# 1.JavaScript: общая характеристика. История JavaScript

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

Обычно используется как *встраиваемый* язык для программного доступа к объектам приложений.

Широкое применение находит в браузерах для придания интерактивности веб-страницам.

**История JavaScript**

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

**09.1995** – LiveScript (NN 2.0 beta).

**12.1995** – JavaScript (NN 2.0B3).

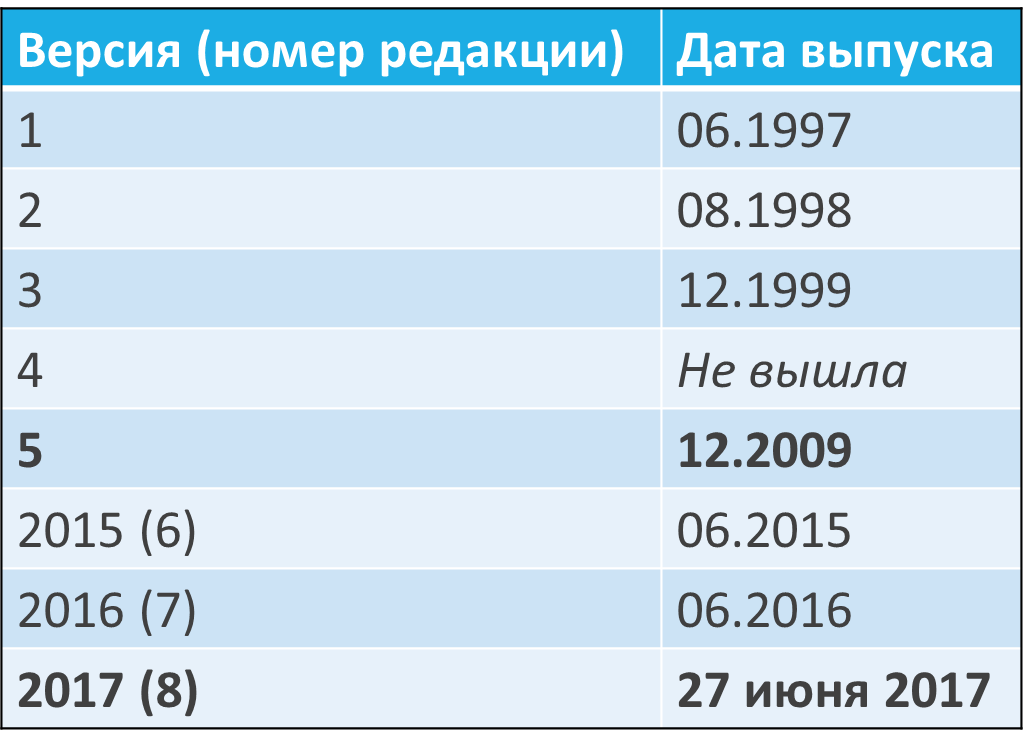
**07.1996** – Microsoft создаёт **JScript** (Internet Explorer 3.0).

В конце 1996 года Netscape обращается в ECMA International с просьбой утвердить стандарт JavaScript.

**06.1997** – спецификация ECMA-262 (первая редакция).

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

**Редакции ECMA-262**

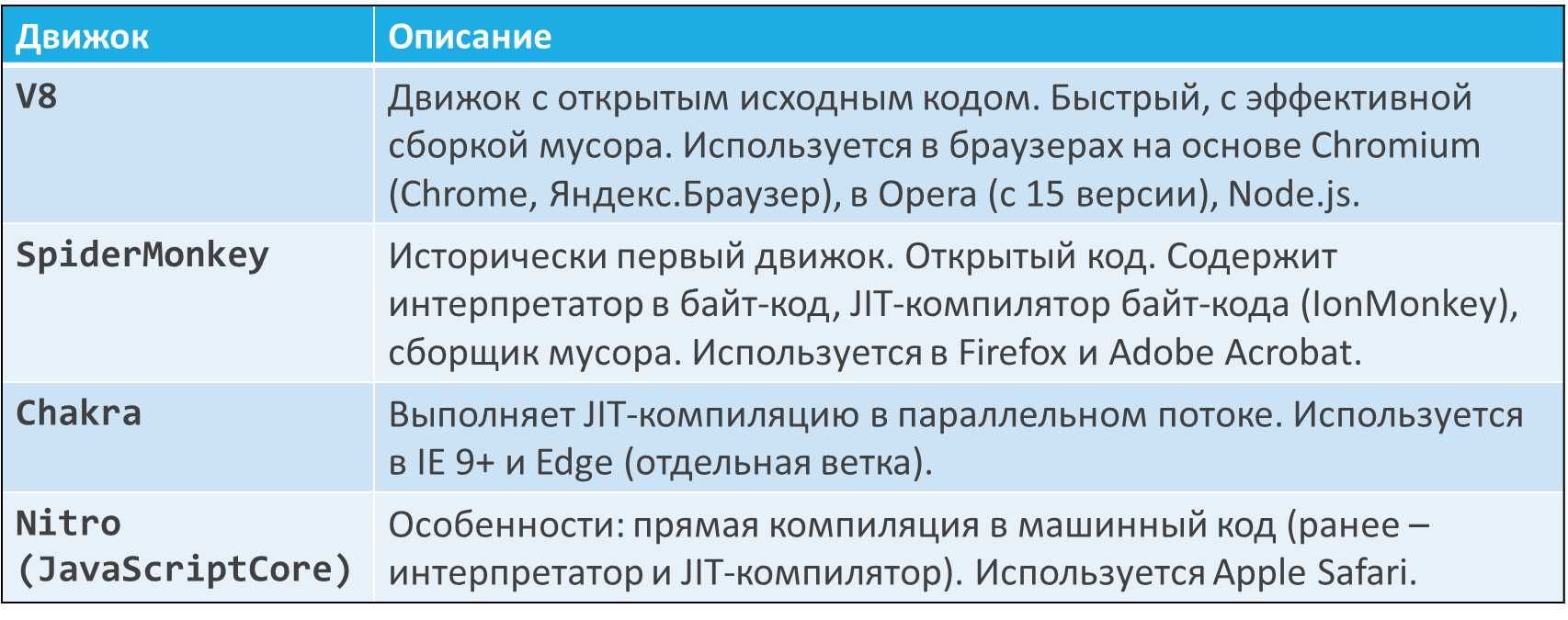


# 2.Движок JavaScript (JavaScript engine)

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

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

**Сводка по движкам JavaScript**



**Как использовать ES2017**

Как использовать везде:

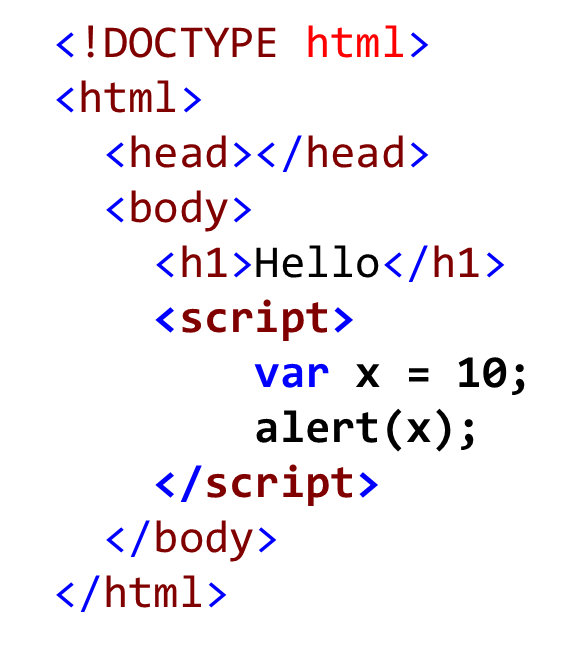
* 1. При помощи *транспайлеров* (синтаксис).
  2. При помощи *полифиллов* (новые объекты и функции).

**Babel** (или **Babel**.**js**) – популярный транспайлер для **ES2015**-**ES2017** (предлагает для использования и полифилл).

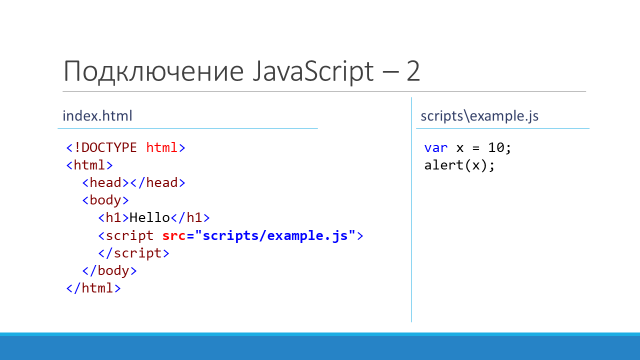
# 3.IDE для JavaScript. Подключение JavaScript

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

**Подключение JavaScript – 1**

****

**Подключение JavaScript – 2**

****

# 4.Лексическая структура JavaScript

*Лексическая структура* языка программирования – набор **элементарных правил**, определяющих как пишутся программы на этом языке.

* Для записи скриптов используется Unicode
* Исходный код – набор *инструкций* и *комментариев*
* Комментарии бывают *строчные* и *блочные*
* Пробелы и пробельные символы игнорируются
* Комментарии игнорируются
* Чувствительность к регистру при записи инструкций

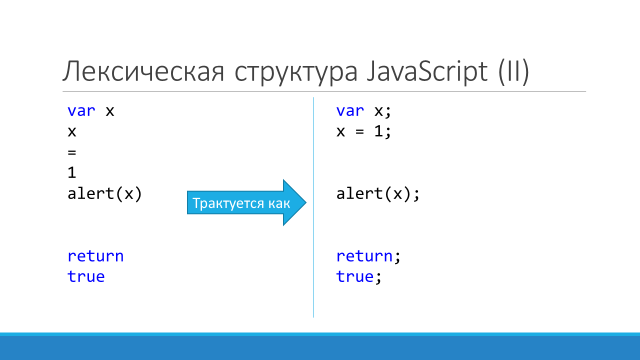
*Пример:*

* // это значит "переменная" (на китайском языке)
* var 变量 = 10;
* /\*
* пробелы, табуляция - игнорируются
* \*/
* alert(变量);

Для отделения инструкций используется точка с запятой.

**JS** трактует переход на новую строку как точку с запятой в следующих случаях:

* сразу после ключевых слов **return**, **break**, **continue**
* перед операторами ++ и --
* если следующий непробельный символ не может быть интерпретирован как продолжение текущей инструкции



Вывод: рекомендуется **всегда** использовать точку с запятой для разделения инструкций. Точка запятой ставиться после инструкции, но не после блока инструкций в фигурных скобках.

# 5.Строгий режим

В **ECMAScript** **5** появился *строгий режим* (**strict** **mode**).

Программирование в строгом режиме накладывает ряд ограничений, чтобы уберечь программиста от опасных частей языка (те части, которые есть исторически, но лучше, чтобы их не было) и снизить вероятность ошибки.

**Включение строгого режима**

* "use strict"; (или 'use strict';)

В первой строке скрипта – действует на весь скрипт.

В первой строке функции – действует внутри функции.

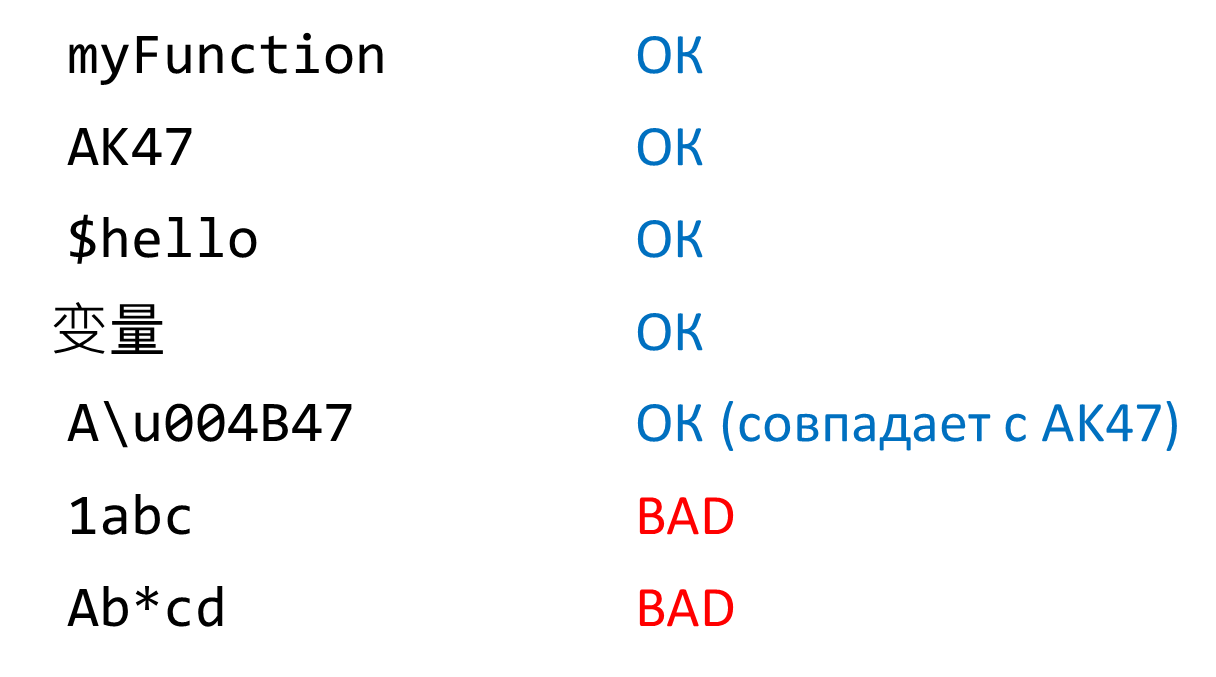
Не в первой строке – ни на что не влияет.

# 6.Идентификаторы в JavaScript. Зарезервированные слова

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

Длина идентификатора стандартом не оговаривается.

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



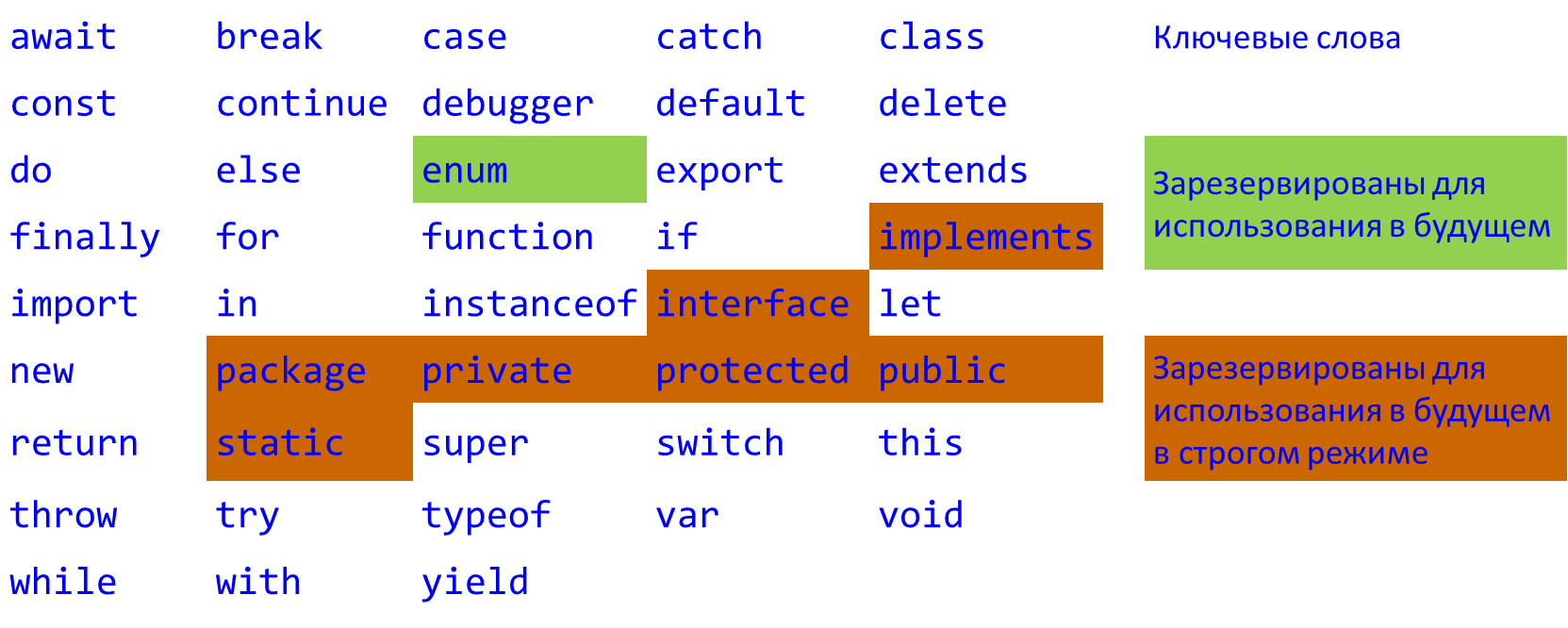
**Зарезервированные слова**

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

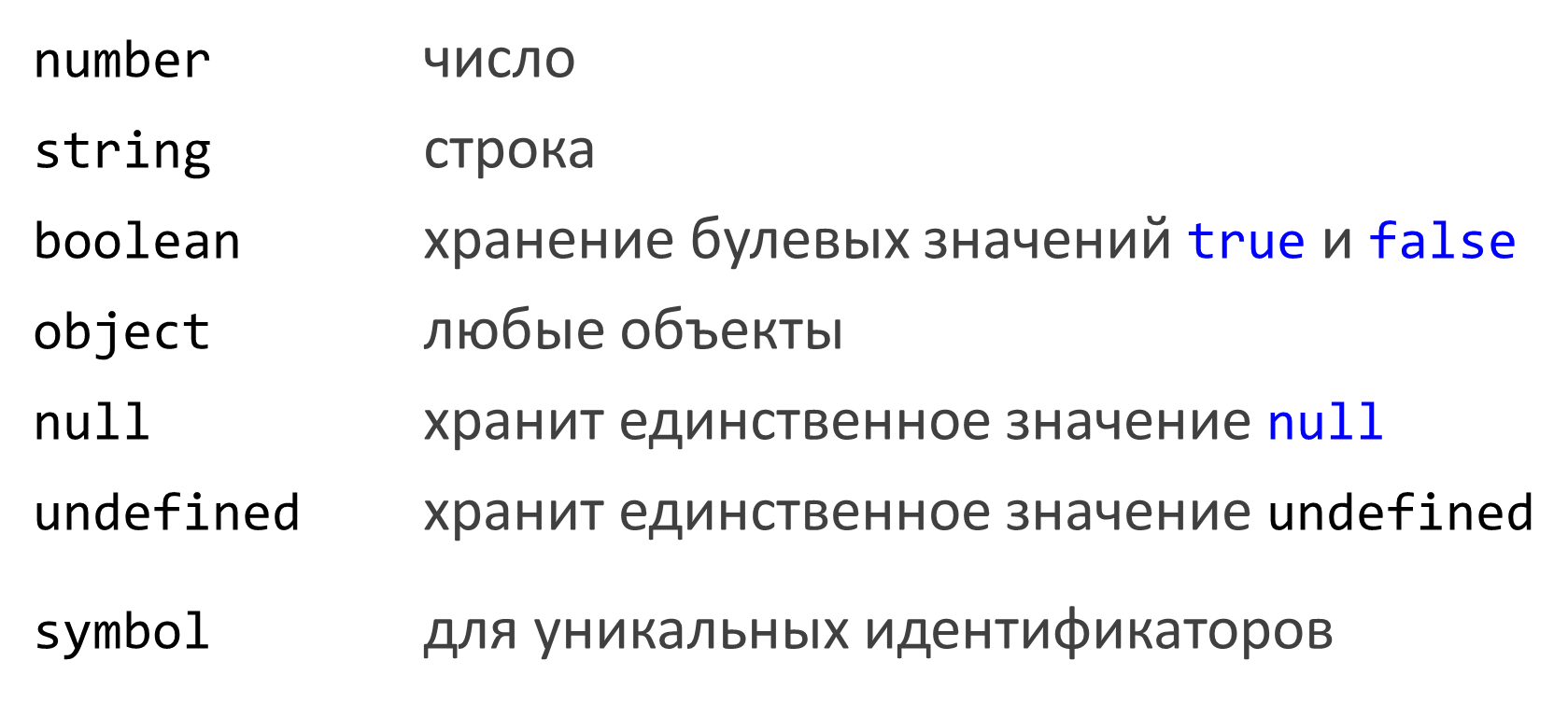
Категории зарезервированных слов:

* + *ключевые слова*,
  + зарезервированы для использования в будущем,
  + литералы для типов **boolean** и **null**.

