прерываний происходит в ходе исполнения сценария, либо производится соответствующей записью в коде сценария.
Выброшенные прерывания должны быть обработаны пользовательским обработчиком, иначе же они будут обработаны системным обработчиком (а он просто выводит сообщение об ошибке и завершает исполнение сценария). В языке JavaScript прерывания сопоставляются с с соответствующими им объектами. Это объекты Error и прочие объекты, ходные с ними.
Если выброс прерывания делается в коде сценария, то эта запись имеет следующий вид:

```
throw new Error( … );
```

Перехват выброшенных прерываний производится следующей структурой кода:

```
try { … }
/* блок try */
catch( … ) { … }
/* блок catch;
в ( … ) объявляется переменная
(доступная только этому блоку),
эта переменная будет использоваться
для ссылки на перехваченное прерывание -
такое поведение будет организовано интерпретатором */
finally { … }
/* блок finally выполняется безопционально,
независимо от выбросов прерываний! */
```

Как обычно, блоки catch и finally опциональны (но должен присутствовать хотя бы один из них). Блоки try, как обычно, могут быть вложенными. И поиск обработчика, как обычно, охватывает эту вложенную структуру. Причем, если блока catch в непосредственной близости от try нет (и поиск идет во вложенной структуре перехвата прерываний), то сперва будет исполнен блок finally (если он конечно же есть) — и лишь потом начнется поиск блока catch во вложенной структуре перехвата прерываний. Если же блок catch в непосредственной близости от try есть, то сперва, как обычно, будет исполнен он — и лишь потом блок finally.
В языке JavaScript в непосредственной близости от блока try может быть не более одного блока catch. Это потому, что принятая в языке JavaScript не строгая типизация делает запись ряда обработчиков избыточной.

Действительно, при не строгой типизации, различные прерывания могут быть перехвачены единственным обработчиком, и обработаны по его логике.

Члены объектов Error

Конструктор объектов Error

`Error()`

конструктор объектов `Error`; параметров либо не принимает, либо принимает, либо принимает строку (пользовательское сообщение об ошибке, если этот параметр не передан, то будет использоваться сообщение по умолчанию); создает объект `Error`; возвращает объект `Error`;

Методы объектов Error

`toString()`

используется для возврата строкового представления своего объекта; параметров не принимает; возвращает строковое представление своего объекта (лучше сказать, строку сопоставленную тому или иному языковому объекту); возвращает строку (содержимое этой строки всегда различается в разных реализациях языка);

Переменные объектов Error

`message`

переменная, содержит строку (сообщение об ошибке, пользовательское сообщение, или сообщение по умолчанию);

`name`

константа; содержит строку (имя типа прерывания, в него выступает имя конструктора своего объекта);

Список прерываний

`SyntaxError`

синтаксическая ошибка;

`TypeError`

ошибка при приведении типов (в некоторых реализациях возникает так же при передачефункции большего количества параметров, чем она ожидает получить);

`ReferenceError`

ошибка обращения по чтению к несуществующей переменной;

`RangeError`

ошибка выхода значения за его пределы (напр. выход индекса массива за `0`);

`EvalError`

ошибка выхода за ограничения, налагаемые на функцию `eval()`;

`URIError`

ошибка выбрасываемая функциями `decodeURI()`, `decodeURIComponent()` - если декодируемая ими строка содержит недопустимые шестнадцатеричные эскейп- последовательности; и методами `encodeURI()`, `encodeURIComponent()` - если кодируемая ими строка содержит недопустимые суррогатные пары символов;

Global

Так принято называть глобальный объект в `Pure JavaScript`. Это служебный объект, создаваемый самим интерпретатором, и конструктора он не имеет.

Члены Global

Это поставляемые самим языком члены, и пользовательские члены (необходимо помнить, что все в сценарии инкапсуллируется глобальным объектом). Они доступны непосредственно по имени, всвязи с чем о них и не принято говорить как о членах какого- либо объекта.