**Зарезервированные слова ES2017**



# 7.Типы данных в JavaScript



**Тип number**

Это тип для хранения чисел с плавающей запятой в формате **IEEE** **754** (64-битовые значения). **JS** определяет две глобальные переменные **Infinity** и **NaN** для хранения бесконечности и «не-числа».

**Тип** **string**

Значения типа – строки из нуля или более Unicode-символов (кодировка **UTF16**). Значения неизменяемы.

Тип использует семантику копирования значений (**а не ссылок**) при присваивании.

**Тип object**

Значениями этого типа являются любые объекты. *Объект* в **JS** – это набор пар «имя свойства: значение».

Тип использует ссылочную семантику при присваивании. В **JavaScript** массивы и функции – это особые разновидности объектов.

**Типы null и undefined**

Эти два *тривиальных типа* содержат по одному значению:

* тип **null** представляет значение **null**, которое семантически означает «отсутствие объекта»;
* тип undefined представляет значение **undefined** (хранится в одноимённой глобальной переменной) – «значение отсутствует, не определено».

**Тип данных symbol**

Это новый примитивный тип данных – появился в **ES2015**. Представляет уникальные строки(?), которые могут использоваться как имена свойств объекта.

Зачем? Чтобы избежать конфликта имён.

# 8.Литералы в JavaScript. super в литералах. Управляющие символы внутри строки

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

Литерал – это константа, непосредственно включённая в текст скрипта.

**Литералы в JavaScript**

Вот так в **JavaScript** выглядит обычное объявление и инициализация переменной:

* var x = "This is a string variable";

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

**Целые десятичные числа**

Для записи целых **десятичных** чисел используются цифры 0…9. Перед числом допустимы знаки + или –:

* var x = -123;

**Целые шестнадцатеричные числа**

**Шестнадцатеричное** число начинается с **0x** или **0X**. Используются шестнадцатеричные цифры 0…9 a…f A…F. Перед числом допустимы знаки + или –:

* var x = 0x123abc;
* var y = -0XFFF;

**Новые литералы чисел в ES2015**

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

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

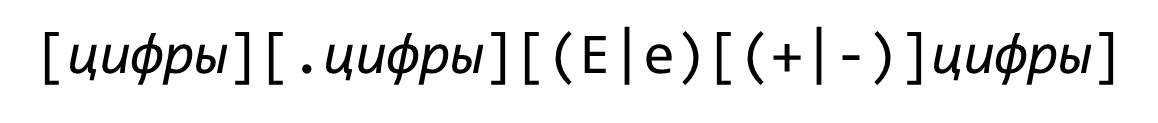
**Замечание о целых числах**

Максимальное целое число, хранимое **точно** **= 253**:

* var biggestInt = 9007199254740992;
* // проверьте, чему равно biggestIntAndOne
* var biggestIntAndOne = 9007199254740993;
* alert(biggestIntAndOne);

**Литералы вещественных чисел**

Синтаксическая форма литерала **вещественных чисел** (используются только десятичные *цифры*):



Перед числом можно указать знак + или –:

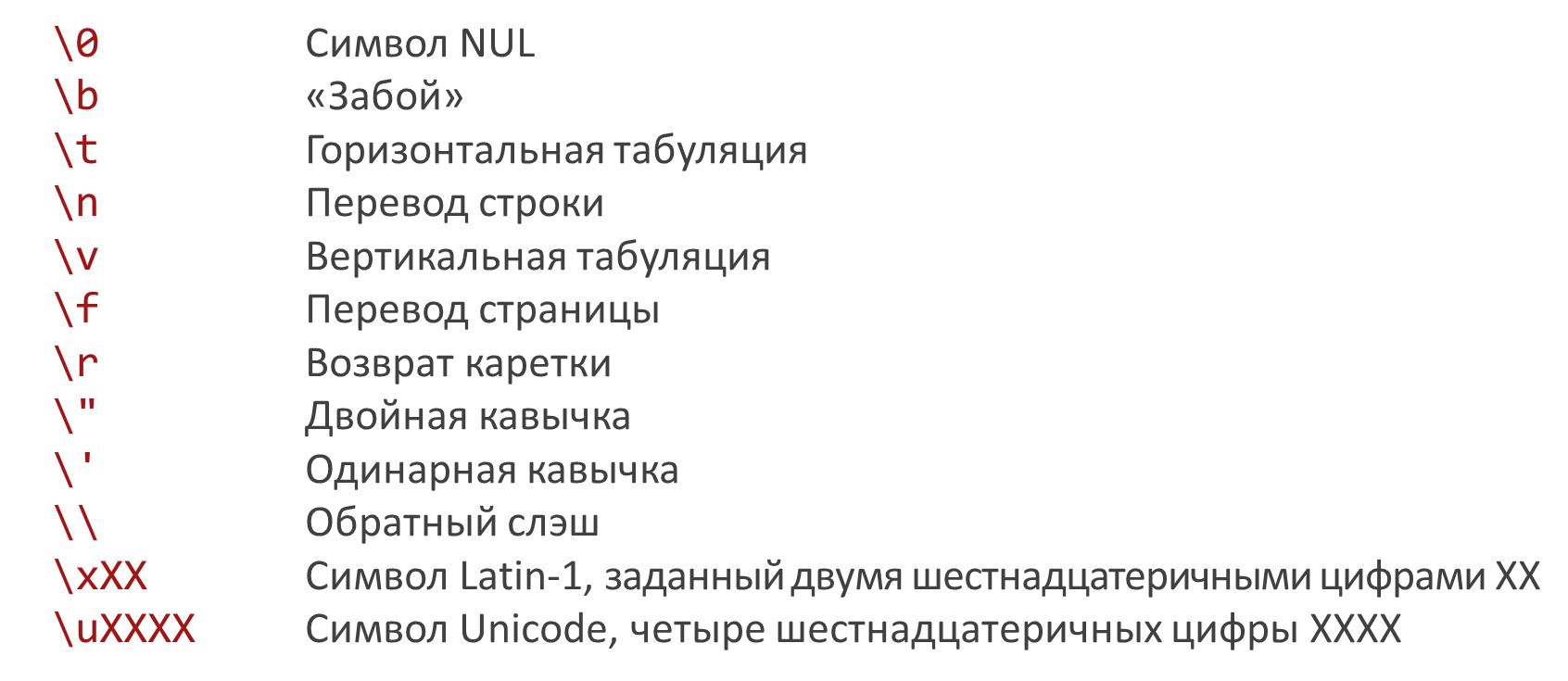
* var x = 3.1415926;
* var y = -4.08e75;

**Строковые литералы**

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

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

**Управляющие символы внутри строки**

****

**Обратный слэш – нюанс**

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

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

**Шаблонные** **литералы**

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

* var str = `template string`;

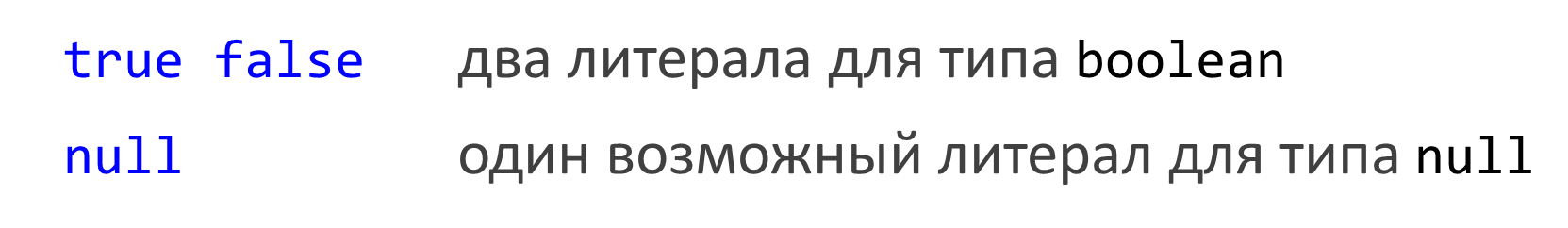
Шаблонный литерал может содержать перевод строки (строка так и отображается – с переводом):

* var str = `template
* string`;

**(!)** При помощи **${…}** в шаблонный литерал можно вставить произвольное выражение:

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

**Литералы для типов boolean и null**



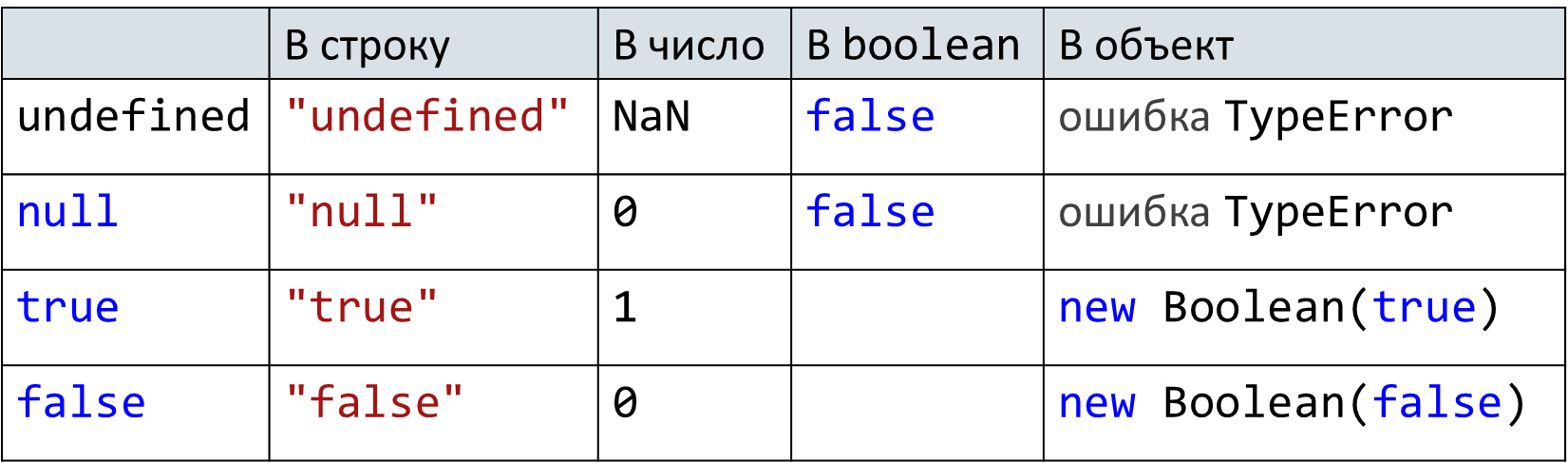
# 9.Преобразование типов данных

Большинство операторов **JavaScript** требуют операндов определённых типов.

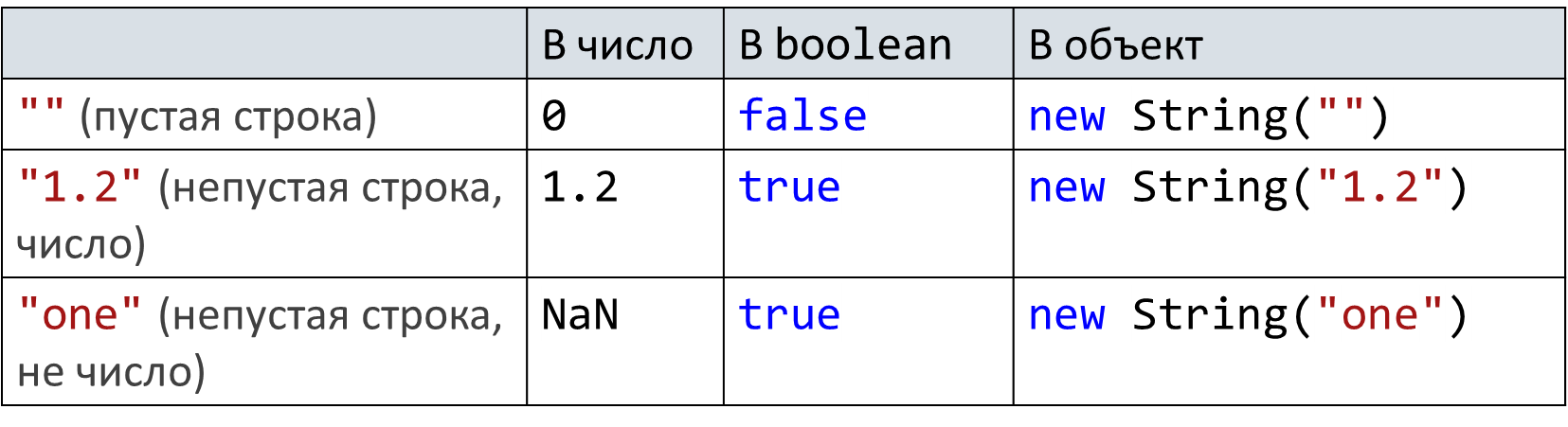
Например, оператор . («точка», доступ к свойству объекта) требует в качестве операнда объект.

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

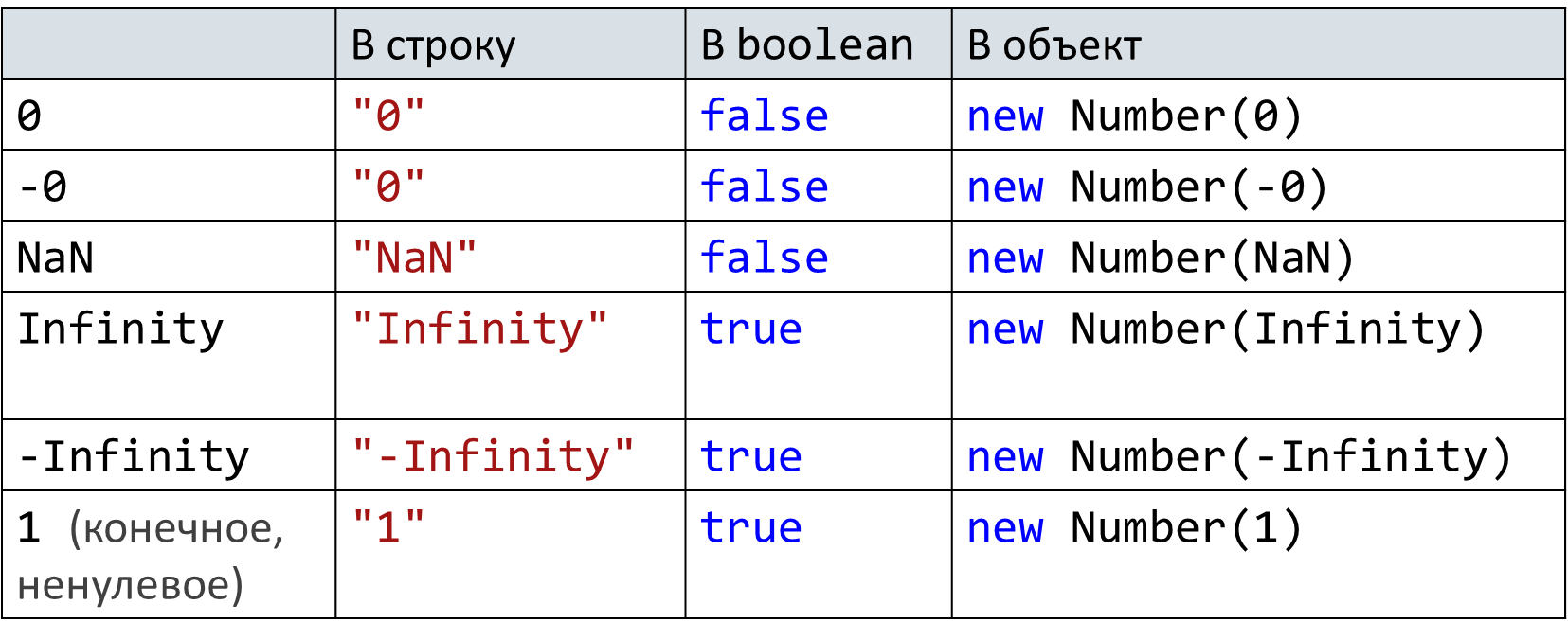
**Преобразование null, undefined, boolean**



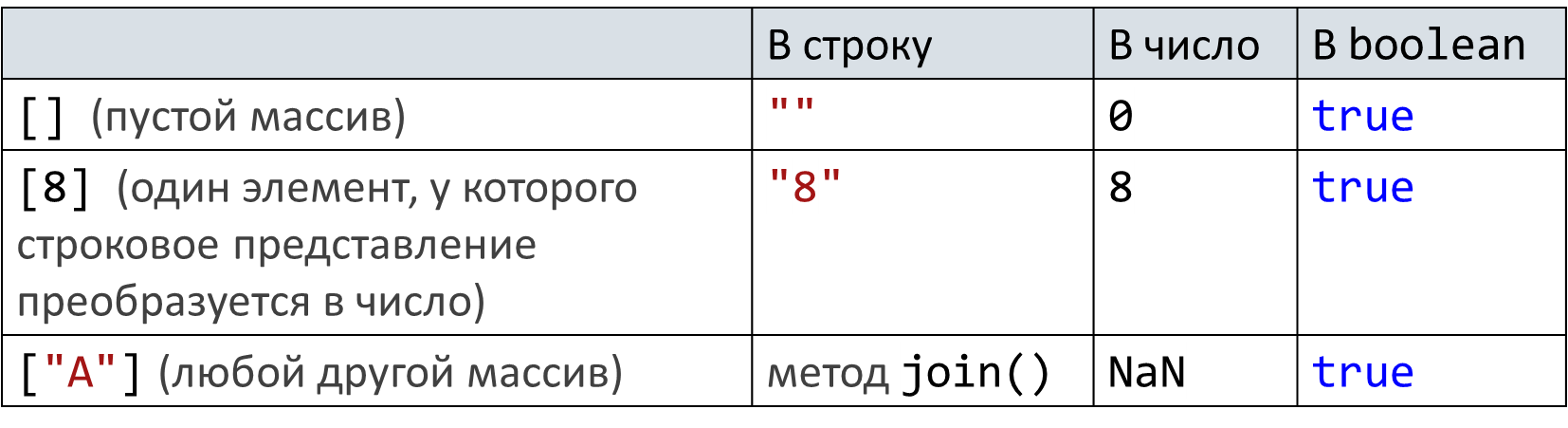
**Преобразование** **строк**



**Преобразование чисел**

****

**Преобразование массивов**

****

Тут есть тонкость одна. Преобразование массива в число выполняется так: массив преобразуется в строку. Затем то, что получилось преобразуется в число.

Если у массива больше чем один элемент, то **join**() будет вставлять запятые и правильного числа в любом случае не получится.

Но если у массива один элемент, который можно преобразовать в строку, преобразуемую в число – всё будет ОК.

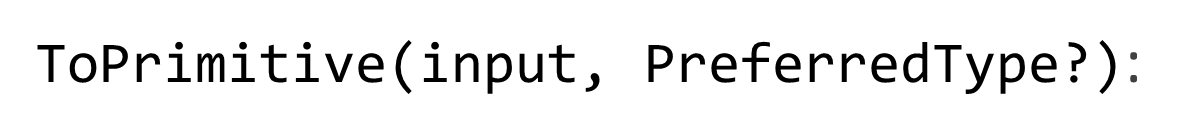
var x = +[8]; // х = 8

var y = +["8"]; // y = 8

**Преобразование объектов**

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

В стандарте описана внутренняя функция



* + **input** – что преобразуем
  + **PreferredType** – целевой тип (необязательный параметр)

*Преобразование объекта в строку (число):*

* + вызвать **ToPrimitive**(), где второй аргумент **string** (**number**);
  + результат функции преобразовать в строку (число).

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

# 10.Алгоритм работы функции ToPrimitive()

Если input – объект, а PreferredType – это string:

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

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

Если **input** – объект, а **PreferredType** не задано:

* + для объектов **Date** полагаем, что **PreferredType** – **string**;
  + для любых других объектов **PreferredType** – это **number**.

Если **input** – примитивное значение, возвращаем input.

# 11.Неявные и явные преобразования. Глобальный объект

Сейчас были описаны *неявные преобразования* типов.

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

**Глобальный объект**

*Глобальный объект* – объект **JavaScript**, автоматически создаваемый при запуске транслятора (т.е. перед началом выполнения скрипта).

Этот объект играет роль глобального контекста.

Его свойства и методы видны и доступны в скрипте «везде» (глобально).

Вот что определяет *нормальный* глобальный объект:

* глобальные свойства (undefined, Infinity, NaN, …)
* глобальные функции (isNaN(), parseInt(), eval(), …)
* функции-конструкторы (RegExp(), String(), …)
* глобальные объекты (Math, JSON, …)

Если скрипт запускается на *странице браузера*, глобальный объект помещает себя в своё собственное свойство с именем **window**:

* var x = parseInt("3"); // так короче
* var y = window.parseInt("3"); // но можно и так

# 12.Выражения. Разновидности выражений. Первичные выражения.

*Выражение* – последовательность символов **JavaScript**, которая может быть **вычислена** транслятором для получения **значения**.

Иногда сам **процесс вычисления** важнее полученного значения, которое может вовсе не использоваться – в этом случае говорят об использовании **побочного эффекта** выражения.

**Разновидности выражений JavaScript**

1. Первичные выражения
2. Инициализаторы
3. Выражения обращения к свойствам
4. Выражения вызова
5. Выражения создания объектов
6. Выражения с операторами

**Первичные выражения**

*Первичные выражения* не включают более простых подвыражений:

* 1. Литералы для типов number, string, boolean, null
* 2. *Литералы регулярных выражений*:
* var x = **/**[0-9]\***/**; // var x = new RegExp("/[0-9]\*/");
* 3. **this** – контекст вызова функции («текущий объект»).
* 4. Ссылки на переменные.

# 13.Инициализаторы, их разновидности

1. Порождающие объект («литералы объектов») –*значением выражения является новый объект*:

* var person = {name: "Alex"};

2. Порождающие массив («литералы массивов»):

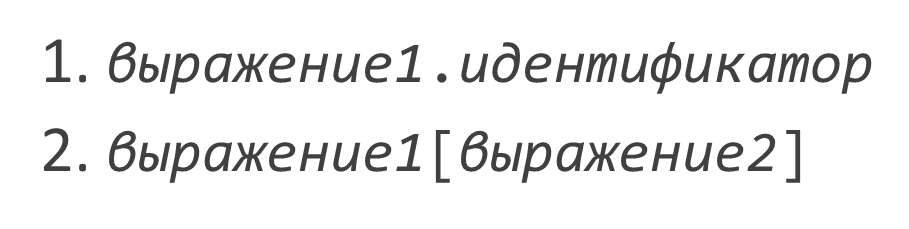
* var arr = [1, 2+3];

3. Определяющие функцию («литералы функций»).

* var f = function (x) { return x > 0; } ;

**Выражения обращения к свойствам**

Такие выражения имеют две формы:



Здесь *выражение1* вычисляется и преобразуется в объект, *выражение2* вычисляется и преобразуется в строку. Значением выражения является *значение свойства* или *элемент массива*.

*Пример:*

* // переменные person и arr определили ранее
* person.name // значение = "Alex"
* person["name"] // значение = "Alex"
* arr[1] // значение = 5

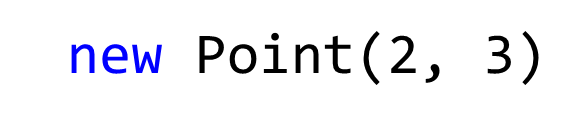
**Выражения** **вызова**

*Выражение вызова* состоит из:

* выражения, возвращающего функцию;
* круглых скобок;
* в скобках через запятую – выражения для аргументов.

*Значением* выражения является то, что вернула функция.

**Выражения создания объекта**



Выражение создаёт новый объект, затем передаёт этот объект указанной функции (***конструктору***) в качестве **this**. ***Значением*** выражения является:

– созданный объект (если конструктор ничего не возвращает или возвращает примитивное значение)

*или*

– объект, возвращённый конструктором.

# 14.Операторы. Инкремент и декремент

Использование операторов – основной способ конструирования выражений.

Характеристики любого оператора:

* + приоритет;
  + «арность» (т.е. количество операндов);
  + ассоциативность (слева направо или справа налево);
  + типы операндов.

Ассоциативность слева направо (**L**):

* *w = x – y – z w = (x – y) – z*

Ассоциативность справа налево (**R**):

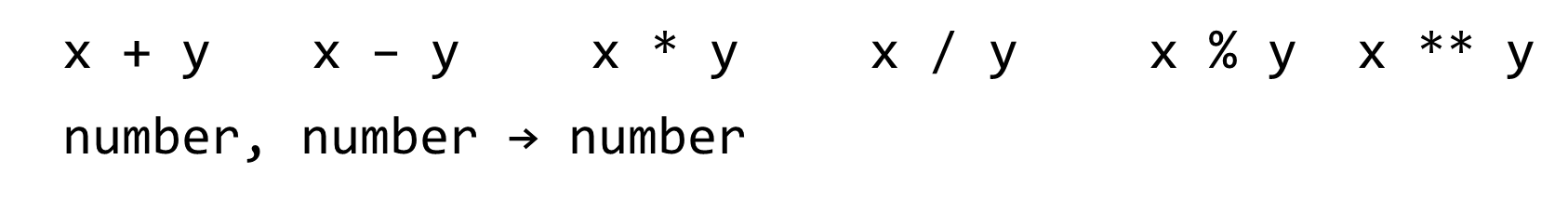
* *w = x = y = z w = (x = (y = z))*

**«Левостороннее выражение»**

Этот термин встречается в спецификации и на слайдах. Он означает одно из трёх:

* 1. переменная;
  2. свойство объекта;
  3. элемент массива.

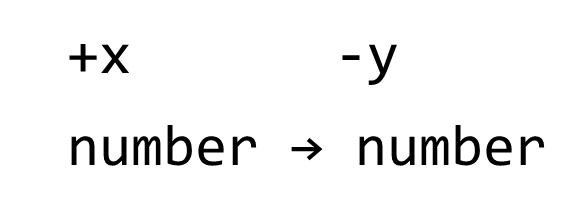
**Арифметические операторы**

****

У этих операторов традиционная семантика. Их операнды преобразуются к числам, результатом является число. Возведение в степень – правоассоциативно.

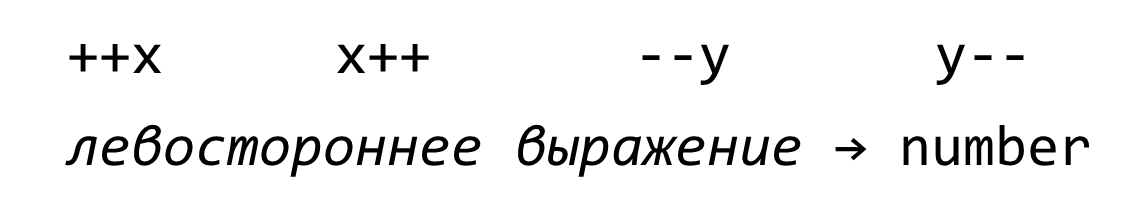
Любая арифметическая операция с **NaN** даёт **NaN**.

**Унарный плюс и унарный минус**



Польза операции «унарный плюс» в том, что она выполняет преобразование операнда в число.

**Инкремент и декремент**



Оператор **++** преобразует свой операнд в число, прибавляет единицу, помещает новое значение обратно в операнд.

Оператор -- работает по аналогичной схеме.

Возвращаемое значение операторов зависит от позиции:

* x++ значение операнда **после преобразования в число**, но **до увеличения**.
* ++x значение операнда **после увеличения**.

Аналогично для декремента.

# 15.Конкатенация vs. Сложение. Поразрядные битовые операторы

Если хотя бы один из операндов оператора + является строкой, выполняется конкатенация строк.

Если операндом является объект, для него вызывается **ToPrimitive**() **с одним аргументом**. И поведение операции + зависит от возвращённого значения.

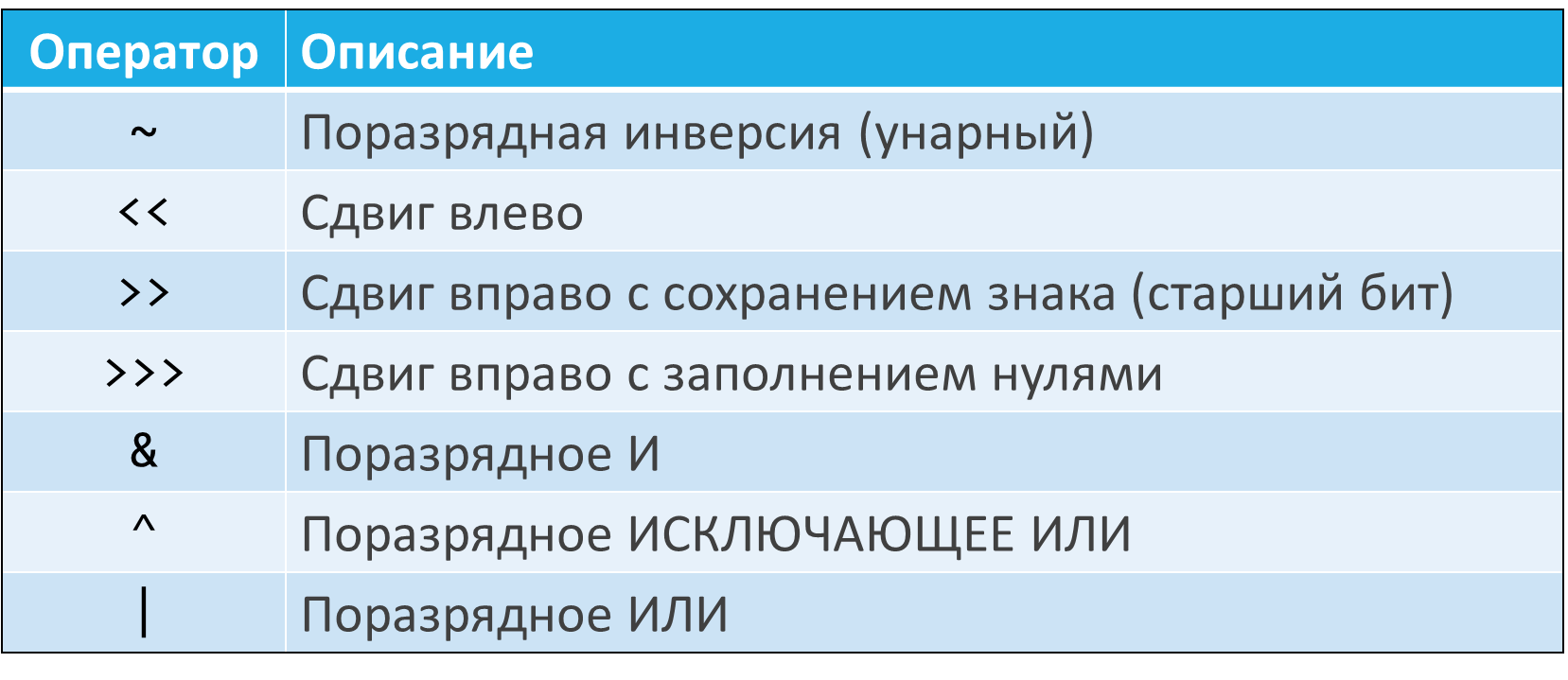
**ToPrimitive**() с одним аргументом вызывается только для операции +. Для остальных арифметических операций вызывается **ToPrimitive(obj, number).**

Напоминаем, что **ToPrimitive**() с одним аргументом эквивалентно вызову **ToPrimitive**(**obj**, **number**) для всех объектов, кроме объектов **Date**.

**Конкатенация vs. сложение**

****

**Поразрядные битовые операторы**

****

Битовые операторы работают с **тридцатидвухразрядными целыми числами**. Если нужно, выполняется преобразование операндов к числу, отбрасывание дробной части и «лишних» старших битов. Значения **NaN** и **Infinity** преобразуются в 0.

**Дополнительно:** у операторов сдвига второй операнд должен быть пятиразрядным (0..31).

# 16.Логическое И/ИЛИ/НЕ. Проверка равенства, идентичности.

**Логическое И**



Вычисляем значение выражения **x**.Если это значение, преобразованное в **boolean**, равно **false**, возвращаем его (не **false**, а именно непреобразованное значение выражения **x**).Иначе вычисляем и возвращаем **y**.

*Пример:*

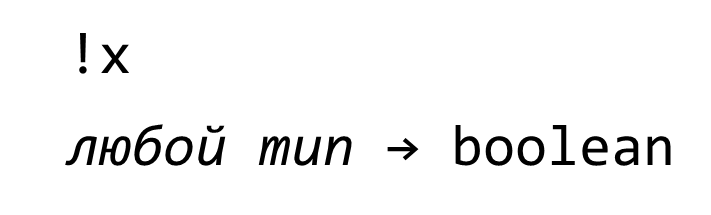
* var o = { x: 1 };
* var p = null;
* var a = o && o.x; // a = o.x (o – "истинное")
* var b = p && p.x; // b = null (p – "ложное")

**Логическое ИЛИ**



Вычисляем значение выражения **x**. Если это значение, преобразованное в **boolean**, равно **true**, возвращаем его (не **true**, а именно непреобразованное значение выражения **x**).Иначе вычисляем и возвращаем **y**.

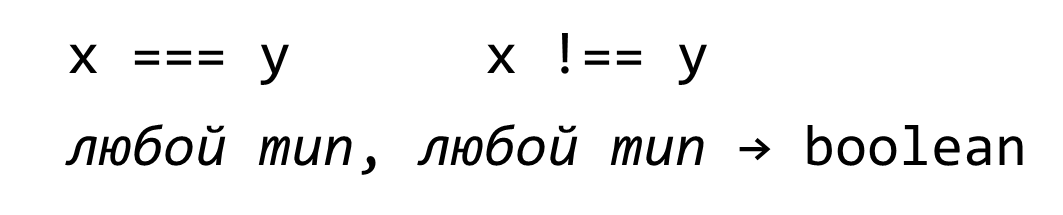
**Логическое НЕ**



Операнд преобразуется в логическое значение, которое затем инвертируется и возвращается.

Дважды применив этот оператор, можно преобразовать любое значение **x** в его логический эквивалент.

**Проверка идентичности**



Это проверка идентичности значений, выполняемая без преобразования типов операндов. Является более строгой в сравнении с операторами **==** и **!=.**

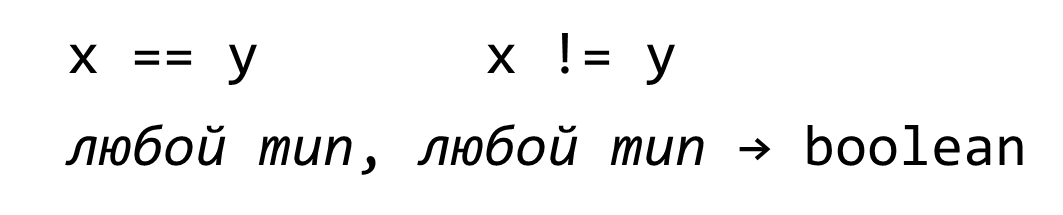
**Не идентичны**:

* операнды имеют разные типы
* один или оба операнда равны **NaN**

**Идентичны**:

* null и null; **undefined** и **undefined**; true и true; false и false
* Оба операнда являются равными числами или строками
* 0 и -0
* Оба операнда ссылаются на один и тот же объект

**Проверка равенства**



Проверка на равенство с предварительной попыткой приведения типов.