Глобальные функции

`eval()`

используется для исполнения указанного кода сценария; принимает параметр — строку (значение э.т.д., если же это обертка — то она будет просто возвращена этой функцией); исполняет код, прописанный в принятой строке (это может быть один оператор или более); возвращает значение вычисл. последним оператором; эта функция может быть вызвана прямо или косвенно (присвоена переменной и вызвана по ее имени);
будучи вызвана в некой области видимости, эта функция в ней и работает; однако, косвенный вызов (из какой бы то ни было области видимости) всегда считается вызовом в глобальной области видимости;

`isNaN()`

используется для проверки своего параметра на не-число; принимает параметр — значение (при необходимости приводится к численному); проверяет, является ли параметр не- числом; возвращает логическое значение (`true` – если является, `false` – если нет);

`isFinite()`

используется для проверки своего параметра на конечность; принимает параметр — значение (при необходимости приводится к численному); проверяет, является ли параметр конечным числом (не плюс- минус бесконечностью); возвращает логическое значение (`true` – если является, `false` – если нет);

`parseInt()`

используется для получения числа (его целой части) из его строкового представления; принимает параметры — строку (указанная строка) и число (основание системы счисления, в которой представлен аргумент (число в диапазоне [`2 … 36`], 0 – в поддерживаемой для литералов (шест. или дес.); этот параметр может быть опущен); считывает число из указанной строки (ситывание прекращается, когда встречается символ, не являющийся цифрой (причем, в указанной системе счисления, если этот параметр передан)); возвращает число (или же не- число, если не удалось считать число);

`parseFloat()`

используется для получения числа из его строкового представления; принимает параметр — строку (указанная строка); считывает число из указанной строки

(ситывание прекращается, когда встречается символ, не являющийся частью числа); возвращает число (или же не- число, если не удалось считать число);

`escape()`

используется для кодирования указанной строки; принимает параметр — указанную строку; кодирует указанную строку (заменяет специальные символы эскейп- последовательностями, специальными в данном случае считаются все символы кроме литер, нумеров, и следующих:`@*_/.+-`); возвращает строку (результат); эта функция считается устаревшей;

`unescape()`

используется для декодирования указанной строки; принимает параметр — указанную строку; декодирует указанную строку (эта функция — обратная вышеописанной функции); возвращает строку (результат); считается устар.;

`encodeURI()`

используется для кодирования указанной строки; принимает параметр — указанную строку; кодирует указанную строку (заменяет специальные символы эскейп- последовательностями, специальными в данном случае считаются все символы кроме литер, нумеров, и следующих:`-_.!~*'()` и `;/?:@&=+$,#`); возвращает строку (результат);
при использовании этой функции следует помнить, что символы `?` и `#` используются как разделители `URI`; если эти символы есть — то в составе `URI` есть разные компоненты, и их следует кодировать поотдельности — при помощи нижеописанной функции;

`encodeURIComponent()`

используется для кодирования указанной строки; принимает параметр — указанную строку; кодирует указанную строку (заменяет специальные символы эскейп- последовательностями, специальными в данном случае считаются все символы кроме литер, нумеров, и следующих:`-_.!~*'()`); возвращает строку (результат);

`decodeURI()`

используется для декодирования указанной строки; принимает параметр — указанную строку; декодирует указанную строку (эта функция — обратная функции `encodeURI()`); возвращает строку (результат);

`decodeURIComponent()`

используется для декодирования указанной строки; принимает параметр — указанную строку; декодирует указанную строку (эта функция — обратная функции `encodeURIComponent()`); возвращает строку (результат);

Глобальные переменные

`Infinity`

константа; содержит численное значение плюс бесконечность;

`NaN`

константа; содержит численное значение не- число;

`undefinied`

константа; содержит специальное значение `undefinied`;

`Object`

объект- функция; конструктор объектов `Object`;

`Boolean`

объект- функция; конструктор объектов `Boolean`;

`Number`

объект- функция; конструктор объектов `Number`;