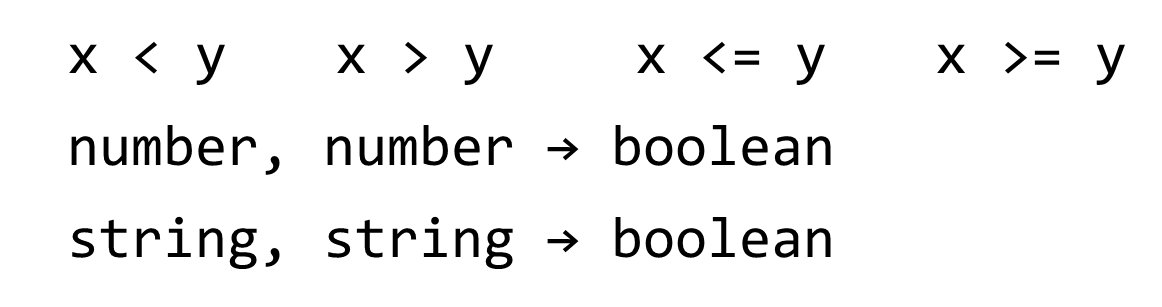
Если операнды имеют одинаковый тип, выполняется проверка на идентичность. **null** равно **undefined**.

Строка и число: строка преобразуется в число, затем выполняется сравнение чисел.

Если операнд имеет тип **boolean**: преобразуем его в число, затем выполняем сравнение.

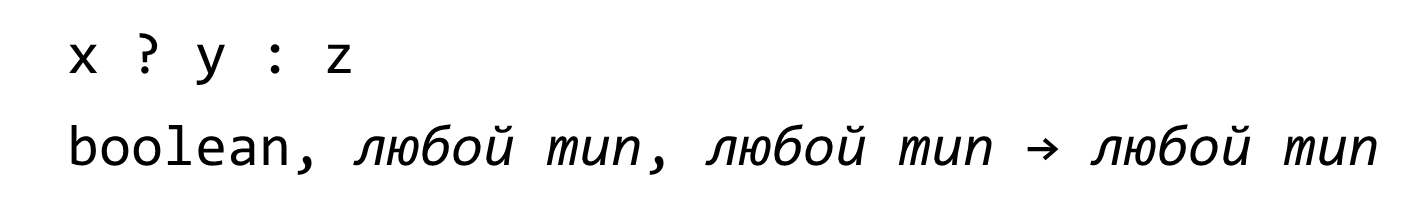
Если ровно один операнд является объектом, вызвать для него **ToPrimitive**() с одним аргументом.

# 17.Операторы сравнения. Условный оператор.



Лексикографическое сравнение строк выполняется, только если оба операнда являются строками. В противном случае выполняется сравнение чисел.

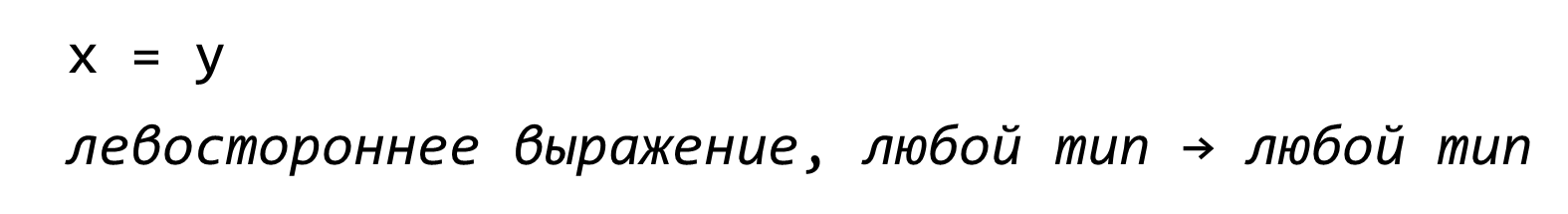
**Условный оператор**



Значение выражения x преобразуется к типу **boolean**. Если получилось **true** – вычисляем и возвращаем y, иначе вычисляем и возвращаем **z**.Это единственный тернарный оператор в **JavaScript**

# 18.Оператор typeof. Оператор instanceof. Оператор in.

**Оператор присваивания**



Оператор выполняет присваивание переменной или свойству **x** значения **у**. Само выражение тоже равно **y**.

Существуют «присваивания с операцией»: += -= \*= /= %= <<= >>= >>>= &= |= ^=.

**Оператор typeof**



Оператор возвращает строку с именем «типа» операнда.

|  |  |
| --- | --- |
| x | typeof x |
| undefined | "undefined" |
| null | "object" |
| true *или* false | "boolean" |
| *любое число* | "number" |
| *любая строка* | "string" |
| *любая функция* | "function" |
| *любой объект, не являющийся функцией* | "object" |
| *значение symbol* | "symbol" |

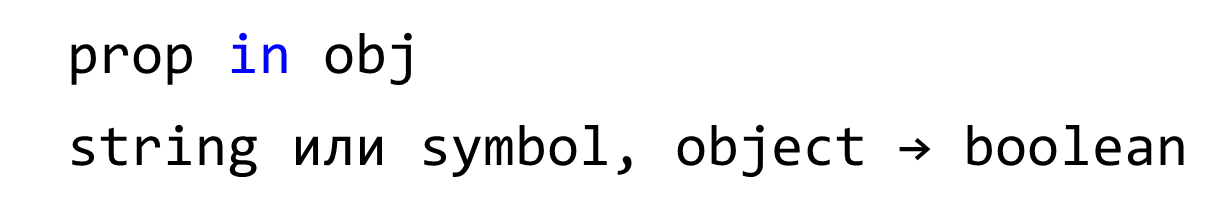
**Оператор instanceof**



Оператор возвращает **true**, если объект **obj** (или его прототип, или прототип прототипа, …) был создан с использованием функции-конструктора **Constr**.

* var d = new Date(); // конструктор Date()
* var a = d instanceof Date; // true
* var b = d instanceof Object; // true

**Оператор in**



Оператор возвращает **true**, если объект **obj** (или его прототип) содержит свойство с именем **prop**.

*Пример:*

* var point = {x: 1, y: 1};
* "x" in point; // => true
* "z" in point; // => false
* "toString" in point; // => true
* var data = [7, 8, 9];
* "0" in data; // => true
* 1 in data; // => true
* 3 in data; // => false

# 19.Оператор delete. Оператор void. Оператор расширения (spread operator)

**Оператор delete**



Пытается удалить свойство из объекта, индекс (и элемент) из массива или переменную. Если свойство существует и не является удаляемым, возвращает **false**, иначе – **true**. Часто возвращаемое значение игнорируется.

*Пример:*

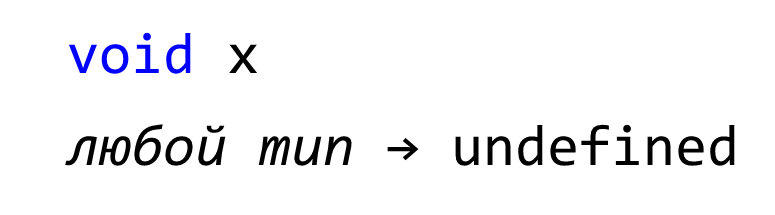
* var o = {x: 1, y: 2}; // определить объект
* delete o.x; // удалить его свойство
* var a = "x" in o; // => false
* var a = [1, 2, 3]; // создать массив
* delete a[2]; // удалить элемент (3)
* var b = 2 in a; // => false (нет индекса 2)

**Оператор delete в строгом режиме**

1. При попытке применить delete к переменной генерируется ошибка **SyntaxError**.

2. При попытке удалить несуществующее свойство генерируется ошибка **TypeError**.

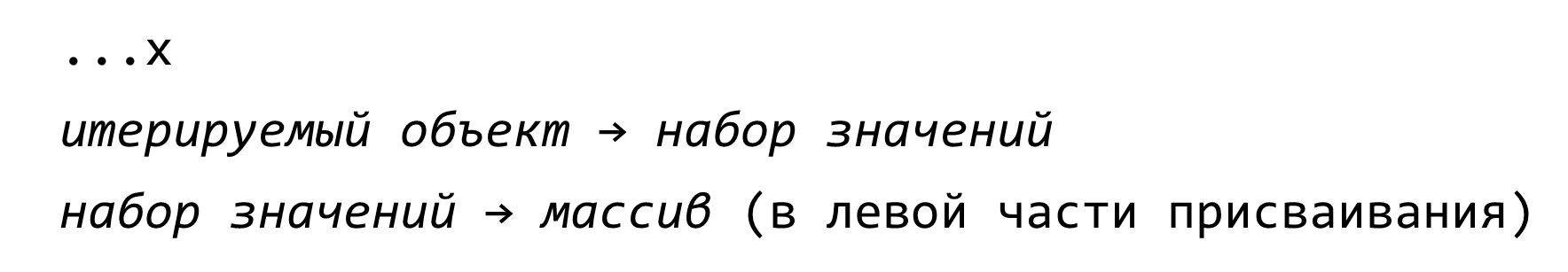
**Оператор void**



Оператор вычисляет значение **x**, отбрасывает его и возвращает **undefined**. Часто применяется в **URL**-адресах вида **javascript**:, где позволяет вычислить выражение ради его побочного эффекта:

* <a href="javascript: void window.open();">Новое окно</a>

**Оператор расширения (spread operator)**



Оператор возвращает набор значений итерируемого объекта. Применяется при вызовах функций и в «литералах массивов». Возможно использование при деструктурирующем присваивании.

*Пример:*

* var a = [1, 2, 3]; // массив – итерируемый объект
* var b = [...a, 4, 5]; // b = [1, 2, 3, 4, 5]
* function f(x, y, z) { }
* f(...a); // аналогично f(1, 2, 3)
* [k, t, ...r] = [1, 2, 3, 4]; // r = [3, 4]

# 20.Конструкция rest. Оператор , (запятая).

**Конструкция rest**

**Внимание**: есть синтаксическая конструкция **rest**, применяемая при определении функции.

Выглядит как оператор расширения, но «собирает» отдельные аргументы в массив.

* function f(...x) { }
* f(1, 2, 3); // в функции f x - массив [1, 2, 3]

**Оператор , (запятая)**



Оператор вычисляет левый операнд, затем вычисляет правый операнд и возвращает значение правого операнда. Обычно используется в циклах для работы с несколькими счётчиками:

* for (i = 0, j = 10; i < j; i++, j--) alert(i + j);

# 21.Приоритет операторов и выражений

1. Применение скобок ( и )
2. Обращение к свойствам, new (со списком аргументов)
3. Вызов функций, new (без списка аргументов и скобок)
4. Постфиксный инкремент и декремент
5. Префиксные ++ и --, унарные + и –, ~, !, delete, typeof, void
6. \*\*
7. \*, /, %
8. Бинарные + и –
9. <<, >>, >>>
10. <, <=, >, >=, instanceof, in
11. ==, !=, ===, !==
12. &
13. ^
14. |
15. &&
16. ||
17. ?:
18. =
19. ...
20. ,

# 22.Деструктурирующее присваивание. Деструктуризация массива. Деструктуризация объекта.

**Деструктурирующее** **присваивание** (destructuring assignment) – расширение синтаксиса оператора присваивания. Позволяет присвоить значения нескольким левосторонним выражениям на основе данных массива (итерируемого объекта) или объекта.

*Пример:*

* [firstName, lastName] = ["Alex", "Volosevich"];
* // firstName = "Alex", lastName = "Volosevich"
* a = 10, b = 20;
* [a, b] = [b, a]; // a = 20, b = 10

**Деструктуризация массива**

При деструктуризации массива могут использоваться:

* Запятые для отбрасывания значений
* Значения по умолчанию для переменных (без них – **undefined**). В качестве значения по умолчанию можно использовать выражение (включая вызов функции).
* Оператор расширения ... для получения в переменную остатка массива в виде массива.

*Пример:*

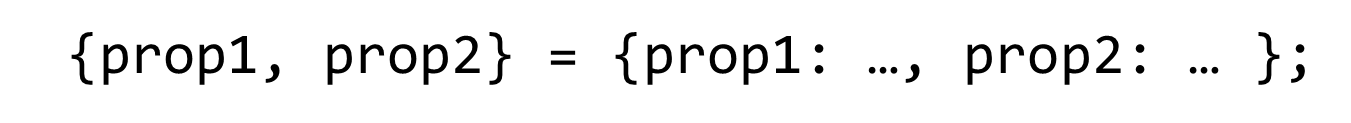
* // пропускаем первый и третий, игнорируем пятый
* [ , a, , b] = [1, 2, 3, 4, 5]; // a = 2, b = 4
* [x, y = 10, z] = [1]; // x = 1, y = 10, z = undefined
* [firts, ...rest] = [1, 2, 3, 4, 5];
* // rest – это массив [2, 3, 4, 5]

Термин «деструктуризация массива» используется в спецификации, но он не совсем точен. На самом деле деструктуризация работает для итерируемого объекта:

* [a, b] = "foo"; // a = "f"; b = "o"

**Деструктуризация объекта**

Базовый синтаксис:



Справа – **существующий** **объект**, который разбиваем.

Слева – **список** **переменных**, соответствующих свойствами (по имени).

*Пример:*

* options = {
* title: "Menu",
* width: 100,
* height: 200
* };
* // три переменные получат значения
* let {title, width, height} = options;

НЮАНСЫ:

Если надо присвоить свойство в переменную с другим именем, используем синтаксис свойство: **переменная**.

* // переменные w, h, title
* let {width: w, height: h, title} = options;

Если каких-то свойств в объекте нет, можно указать значение по умолчанию через знак равенства:

* let {width = 100, height: h = 200} = options;

# 23.Инструкции. Пустые инструкции.

**Инструкция** (**statement**) – автономная часть скрипта, которая может быть выполнена (иначе – команда).

Любой скрипт представляет собой набор инструкций.

Для разделения инструкций **JS** использует символ **;** или переход на новую строку (нюансы обсуждались ранее).

**Пустые инструкции**

Иногда синтаксис **JavaScript** требует в определённом месте наличия инструкции (любой).

А по смыслу скрипта инструкция не нужна.

В этом случае следует использовать пустую инструкцию (вроде и инструкция, но ничего не делает).

Пустая инструкция в **JavaScript** – это символ **;**

# 24.Составные инструкции. Инструкция var.

Иногда синтаксис **JavaScript** требует в определённом месте наличия единственной инструкции.

А по смыслу скрипта инструкций надо несколько.

В этом случае следует использовать составную инструкцию (несколько инструкций, рассматриваемых как одна). Чтобы в **JavaScript** создать составную инструкцию, применяются фигурные скобки **{ }**.

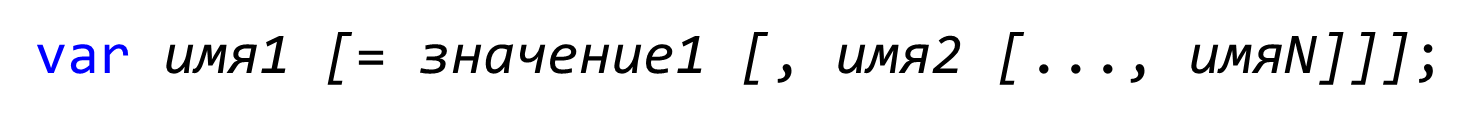
**Инструкции-выражения**

Простейший вид инструкции – это выражение, обычно с побочным эффектом (не забываем про точку с запятой):

* greeting = "Hello, User!"; // присваивание
* counter++; // инкремент (и декремент)
* delete obj.prop; // оператор delete
* alert(greeting); // вызов функции

**Инструкция var**

Инструкция **var** позволяет объявить одну или несколько переменных (опционально – с начальным значением):



Если значение не указано, подразумевается **undefined**.

Инструкция **var** может размещаться внутри функции (локальная переменная функции) или вне функции – на верхнем уровне скрипта (глобальная переменная).

*Пример:*

* var x = 10; // глобальная переменная
* var y; // неинициализированная переменная
* function f() {
* var z = "Alex"; // локальная переменная функции
* }
* // переменная i - счётчик цикла
* for(var i = 0; i < 10; i++) alert(i);

**Инструкция var – ловушка!**

|  |  |
| --- | --- |
| var x = y = 10; | |
| Наверное, это означает  var y = 10;  var x = 10; | **Нет!** Это вот что:  y = 10;  var x = 10;  Дальше станет понятно, что это **очень разные вещи**. |

В JS при объявлении переменной не указывается тип, так как переменные могут хранить значения *любых типов*:

* var x = 10; // сначала храним в x число
* x = "Alex"; // затем строку
* x = [2, 3, 5]; // затем массив (объект)

# 25.Область видимости переменной. Глобальная область видимости. Локальная область видимости.

*Область видимости переменной* (**scope**) – та часть скрипта, для которой эта переменная определена.

В **JavaScript** область видимости бывает только *глобальной* и *локальной*. Область видимости связана с **функциями** (а не с блоками кода, как в других языках).

**Глобальная область видимости**

Переменная, объявленная вне функции, называется глобальной. Она имеет глобальную область видимости – определена для всего скрипта.

* var x = 10; // глобальная переменная x
* function f() {
* alert(x); // x видна везде в скрипте
* }

**Локальная область видимости**

Переменные, объявленные в функции, определены только в её теле. Они называются локальными и имеют локальную область видимости.

Параметры функций также считаются локальными переменными. Локальная переменная имеет преимущество перед одноимённой глобальной переменной!26.Подъём локальной переменной. Подъём глобальной переменной.

Внутри функции локальные переменные, объявленные при помощи **var**, видимы даже **до строки с объявлением**! Выглядит так, будто все объявления переменных внутри функции транслятор «подымает» к началу функции (**variable** **hoisting**).

<script>

// выводит NaN

alert(x + 3);

var x = 10;

</script>

<script>

// Uncaught ReferenceError:

// x is not defined

alert(x + 3);

</script>

<script>

var x = 10;

</script>

Правда при таком «подъёме» начальные значения не учитываются.

<script>

// Uncaught ReferenceError:

// x is not defined

alert(x + 3);

</script>

<script>

var x = 10;

</script>

*Пример:*

<script>

// выводит NaN

alert(x + 3);

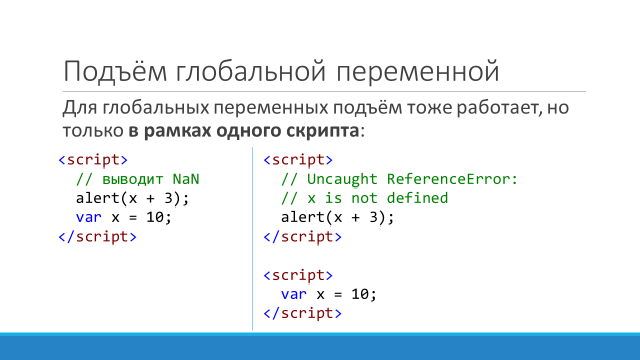
var x = 10;

</script>

* var scope = "global";
* function f() {
* alert(scope); // выведет "undefined"
* var scope = "local"; // инициализация здесь
* alert(scope); // выведет "local"
* }
* var scope = "global";
* function f() {
* var scope; // примерно так
* alert(scope); // поступает
* scope = "local"; // транслятор
* alert(scope);
* }

**Подъём глобальной переменной**

Для глобальных переменных подъём тоже работает, но только в рамках одного скрипт а:

****

**Следствие variable hoisting – 1**

Объявление переменной (но не инициализация) работает, даже если делается в «недосягаемых» местах:

* function f() {
* alert(scope); // выведет "undefined"
* if (false)
* {
* var scope = "local"; // инициализация не работает!
* }
* alert(scope); // выведет "undefined"
* }

**Следствие variable hoisting – 2**

Переменную можно объявлять повторно. Без инициализации она даже не потеряет своего значения:

* var x = 10;
* alert(x); // выведет "10"
* var x; // повторное объявление
* alert(x); // всё равно выведет "10"

# 27.Переменные как свойства. Объявление переменных: let и const

С определёнными оговорками можно считать, что объявлении переменной создаёт одноимённое свойство у глобального объекта (глобальная переменная) или у объекта-функции (локальная переменная).

Но это особое свойство – оно не удаляется оператором **delete**.

В скрипте можно инициализировать «переменную», «объявленную» без **var**:

* scope = "global";
* alert(scope);

В этом случае всегда создаётся удаляемое свойство глобального объекта. Кстати, так нельзя сделать в строгом режиме!

*Использование* переменной (в смысле – чтение её значения) **без объявления и инициализации** ведёт к ошибке **ReferenceError**:

* // пусть scope ранее нигде не упоминали в скрипте
* alert(scope); // ошибка ReferenceError

**Объявление переменных: let и const**

В ES2015 предусмотрены новые способы объявления переменных: через **let** и **const** вместо **var**:

* + Область видимости переменной **let** – блок {...}.
  + Переменная **let** видна только после объявления, *поднятие* (variable hoisting) не работает.
  + Объявление **const** задаёт переменную, которую нельзя менять. В остальном **const** полностью аналогично **let**.

*let – Пример:*

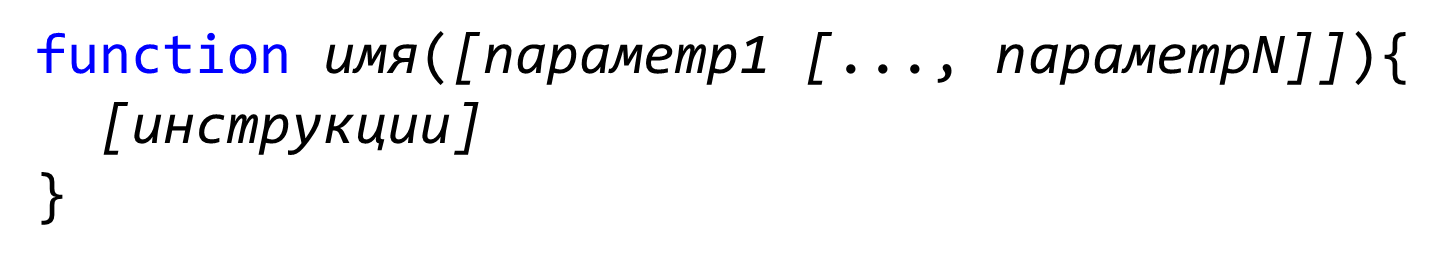
* let x = 5;
* if (true) {
* let x = 10;
* alert(x); // 10 (внутри блока)
* }
* alert(x); // 5 (вне блока значение не изменилось)

*const – Пример:*

* function getConst(x) {
* return x \* 2;
* }
* const y = 20;
* const z = getConst(10);
* y = 1; // SyntaxError: "y" is read-only

# 28.Инструкция function. Инструкция if.

Инструкция **function** служит для определения функции:



Круглые и фигурные скобки обязательны (даже если нет параметров, даже если в теле функции нет инструкций или одна инструкция)!

**Инструкция function – вложение**

Инструкция **function** может располагаться на «верхнем уровне» скрипта или быть вложенной в функцию.

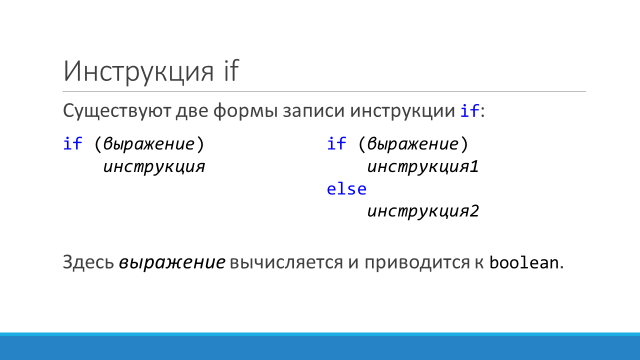
В любом случае, стандарт не допускает вложение этой инструкции в ветвления, циклы и тому подобное! (нужен «первый уровень» вложенности).

Как и в случае **var**, инструкцию **function** можно рассматривать как объявление соответствующего неудаляемого свойства.

Транслятор обеспечивает «подъём» инструкции function, при этом подымается не только объявление, но и инициализация (в отличие от **var**).

**Инструкция if**

Существуют две формы записи инструкции **if**:



Ложными являются следующие значения:

* + **false**
  + **null**
  + **undefined**
  + **""** (пустая строка)
  + **0**
  + **NaN**

Все остальные значения являются истинными.

# 29.Инструкция switch. Инструкция break. Инструкция continue/return.

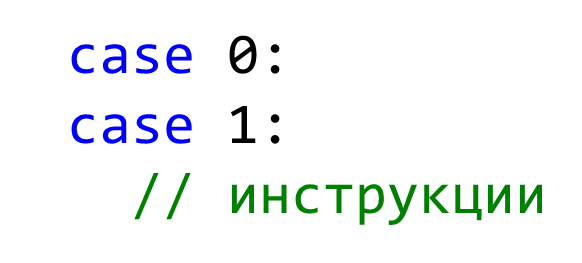
* switch (*выражение*) {
* case *выражение1*:
* // инструкции, соответствующие значению выражение1
* *[*break;*]*
* ...
* case *выражениеN*:
* // инструкции, соответствующие значению выражениеN
* *[*break;*]*
* *[*default:
* // эта часть выполняется при отсутствии совпадений
* *[*break;*]*
* *]*
* }

На первый взгляд похожа на аналоги из других языков. Однако после **case** указываются выражения:

* Сначала вычисляем выражение.
* Затем вычисляем выражение **выражение1**.
* Если значения совпали **(===),** выполняем соответствующий набор инструкций.
* Если не совпали, переходим к вычислению **выражение2**.
* . . .

1. Часть **default** не является обязательной.

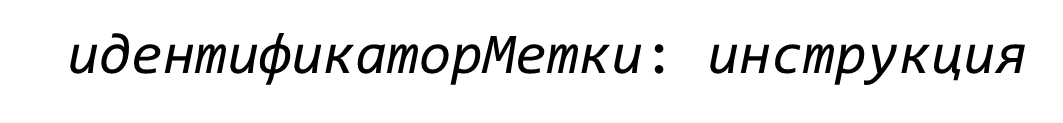
2. Части **case** можно группировать:



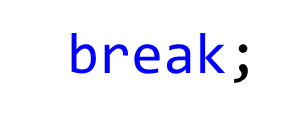
3. Набор инструкций после **case** (и **default**) может, но не обязан, завершаться переходом (**break**, **return**).

**BREAK/CONTINUE**

В скрипте любая инструкция может быть помечена:

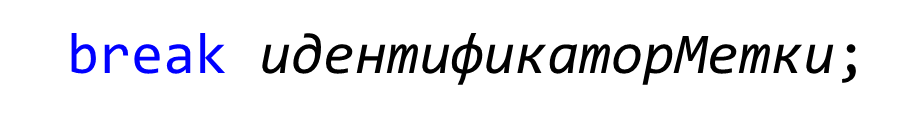
* ******
  1. Метки используются для переходов при помощи инструкций **break** или **continue**.
  2. Переход на метки работает только внутри тех инструкций, к которым метки применяются.

**Инструкция break – первая форма**

****

Эта форма используется для выхода из **switch** или для выхода из цикла (самого внутреннего цикла, если циклы вложены).

**Инструкция break – вторая форма**

****

В этой форме **break** выполняет переход на следующую инструкцию за помеченной.

**Инструкция continue**

Как и **break**, эта инструкция существует в двух формах:

* continue;
* continue *идентификаторМетки*;

Используется **только в циклах** для запуска следующей итерации. Если используется с меткой, то запускается следующая итерация помеченного цикла.

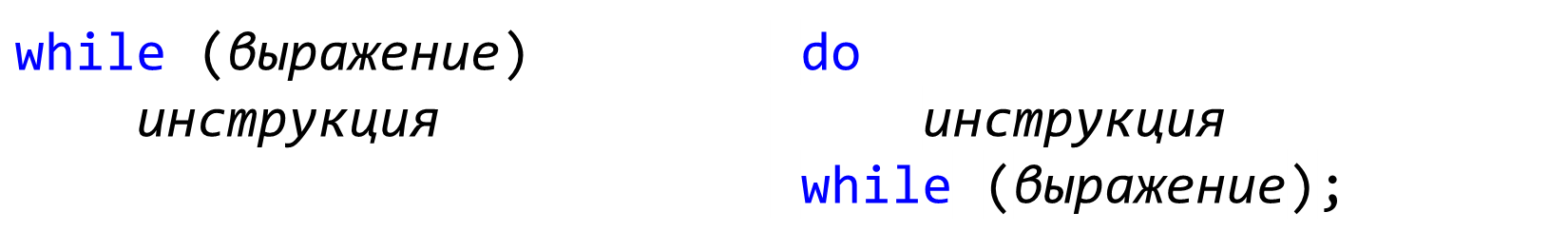
**Инструкция return**

Инструкция **return** осуществляет немедленный выход из функции, возвращая указанное значение или **undefined**:

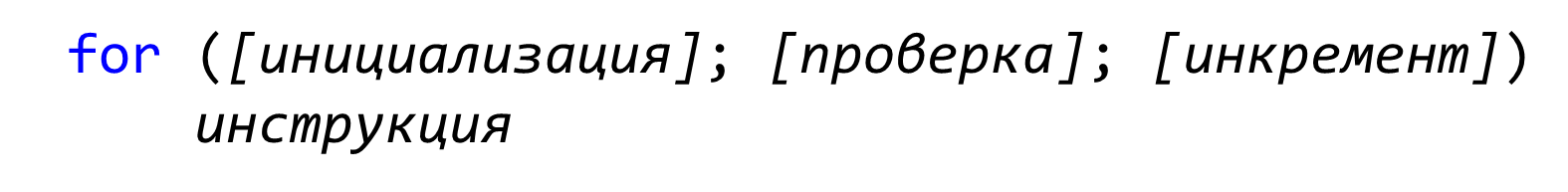
* return;
* return *выражение*;

Инструкция **return** используется только в функциях.

# 30.Циклы while и do-while. Цикл for.



Циклы работают, пока выражение, приведённое к **boolean**, равно **true**.Обратите внимание на обязательную **;** в цикле **do**-**while**.



*Инициализация* выполняется один раз. Обычно задаётся счетчик цикла. Можно использовать **var**, **let**, **const**.

Выражение *проверки* вычисляется перед каждой итерацией цикла. Цикла работает, пока получаем **true**.

Выражение *инкремента* вычисляется в конце цикла. Обычно это изменение счётчика цикла.

**Цикл for / in**

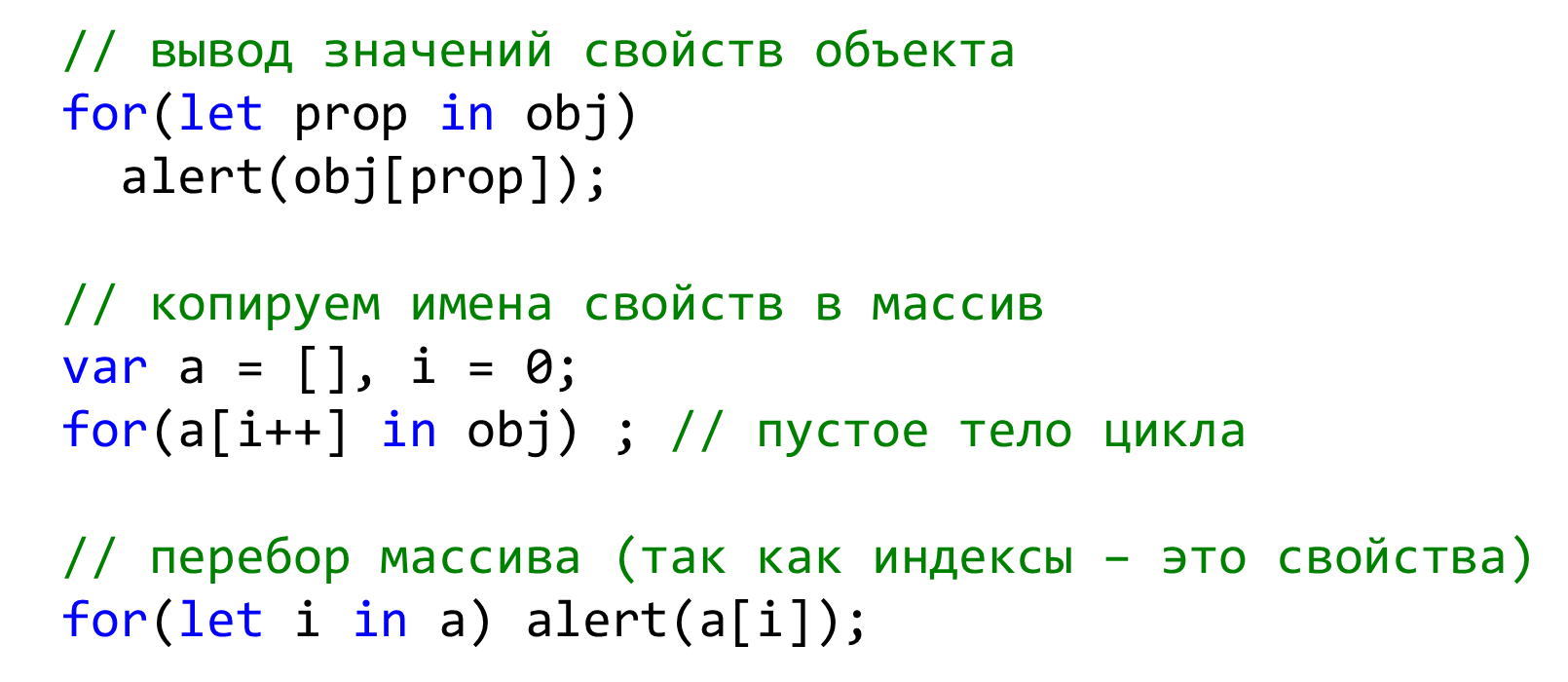


Этот цикл выполнят перебор имён свойств объекта (а имя свойства – это всегда строка):

*объект* – выражение, которое преобразуется в объект.

*переменная* – инструкция var или левостороннее выражение.

*Примеры:*

**

Цикл **for**/**in** перебирает только перечислимые свойства объекта (как сделать у объекта свойство перечислимым или не перечислимым – это рассмотрим позже).

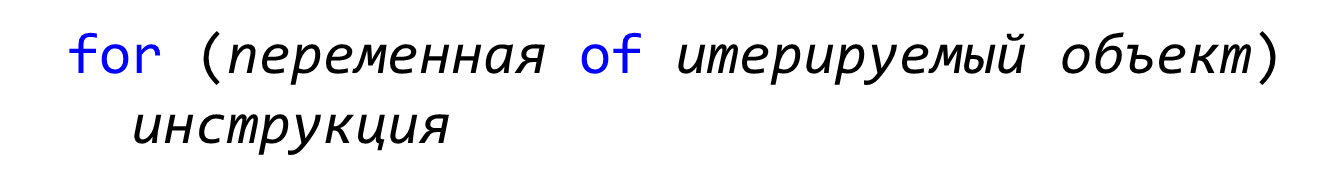
Перебираются свойства, которые есть и в объекте, и во всех его прототипах.

Обычно порядок перебора соответствует порядку определения свойств (свойства прототипов – в конце), но спецификация не даёт указаний на этот счёт.

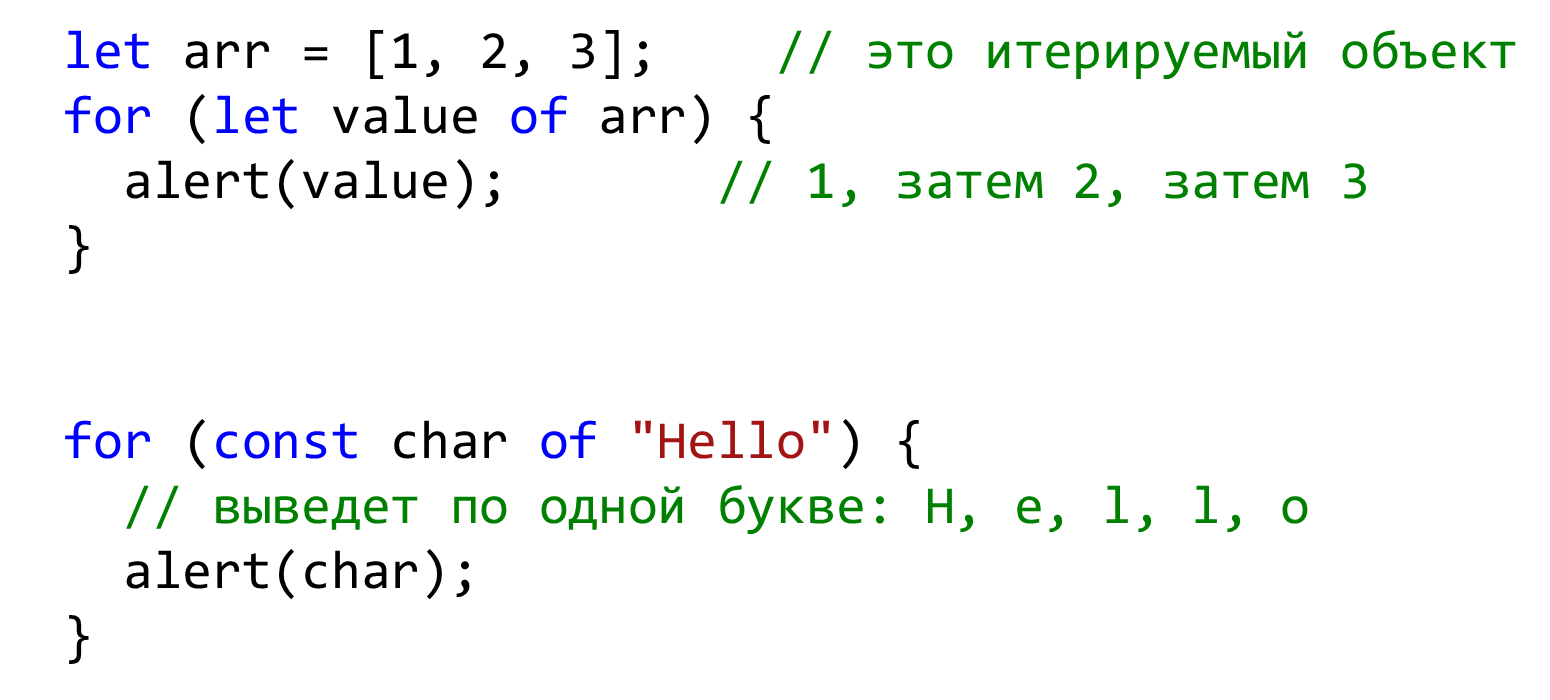
**Цикл for / of и итерируемые объекты**

***Итерируемые*** (или «**перебираемые**») объекты – такие объекты, содержимое которых можно перебрать в цикле. Примеры итерируемых объектов: массив, строка.