`String`

объект- функция; конструктор объектов `String`;

`Array`

объект- функция; конструктор объектов `Array`;

`RegExp`

объект- функция; конструктор объектов `RegExp`;

`Date`

объект- функция; конструктор объектов `Date`;

`Function`

объект- функция; конструктор объектов `Function`;

`Error`

объект- функция; конструктор объектов `Error`;

`SyntaxError`

объект- функция; конструктор объектов `SyntaError`;

`TypeError`

объект- функция; конструктор объектов `TypeError`;

`ReferenceError`

объект- функция; конструктор объектов `ReferenceError`;

`RangeError`

объект- функция; конструктор объектов `RangeError`;

`EvalError`

объект- функция; конструктор объектов `EvalError`;

`URIError`

объект- функция; конструктор объектов `URIError`;

`JSON`

служебный объект (инкапсуллирующий функции `JSON.stringify()`, и `JSON.parse()`), конструктора не имеет;

`Math`

служебный объект (инкапсуллирующий математические функции и константы), конструктора не имеет;

Объект Math

Методы объекта Math

Это следующие математические функции.

`sin()`	sin(x)
`cos()`	cos(x)
`tan()`	tg(x)
`asin()`	arcsin(x)
`acos()`	arccos(x)
`atan()`	arctg(x)
`atan2()`	atan2(x)
`abs()`	module(x)
`sqrt()`	sqrt(x)
`pow()`	x pow(y)
`exp()`	e pow(x)
`log()`	ln x
`round()`	round(x), near
`floor()`	round(x), down
`ceil()`	round(x), up
`min()`	min(x, y), minimal value
`max()`	max(x, y), maximal value

Переменные объекта Math

Это следующие математические константы.

`E`	e
`LN10`	ln 10
`LN2`	ln 2
`LOG10E`	lg e
`LOG2E`	$\log_2 e$
`PI`	pi

SQRT1_2	2 pow(-1/2)
SQRT2	2 pow(1/2)

Объект JSON

Методы объекта JSON

stringify()

используется для преобразования к JSON; принимает параметры — значение (которое требуется преобразовать — им может быть объект, массив, или значение элементарных типов), значение (определяющее, что следует отфильтровать в первом аргументе для преобразования к JSON (это значение может задаваться как пользовательской функцией, определяющей что фильтровать, так и массивом предопределенных значений), этот аргумент может быть опущен), и значение (определяющее пробелы используемые при форматировании результата преобразования, этот аргумент может быть опущен); выполняет преобразование указанного значения к JSON руководствуясь аргументами (причем, если в указанном значении есть вложенные в него объекты — то из них предварительно будет вызываться метод toJSON(), и лишь затем будет происходить дальнейшее преобразование возвращенного им результата; если же метод toJSON() отсутствует, то он соответственно не вызывается, и дело обходится без него; если же второй аргумент этого метода задан — и задан функцией, то она вызывается наподобие метода toJSON() для вложенных объектов (вызывается самим языком, тело — на усмотрения пользователя, в интересах работоспособности эта функция разве-что должна принимать два аргумента (имя вложенного объекта и его значение соответственно) — выполнять тело, и возвращать результат)); возвращает строку (с результатом);

parse()

используется для преобразования из JSON; принимает параметры — строку (JSON), и функцию (эта пользовательская функция будет вызываться языком в целях создания членов создаваемого объекта из значений э.т.д. (полученных в ходе преобразования), этот параметр может быть опущен); преобразует указанное значение из JSON в представление на JavaScript, руководствуясь переданными параметрами (если же второй аргумент этого метода задан, то эта пользовательская функция будет вызываться от каждого значения э.т.д., полученного в ходе преобразования (вызываться самим языком, тело — на усмотрения пользователя, в интересах работоспособности эта функция разве-что должна принимать два аргумента (имя создаваемого члена и его значение

соответственно) — выполнять тело, и возвращать результат); возвращает строку (результат);

Переменные объекта JSON

Их нет.