Для перебора итерируемых объектов в **ECMAScript** **2015** используется новый цикл **for**/**of**:



*Пример:*

**

# 31.Отступление – метки инструкций

В скрипте любая инструкция может быть помечена:

* *идентификаторМетки*: *инструкция*

1. Метки используются для переходов при помощи инструкций **break** или **continue**.
2. Переход на метки работает только внутри тех инструкций, к которым метки применяются.

**Вывод**: метки применяют для switch, while, do, for.

# 32.Генерация исключений. Обработка исключений.

Генерация исключения выполняется инструкцией **throw**:



Здесь *выражение* может иметь **любой тип результата**. Существует стандартный конструктор **Error**(), которому передаётся строка с описанием ошибки (она сохраняется в объекте исключения в свойстве message).

**Случай 1**: **throw** используется внутри функции.

Выполнение функции прекращается, управление передаётся на ближайший **catch** в стеке вызова.

Если catch отсутствует, прекращается выполнения всего скрипта (но не других скриптов на странице).

**Случай 2**: **throw** используется вне функции («глобально»).

Если есть обрамляющий блок обработки исключений, управление передаётся на его **catch**.

Если обрамляющий блок обработки отсутствует, прекращается выполнения всего скрипта (но не других скриптов на странице).

**Обработка исключений**

Все фигурные и круглые скобки обязательны.

Присутствует или блок **catch**, или блок **finally**, или оба.

Переменная **e** (имя может быть любым) видима только в **catch**-блоке и имеет значение, которое указывали после **throw** при генерации исключения.

В блоке catch можно обработать исключение или (и) сгенерировать исключение повторно.

Блок **finally** выполняется, если блок try завершился:

* 1. как обычно, достигнув конца блока;
  2. из-за инструкции **break**, **continue** или **return**;
  3. с исключением, обработанным в блоке **catch**;
  4. с не перехваченным исключением, которое продолжает своё распространение на более высокие уровни.

# 33.Функции. Функции в JavaScript – особенности. Вложенные функции.

*Функция* – блок кода, который определяется один раз и может выполняться (*вызываться*) многократно.

При определении функции можно задать *параметры*.

При вызове функции на месте параметров указываются *аргументы*.

Результатом вызова функции является *возвращаемое значение*.

В **JavaScript** любая функция:

* получает при вызове дополнительный аргумент – *контекст вызова* (внутри функции доступен через **this**);
* *всегда* возвращает некое значение;
* может выступать в качестве *подпрограммы*, *метода* объекта, *конструктора* объекта;
* сама является объектом.

**Инструкция function**

Первый способ задания функции – *инструкция* **function**:

* function *имя*(*[параметр1* *[*..., *параметрN]]*){
* *[инструкции]*
* }

Имя, круглые скобки, фигурные скобки – обязательны. В круглых скобках находится *список параметров* через запятую (возможно пустой). В фигурных скобках – *тело функции* (возможно пустое).

**Вложенные функции**

Инструкции **function** могут быть вложены в другие определения функций (должен быть первый уровень вложенности – не циклы, не условные инструкции):

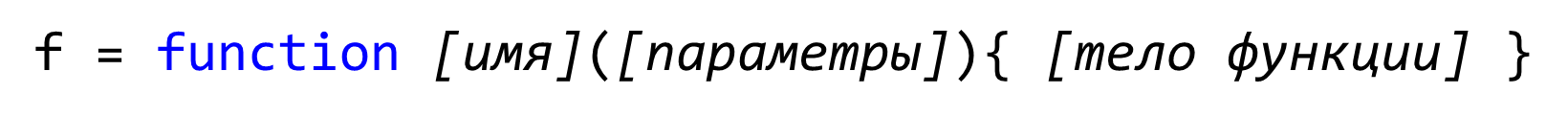
* function hypotenuse(a, b) {
* // square() – вложенная функция
* function square(x) { return x \* x; }
* return (square(a) + square(b)) \*\* 0.5;
* }

**В JavaScript нет перегрузки функций!** Если в **JavaScript** объявлять функции с одним именем, последнее объявление перекрывает все предыдущие:

* function f() { alert("first"); }
* function f() { alert("second"); }
* function f(x) { alert("third"); }
* f(); // выводит "third"

# 34.Выражение function. Конструктор Function

*Выражение* **function** создаёт и возвращает функцию:

****

Чтобы вызвать функцию, значение выражения нужно запомнить в переменной (левостороннем выражении).

Синтаксис выражения совпадает с синтаксисом инструкции. Но имя функции – опционально и может использоваться только внутри выражения. По этой причине часто используется термин *анонимная функция*.

*Пример:*

* let square = function(x) { return x \* x; }
* const f = function factorial(n) {
* if (n <= 1)
* return 1;
* else
* return n \* factorial(n-1);
* };

**Конструктор Function (экзотика ☺)**

Конструктор Function возвращает объект-функцию. Ему передаются строки: имена параметров, тело функции:

* sum = new Function("x", "y", "return x + y;");
* mul = new Function("x, y", "return x \* y;");
* alert(sum(3, 4));
* alert(mul(4, 3));

Это медленный способ; ошибка ведёт к **SyntaxError**. Этот способ перекликается с работой функции **eval**()

# 35.Стрелочные функции. Аргументы и параметры функций.

В **ES2015** появились выражения *стрелочных функций* –короткий аналог выражений function.

Общий синтаксис (нестрогая форма):

* (*список параметров*) => { *тело функции* }
* (*список параметров*) => *выражение*

Второй случай – значение **выражения** возвращается функцией.

*Пример:*

* // inc = function(x) { return x + 1; };
* inc = x => x + 1; // один параметр – скобки не нужны
* sum = (a, b) => a + b;
* getTime = () => {
* let date = new Date();
* let hours = date.getHours();
* let minutes = date.getMinutes();
* return hours + ":" + minutes;
* };

Особенности:

* Стрелочные функции не имеют своего **this**. Внутри функций-стрелок – тот же **this**, что и снаружи.
* Стрелочные функции нельзя использовать в качестве конструктора, то есть нельзя вызывать через new.
* Стрелочные функции не имеют своего arguments. В качестве arguments используются аргументы внешней «обычной» функции.

**Параметры функций**

В простейшем варианте – указываем список имён параметров через запятую (без типа).

Следствие слабой типизации: часто необходим ручной контроль и (или) приведение типов аргументов.

* function foo(str) {
* if (typeof str === "string") return "foo " + str;
* else throw new TypeError("We need string!");
* }

**Аргументы и параметры**

Если аргументов при вызове меньше, чем параметров, незаданные параметры будут равны **undefined**:

* function f(x, y) {
* alert(x);
* alert(y);
* }
* f(1); // выведет "1" и "undefined"

Для незаданных параметров обычно предусматривают разумные значения по умолчанию в теле функции:

* function f(x, y) {
* if (y === undefined) y = -1;
* x = x || -1; // проверка в стиле JS :)
* alert(x);
* alert(y); }

# 36.Параметр-объект. Объект Arguments.

**Параметр-объект**

В **JavaScript** распространён следующий приём: нужные аргументы передаются функции через свойства объекта.

Отчасти это решает проблему с именованными и необязательными параметрами.

*Пример:*

* function printRange(p) {
* p = p || {};
* let lo = p.lo || 1;
* let hi = p.hi || 1;
* let delta = p.delta || 1;
* for (let i = lo; i <= hi; i += delta)
* alert(i);
* }
* printRange({ lo: 1, hi: 10, delta: 2 });

**Объект Arguments**

Объект, созданный конструктором Arguments, используется для хранения **всех** аргументов, переданных функции при вызове.

Этот объект *подобен массиву* (не массив, но имеет схожие свойства).

В теле любой функции этот объект доступен через идентификатор arguments (обращение к arguments вне функции генерирует ошибку).

*Пример:*

* function max() {
* let m = Number.NEGATIVE\_INFINITY;
* for(let i = 0; i < **arguments**.length; i++) {
* if (**arguments**[i] > m) m = **arguments**[i];
* }
* return m;
* }
* let largest = max(1, 10, 10000, 4, 5, 6); // => 10000

# 37.Значение параметра по умолчанию. Возврат значения.

Начиная с **ES2015**, при описании функции для параметра можно указать *значение по умолчанию*. Это значение используется при отсутствующем аргументе, или если аргумент равен **undefined**.

Значение по умолчанию может быть выражением. Это выражение будет вычислено, только если необходимо, то есть когда функция вызвана без аргумента.

Функция **может** содержать инструкцию return для выхода и возврата некоего значения.

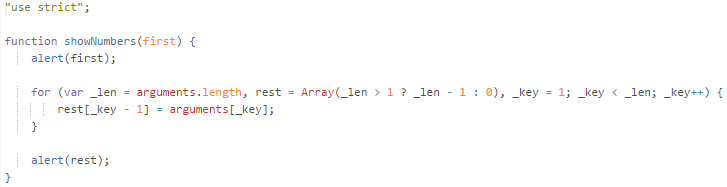
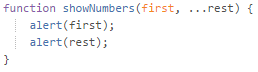
При этом **return**; эквивалентно **return** **undefined**;

Отсутствие **return** в теле функции можно воспринимать как **return** **undefined**; в конце функции.

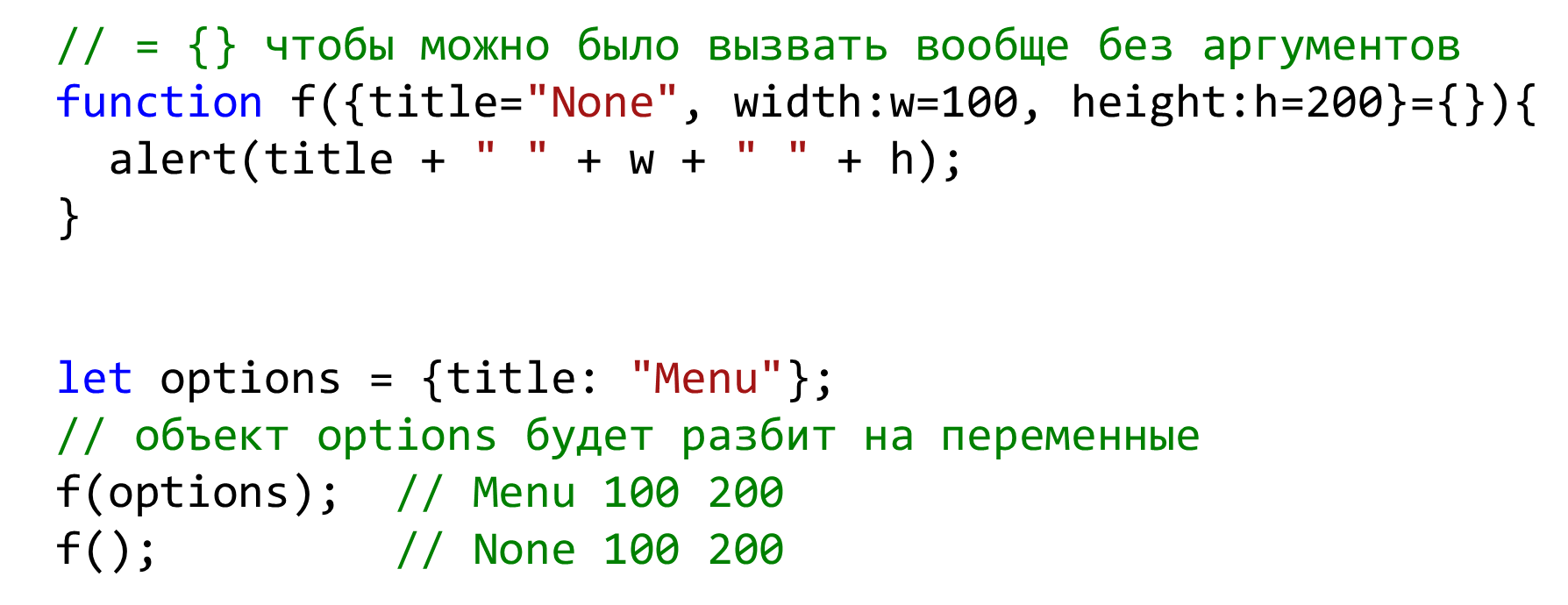
# 38.ES2015 – массив аргументов

Массив аргументов (и это настоящий **Array**) можно получить при помощи синтаксической конструкции ... (нужно использовать в конце списка параметров):

* function showNumbers(first, ...rest) {
* alert(first);
* alert(rest); // rest – это Array
* }
* showNumbers(1, 2, 3, 4);



**Деструктуризация в параметрах**

* ****

# 39.Вызов функции как функции. Вызов функции как метода.

В JS функции могут вызываться четырьмя способами:

* • как *функции* ☺
* • как *методы*
* • как *конструкторы*
* • *косвенно*, с помощью методов **call**() и **apply**()

**Важно**: способ вызова влияет на контекст вызова this.

**Вызов функции как функции**

Такой вызов выполняется в виде *выражения вызова*:

* let h = hypotenuse(3, 4);

Контекст вызова **this** в нестрогом режиме – это глобальный объект, а в строгом режиме – **undefined**:

* "use strict";
* function checkStrictMode() { return !this; }
* alert(checkStrictMode());

**Вызов функции как метода**

*Метод* – функция, которая хранится в свойстве объекта.

**Вызов метода** **=** **выражение обращения к свойству** + **выражение вызова**.

Контекстом вызова является объект, у которого выполняется обращение к свойству (вызывается метод).

*Пример:*

* // определяем объект
* let matrix = {
* size: 0,
* setSize: function(s) { this.size = s; },
* printSize: function () { alert(this.size); }
* };
* // вызываем методы этого объекта
* matrix.setSize(20);
* matrix.printSize();

# 40.Контекст вызова this

Контекст вызова определяется именно **способом вызова** функции (а не местом вызова, как можно подумать)!

*Это важно, если функция, которая вызывается как метод, вызывает свою вложенную функцию как функцию.*

* let obj = {
* method: function () {
* function checkThis() {
* alert(this === obj);
* };
* checkThis();
* }
* };
* **// выводит false**
* obj.method();
* let obj = {
* method: function () {
* **let self = this;**
* function checkThis() {
* alert(**self** === obj);
* };
* checkThis();
* }
* };
* **// выводит true**
* obj.method();

# 41.Вызов функции как конструктора. Косвенный вызов

Это вызов функции (как функции или как метода), который предварён ключевым словом new. Алгоритм:

1. Создаётся новый объект и назначается в качестве **this**.
2. Отрабатывает функция.
3. Возвращаемым значением функции **всегда** будет **объект** – либо объект после return, либо созданный на шаге 1 (если return отсутствует или возвращает примитивное значение).

Если функция вызывается как конструктор и не имеет аргументов, можно не указывать круглые скобки:

* // (почти) эквивалентные строки
* let o = new Object();
* let o = new Object;

У **new** без скобок более низкий приоритет, чем у **new** со скобками (и ниже, чем у выражения доступа к свойствам).

**Косвенный вызов**

Любая функция – это особый объект. У этого объекта есть методы **call**() и **apply**().

Оба метода принимают в качестве первого аргумента контекст вызова функции. Далее идут аргументы функции: у **call**() через запятую, у **apply**() – в виде массива (или в виде *объекта, подобного массиву*).

*Пример:*

* function delta(dx) {
* return this.x + dx;
* }
* let obj = { x: 10 };
* // this для delta будет равен obj
* let dc = **delta.call(obj, 1);**
* let da = **delta.apply(obj, [2]);**
* alert(dc); // "11"
* alert(da); // "12"

Первый аргумент **call**() и **apply**():

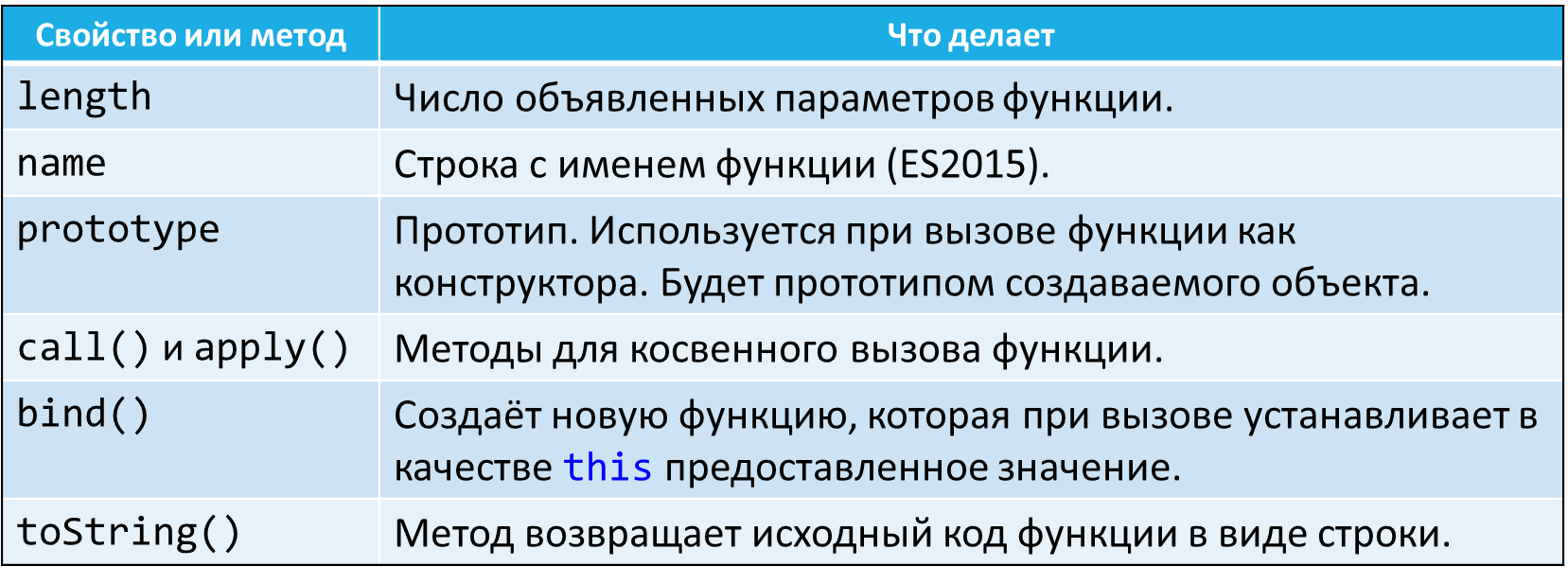
* В строгом режиме всегда становится контекстом вызова «как есть».
* В нестрогом режиме: если это **null** или **undefined** – контекстом будет глобальный объект, если это примитивное значение – контекстом будет объект-обёртка.

# 42.Функции как объекты. Методы объекта. Функция Object(). Метод bind()

**Функции как объекты**

Ранее упоминалось, что функции в JavaScript являются полноценными объектами.Заметим, что оператор **typeof** возвращает для функций не "**object**", а "**function**".

У объекта-функции есть свойства и методы:



Так как функции – это объекты, их можно хранить и обрабатывать как объекты:

* function f(x) {
* return "Alex";
* }
* let arr = [f]; // массив, в котором "сидит" функция
* arr[0](1);

В **JS** к объекту можно в любой момент присоединить и инициализировать новое свойство. Функции не исключение:

* function f(x) {
* return "Alex";
* }
* f.newProp = "XYZ";
* alert(f.newProp);

Функции могут быть аргументами других функций:

* function f(x, y, action) {
* alert(action(x, y));
* }
* function sum(x, y) {
* return x + y;
* }
* f(10, 15, sum);
* f(10, 15, function (x, y) { return x \* y; });

Возможна ситуация, при которой объект-функция возвращается другой функцией:

* function func() {
* function res(x) { return x; }
* return res;
* }
* let f = func();
* alert(f(10));

**bind() – закрепление контекста вызова**

Вызов у функции метода **bind**() возвращает новую функцию, у которой контекст вызова this жёстко установлен в указанное значение:

* function f(y) { return this.x + y; }
* let obj = { x: 10 };
* let g = f.bind(obj); // вызов g(x) вызовет obj.f(x)
* let result = g(2); // result = 12

**bind() – закрепление аргументов**

При вызове **bind**() можно закрепить не только this, но и значения некоторых аргументов функции:

* function f(y) { return this.x + y; }
* let obj = { x: 10 };
* // теперь вызов g() будет всегда означает obj.f(2)
* let g = f.bind(obj, 2);
* let result = g(); // result = 12
* **Функция Object()**

Вызов **Object**() без аргумента, с аргументом **null** или **undefined**, ведёт к созданию нового пустого объекта.

Для аргумента типов **number**, **string**, **boolean** создаётся соответствующий объект-обёртка.Как и **Array**(), **Object**() может вызываться без **new**.

**«Настоящие» методы объекта**

До **ES2015** «метод объекта» – это альтернативное название для свойства-функции.

В **ES2015** добавлены именно «методы объекта», которые, по сути, являются свойствами-функциями, привязанными к объекту.

Особенности «настоящих» методов объекта:

* + Короткий синтаксис объявления: вместо **prop**:**function**(){…} пишем просто **prop**(){ … }.
  + В методе есть спецсвойство [[**HomeObject**]], ссылающееся на объект, которому метод принадлежит.
  + Методами объекта автоматически станут объявления геттеров **get** **prop**() и сеттеров **set** **prop**() (позже).

*Пример:*

* let name = "Alex";
* let user = {
* name,
* // вместо "sayHi: function() {...}"
* sayHi() { alert(this.name); }
* };
* user.sayHi(); // Alex

# 43.Замыкание. Немедленно вызываемые функции

Если функция F() возвращает свою вложенную функцию g(), то все переменные из **scope** F() (и их значения) будут доступны в g(). Не только переменные, но и аргументы функции.

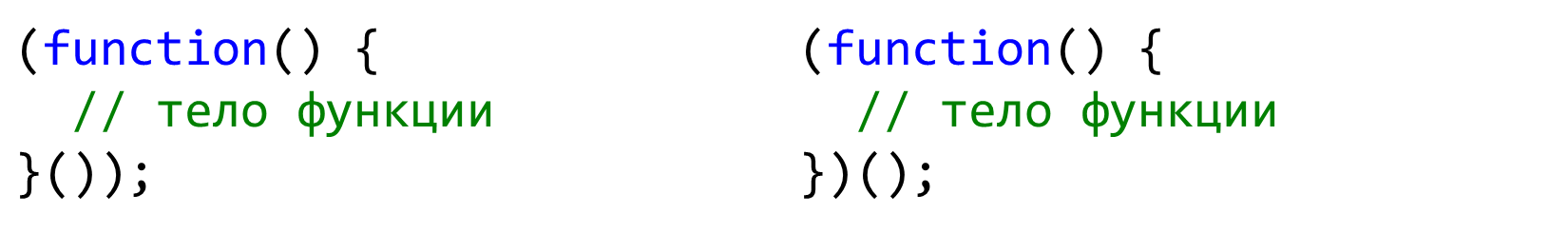
Это явление называется *замыканием* (**closure**).

*Примеры:*

* function func(param) {
* var clos = param;
* function res(x) { return x + clos; }
* return res;
* }
* let f1 = func(100);
* alert(f1(10)); // "110"
* let f2 = func(200);
* alert(f2(10)); // "210"
* function counter() {
* var n = 0;
* return {
* count: function () { return n++; },
* reset: function () { n = 0; }
* };
* }
* var c = counter(), d = counter(); // два счётчика
* alert(c.count()); // => 0
* alert(c.count()); // => 1
* alert(d.count()); // => 0: действуют независимо
* c.reset();
* alert(c.count()); // => 0

**Немедленно вызываемые функции**

В современном **JavaScript** распространён такой приём: объявляется выражение-функция (без имени), и сразу же происходит вызов этой функции:



Внешние скобки нужны, чтобы транслятор понимал это как выражение-функцию, а не как инструкцию.

Немедленно вызываемые функции хороши тем, что дают изолированную область (по переменным), выполняемую один раз.

Это используют, например, в различных сценариях инициализации. Или делают из немедленно вызываемой функции подобие замкнутого модуля.

# 44. Создание объектов. Оператор new.

**Факты об объектах в JavaScript**

1.Любой объект – это значение типа данных **object**.

2. Объект – это неупорядоченная *коллекция свойств*.

3. Каждое свойство имеет *имя* и *значение*.

4. Имена свойств – это *строки* или ***symbol***, значения свойств могут иметь *любой тип* (число, объект, функция).

5. Объект позволяет *читать и записывать* значения своих свойств.

6. По умолчанию объекты являются *динамическими* – позволяют добавлять и удалять свои свойства.

7. Любой объект может быть связан с другим объектом – *прототипом*, и иметь доступ к свойствам прототипа.

8. *Класс* – набор объектов с общим прототипом.

9. Элементы классического ООП моделируются.

10. Объекты используют *ссылочную* семантику при присваивании.

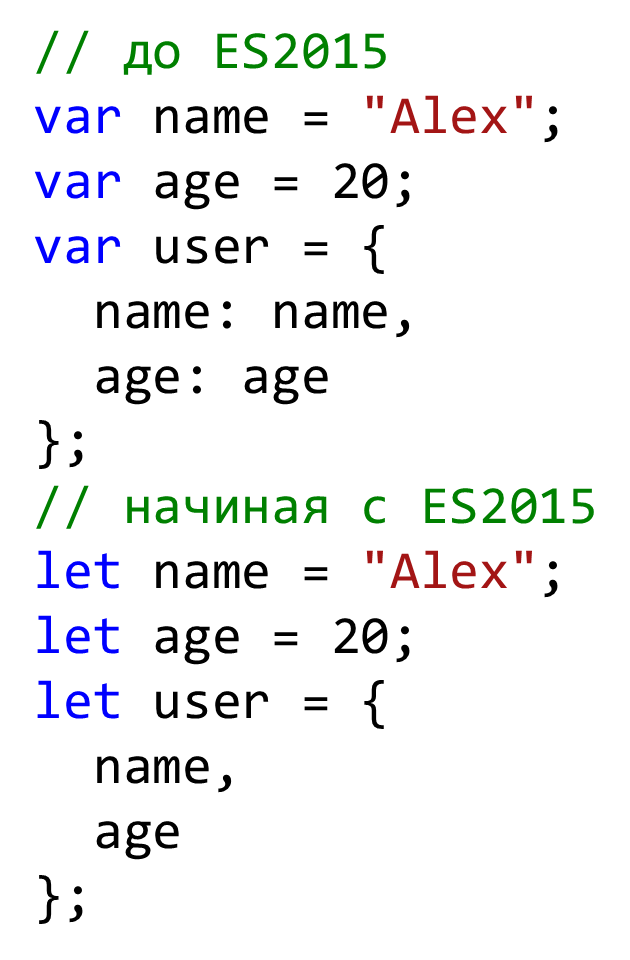
**Создание объектов**

Простейший способ – использовать *литерал объекта*.

«Классический» литерал объекта – это список пар «имя свойства: значение» в фигурных скобках.

*Имя свойства* – идентификатор или строковый литерал, *значение свойства* – любое выражение.

*Примеры литералов объектов:*

* let empty = {}; // это объект без свойств
* let point = {x: 0, y: 0};
* let point3D = {x: point.x, y: point.y + 1, z: 0**,** };
* let book = {
* "main title": "JavaScript",
* "sub-title": "The Definitive Guide",
* **for**: "all audiences"
* };
* **ES2015: «короткие» свойства в литералах**
* 

**ES2015: выражения как имена свойств**

В **литерале объекта** в качестве имени свойства можно использовать выражение (в квадратных скобках):

* let property = "Name";
* let user = { **[property.toLowerCase()]**: "Alex" };
* alert(user[property.toLowerCase()]); // Alex
* alert(user["name"]); // Alex
* alert(user.name); // Alex

**ES2015: symbol как имя свойства**

Значения типа **symbol** (созданные, из реестра, **well**-**known**) могут использоваться как имена свойств объекта:

* let isAdmin = Symbol("isAdmin");
* let user = {
* name: "Alex",
* **[isAdmin]**: true
* };
* alert(user[isAdmin]); // true

**Создание объектов – оператор new**

Для создания и инициализации объекта можно использовать оператор **new**.

Функция после new, называется *конструктором* и служит для инициализации вновь созданного объекта.

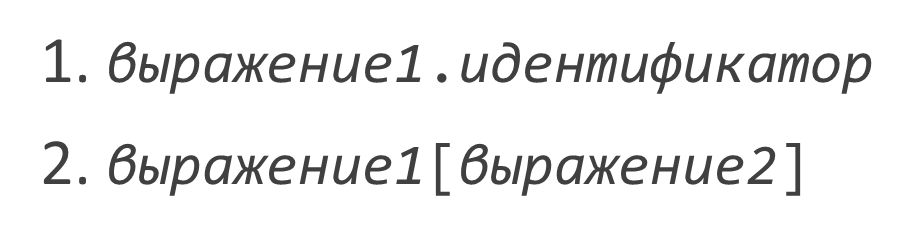
Можно использовать один из встроенных конструкторов или определить собственный конструктор.

*Пример:*

* // создать новый пустой объект: то же, что и {}
* var o = new Object();
* // создать пустой массив: то же, что и []
* var a = new Array();
* // создать объект RegExp
* var r = new RegExp("js");

# 45.Работа со свойствами и прототипы

Выражения обращения к свойству:



Здесь *выражение1* вычисляется и преобразуется в объект, *выражение2* вычисляется и преобразуется в строку (или тип этого выражения должен быть **symbol**).

Несуществующее свойство:

* 1. Если читаем, получаем значение **undefined**.
* 2. Если записываем, то свойство создаётся в объекте.

*Пример работы со свойствами:*

* // объект book определили ранее
* let title = book["main title"];
* book.author = "David Flanagan";
* let x = book.x; // x = undefined
* // TypeError - нельзя привести undefined к object
* let y = book.x.y;
* book.x = null;
* // TypeError - null тоже нельзя привести к object
* y = book.x.y;
* // безопасное получение свойства (или undefined)
* let y = undefined;
* if (book) {
* if (book.x) y = book.x.y;
* }
* // короткая форма, характерная для JavaScript
* let y = book && book.x && book.x.y;

**Работа со свойствами и прототипы**

**Важно**: при чтении свойства просматриваются и свойства объекта, и свойства прототипа (вернее, всех прототипов в цепочке).

При записи решение о существовании и создании свойства принимается только на основании списка свойств самого объекта, без прототипов.

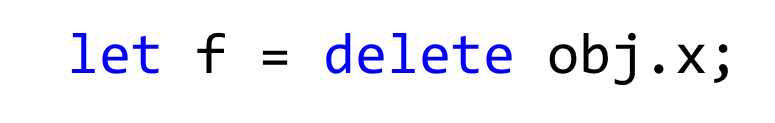
*Пример:*

* let proto = { x: 10 };
* // создаём объект и указываем прототип
* let obj = Object.create(proto);
* alert(obj.x); // выводит "10"
* // создаём собственное свойство у obj
* obj.x = 100;
* alert(obj.x); // выводит "100"
* alert(proto.x); // выводит "10"

# 46.Удаление свойства. Проверка существования свойства.

**Удаление свойства**

Для удаления свойства используется оператор **delete**:



Оператор **delete** может удалить только свойство **самого объекта**, прототип не будет затронут!

Возвращает **false**, только если свойство существует в самом объекте, а не в его прототипах, и не может быть удалено. Во всех остальных случаях возвращает **true**.

**Проверка существования свойства**

Оператор **in**: левый операнд – строка или **symbol**, правый операнд – объект.

Возвращает true**,** если объект имеет собственное или *унаследованное* свойство с указанным именем:

* let o = { x: 1 };
* "x" in o; // true
* "y" in o; // false: o не имеет свойства y
* "toString" in o; // true: o наследует свойство toString

Метод объекта **hasOwnProperty**() проверяет, имеет ли объект **собственное** свойство с указанным именем. Для наследуемых свойств он возвращает **false**:

* let o = { x: 1 };
* o.hasOwnProperty("x"); // true
* o.hasOwnProperty("y"); // false
* o.hasOwnProperty("toString"); // false

# 47.Методы чтения и записи свойств

В **ECMAScript 5** работу с свойством можно транслировать в вызовы **методов** чтения и записи. Самый простой способ сделать это – использовать литерал объекта:

* let obj = {
* // свойство с методами доступа
* // определяется как пара функций
* **get** prop() { /\* тело функции \*/ },
* **set** prop(value) { /\* тело функции \*/ }
* };

Метод чтения объявляется с модификатором **get**, метод записи – с модификатором **set**. Имена методов рассматриваются как имя свойства.

Один из методов можно опустить (например, чтобы получить свойство только для чтения).

Свойство с методами тоже удаляется при помощи **delete**.

*Пример:*

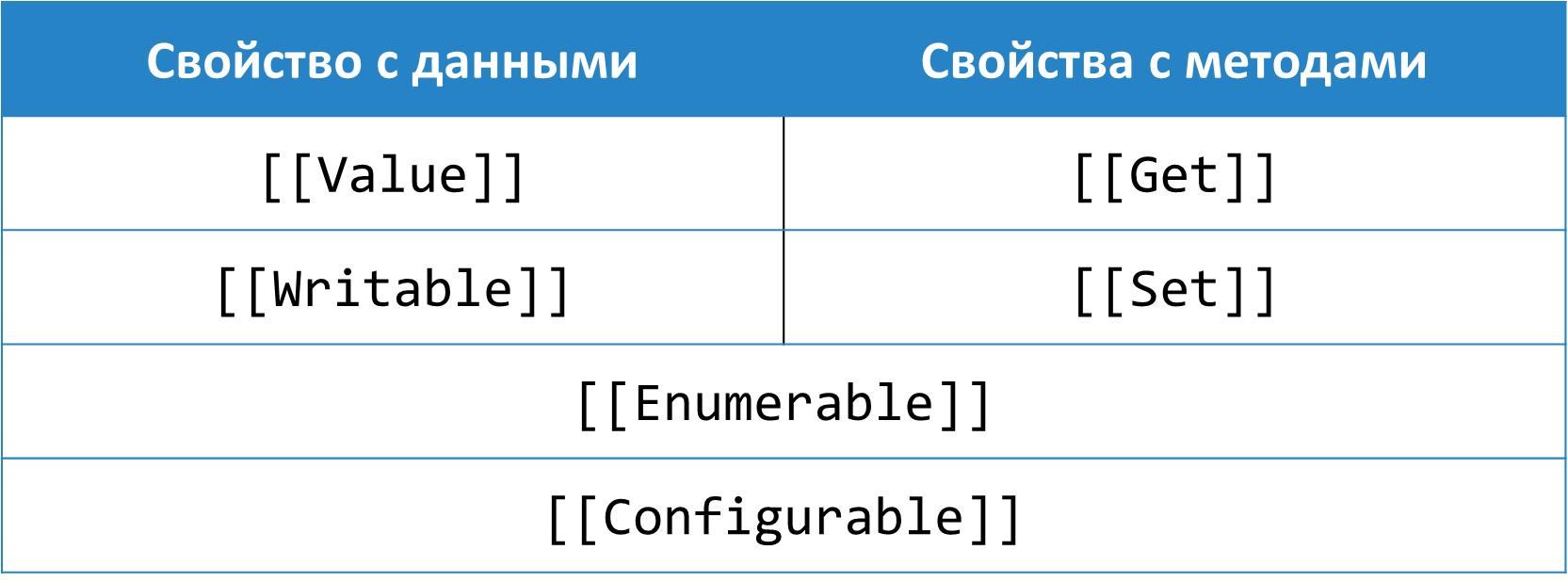
* let person = {
* $age: 0, // притворимся, что это private-поле ☺
* get age() { return this.$age; },
* set age(value) {
* if (value > 0) this.$age = value;
* }
* };
* person.age = 100; // здесь вызов set age(100)
* alert(person.age); // здесь вызов get age()
* person.age = -100;
* alert(person.age); // выводит "100"
* // каждый раз при вычислении выражения "random.octet"
* // будет возвращаться случайное число от 0 до 255
* let random = {
* get octet() { return Math.floor(Math.random() \* 256); }
* };
* alert(random.octet);
* alert(random.octet);
* // а так что будет? ничего! (ничего страшного)
* random.octet = 100;

# 48.Атрибуты свойств (метаданные)

Каждое свойство объекта обладает набором *описательных атрибутов*.

В **ECMAScript** **5** эти атрибуты доступны для чтения и для установки (*тонкая настройка свойства*).

Делается такая настройка при помощи специальных методов и *дескриптора свойства*.



**Атрибуты свойств – расшифровка**

[[**Value**]]: значение свойства.

[[**Writable**]]: значение свойства можно менять (**true/false**).

[[**Enumerable**]]: свойство перечисляется в **for**/**in** (**true/false**).

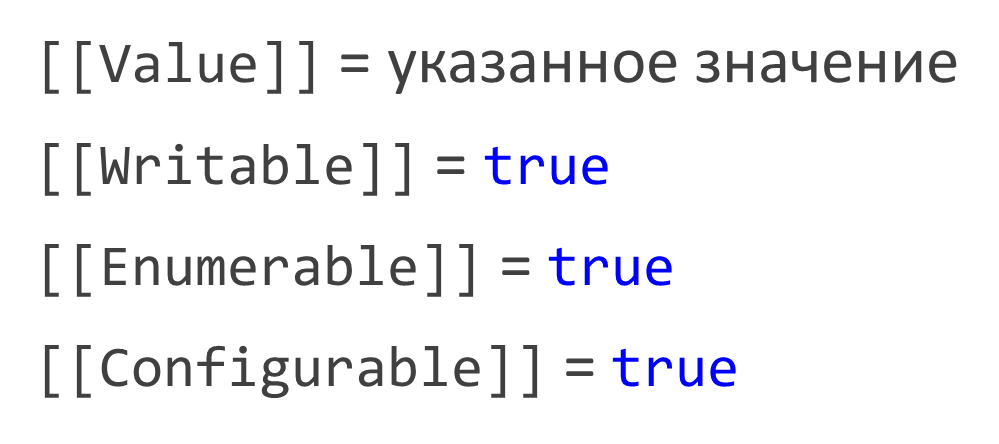
[[**Configurable**]]: атрибуты свойства можно изменить, а свойство можно удалить посредством **delete** (**true/false**).

[[**Get**]]: метод чтения свойства.

[[**Set**]]: метод записи свойства.

**Атрибуты свойств**

Если свойство с данными указывается в литерале, у него:



Для **свойства с методами** вместо [[**Value**]] и [[**Writable**]] будут заданы методы в [[**Get**]] и [[**Set**]].

# 49.Дескриптор свойства. Получение, установка, изменение.

Это объект, обладающий свойствами из набора "**value**", "**writable**", "**enumerable**", "**configurable**"

или из набора "**get**", "**set**", "**enumerable**", "**configurable**".

Дескриптор свойства можно получить у свойства или задать с его помощью атрибуты свойства.

**Дескриптор свойства – получение**

Получить дескриптор свойства можно при помощи метода **Object**.**getOwnPropertyDescriptor()**:

* // d = {value:1, writable:true, enumerable:true, configurable:true}
* let d = Object.getOwnPropertyDescriptor({x:1}, "x");
* // { get: /\*func\*/, set:undefined, enumerable:true, configurable:true}
* d = Object.getOwnPropertyDescriptor(random, "octet");
* // вернёт undefined для унаследованных и несуществующих свойств
* d = Object.getOwnPropertyDescriptor({}, "x"); // d = undefined

**Дескриптор свойства – установка**

Для создания или изменения свойства и атрибутов можно использовать метод **Object**.**defineProperty**().

Аргументы:

* – объект, в котором требуется выполнить изменения,
* – имя создаваемого или изменяемого свойства,
* – дескриптор свойства.

*Пример:*

* let o = {}; // свойств нет
* let meta = {
* value: 1,
* writable: true,
* enumerable: false,
* configurable: true
* };
* // создаём неперечислимое свойство x со значением 1
* Object.defineProperty(o, "x", meta);

**Дескриптор свойства – изменение**

* // сделать x доступным только для чтения
* Object.defineProperty(o, "x", {writable: false});
* // значение x теперь можно изменить только так
* Object.defineProperty(o, "x", {value: 2});
* // сделаем x свойством с методами доступа
* let meta = { get: function() { return 0; } };
* Object.defineProperty(o, "x", meta);

1. Если **создаём** свойство с помощью **defineProperty**() и в дескрипторе чего-то не указываем, то это будет **false** или **undefined**.

2. Если **меняем** свойство с помощью **defineProperty**() и в дескрипторе чего-то не указываем, то это соответствующий атрибут просто не меняется.

3. **defineProperty**() работает только с собственными свойствами объекта.

# 50.Настройка нескольких свойств. Правила согласованности

Если надо создать или изменить несколько свойств, можно использовать **Object.defineProperties()**.

Аргументы:

– объект, в котором требуется выполнить изменения,

– объект, отображающий имена создаваемых или модифицируемых свойств в дескрипторы этих свойств.

* let p = Object.defineProperties({}, {
* x: {value: 1, writable: true, enumerable: true, configurable: true},
* y: {value: 1, writable: true, enumerable: true, configurable: true},
* r: {
* get: function() { return Math.sqrt(this.x\*this.x+this.y\*this.y); },
* enumerable: true,
* configurable: true
* }
* });

Настроить свойства можно и при создании объекта при помощи второго аргумента функции **Object**.**create**():

* let p = Object.create(Object.prototype, {
* x: {value: 1, writable: true, enumerable: true, configurable: true},
* y: {value: 1, writable: true, enumerable: true, configurable: true},
* r: {
* get: function() { return Math.sqrt(this.x\*this.x+this.y\*this.y); },
* enumerable: true,
* configurable: true
* }
* });

Если [[**Configurable**]]=**false** нельзя изменить [[**Configurable**]] и [[**Enumerable**]]

& свойство с данными 1) нельзя превратить в свойство с методами 2) нельзя изменить [[**Writable**]] с false на true, но можно изменить с true на false

& свойство с данными & [[**Writable**]]=false нельзя изменить value

& свойство с методами нельзя изменить методы чтения и записи и нельзя превратить в свойство с данными

Дополнительно: если объект нерасширяемый (как это сделать – чуть позже), то можно изменить существующие собственные свойства этого объекта, но нельзя добавить в него новые свойства.

Вызовы **Object**.**defineProperty**() или **Object**.**defineProperties**() генерируют **TypeError** при нарушении правил согласованности.

# 51.Атрибуты объекта(52. prototype и [[Prototype]], [[Class]], [[Extensible]])

Атрибуты есть не только у отдельного свойства, но и у всего объекта:

*[[****Prototype****]]:* ссылка на прототип объекта,

*[[****Class****]]:* строка с информацией о типе объекта,

*[[****Extensible****]]:* определяет, можно ли добавлять в объект собственные новые свойства (true/false).

*[[****Class****]]* – убрали из спецификации **ES2015**! В новой спецификации просто подробно описывается поведение функции toString() в Object.prototype (как она себя ведёт для встроенных объектов)

***[[Prototype]]*** Устанавливается в момент создания объекта.Метод **Object.getPrototypeOf()** возвращает прототип объекта-аргумента.Если у объекта вызвать метод **isPrototypeOf(),** можно узнать, присутствует ли объект в цепочке прототипов аргумента:

let isProto = p.isPrototypeOf(o);

***[[Class]]***

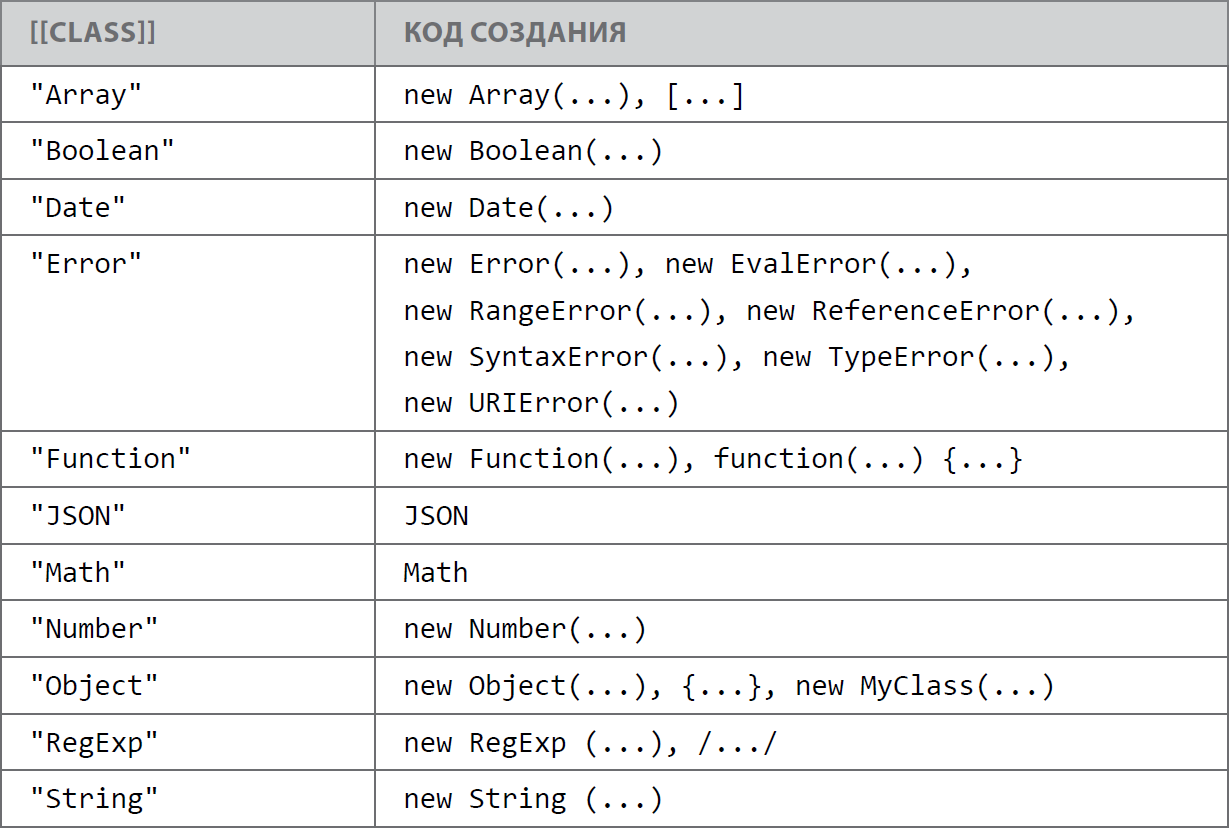
Устанавливается в момент создания объекта. Стандартные конструкторы помещают туда своё имя.

let a = new Array(); // a.[[Class]] = "Array"

Но пользовательские конструкторы так делать не могут!

Прямых методов чтения нет. Но по умолчанию метод объекта **toString()** возвращает строку вида "[object Class]", где Class – это значение атрибута [[Class]].

**Стандартные значения для [[Class]]**

******

**[[Extensible]]**

**ECMAScript** **5** определяет функции для работы с этим атрибутом:

**Object.isExtensible()** – получает объект, а возвращает признак расширяемости (true/false);

**Object.preventExtensions()** – получает объект и делает его нерасширяемым (обратно сделать не получится).

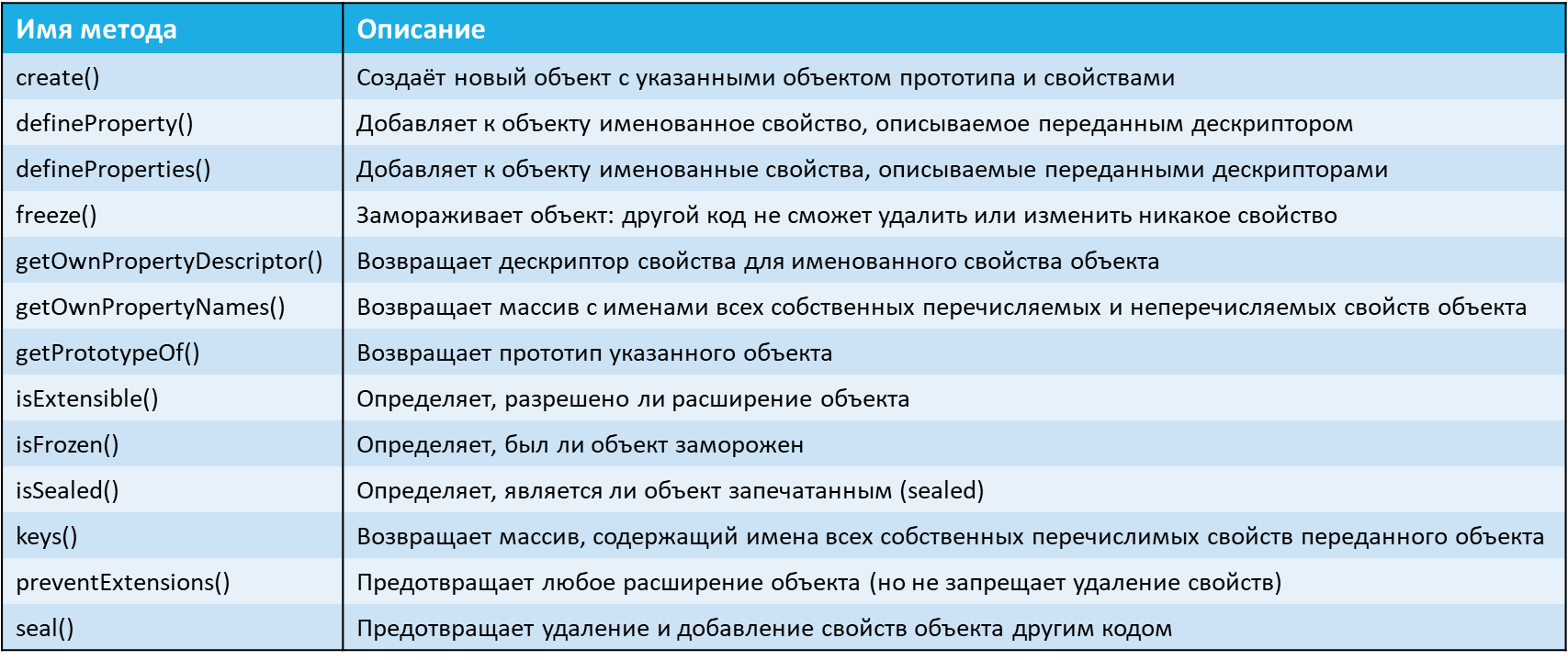
**[[Extensible]] – добавочные методы**

**Object.seal()** – получает объект, делает объект нерасширяемым, и делает все свойства объекта недоступными для настройки.

**Object.freeze()** – это seal() + все собственные свойства с данными делаются доступными только для чтения (это не относится к свойствам с методами).

**Object.isSealed() и Object.isFrozen()** – проверка, что для объекта-аргумента вызывались указанные методы.

**Статические методы Object – сводка**

******

53.Object.values(), Object.entries()

Метод **Object.values()** возвращает массив значений собственных перечисляемых свойств объекта:

* let obj = {foo: "bar", baz: 42};
* alert(Object.values(obj)); // ['bar', 42]
* // аргумент-не объект будет приведён к объекту
* alert(Object.values("foo")); // ['f', 'o', 'o']

**Object.entries()** вернёт массив пар [ключ, значение] для собственных перечисляемых свойств объекта:

* let obj = {foo: "bar", baz: 42};
* alert(Object.entries(obj));
* // [ ['foo', 'bar'], ['baz', 42] ]

# 54.Копирование свойств. Получение символьных свойств

**Object.assign(target, ...sources)**

Этот метод берёт перечисляемые собственные свойства объектов source и копирует их значения в объект target. Возвращается получившийся целевой объект.

* let o1 = { a: 1 };
* let o2 = { b: 2 };
* let o3 = { a: 3 };
* let obj = Object.assign(o1, o2, o3);
* console.log(obj); // { "a": 3, "b": 2}
* console.log(o1); // { "a": 3, "b": 2}

Метод **Object.getOwnPropertySymbols()** возвращает массив всех символьных свойств, найденных непосредственно на переданном объекте.

* let a = Symbol("a");
* let b = Symbol.for("b");
* let obj = { [a]: "Alex", [b]: 40 };
* for(let sym of Object.getOwnPropertySymbols(obj)) {
* alert(sym.toString()); // Symbol(a), Symbol(b)
* }

# 55.Object.getOwnPropertyDescriptors()

**Object.getOwnPropertyDescriptors()**

Эта функция описана в ES2017. Она принимает объект и возвращает объект с дескрипторами всех собственных свойств объекта.

Функция может использоваться для создания клонов объектов (поверхностных):

* let clone = Object.create(
* Object.getPrototypeOf(obj),
* Object.getOwnPropertyDescriptors(obj)
* );

# 56.Проверка одинаковых значений – is()

***var isSame = Object.is(value1, value2);***

Метод проверяет, являются ли два значения одинаковыми:

* оба равны undefined, или оба равны null
* оба равны true, либо оба равны false
* оба являются одинаковыми строками
* оба являются одним и тем же объектом
* оба являются числами и
* оба равны +0 или оба равны -0
* оба равны NaN
* либо оба не равны нулю или NaN и оба имеют одинаковое значение

# 57.Прототипы. Наследование на основе прототипов

В **JavaScript** любой объект имеет атрибут [[**Prototype**]], который хранит ссылку на другой объект или null.

Объект B будем называть *прототипом* объекта A, если выполняется A.[[**Prototype**]]==B.

Если же A.[[**Prototype**]]==null, будем говорить что у *объекта A нет прототипа*.

При **чтении** свойства объекта поиск этого свойства сначала выполняется в самом объекте, а в случае неудачи – в прототипе (и далее по цепочке прототипов).В определённом смысле, **объект наследует свойства своих прототипов**.

**Классом** в JavaScript называется набор объектов, имеющих общего прототипа.

*Напоминание*: поиск свойства в прототипе выполняется только при чтении.

**Установка** свойства **всегда** происходит **в самом объекте** (если нужно, свойство создаётся).

Значит, прототипы не годятся для хранения изменяемых данных многих объектов. Прототипы подходят **для разделения поведения** (т.е. методов) объектов.

**СЛУЧАЙ 1.**Если объект создаётся при помощи литерала объекта, его прототипом будет объект **Object.prototype**:

* let obj = { x: 1 };
* let proto = Object.getPrototypeOf(obj);
* alert(proto == Object.prototype); // "true"

**СЛУЧАЙ 2.**Если объект создаётся при помощи **Object.create(),** его прототип – это первый аргумент данного метода:

* let p = { x: 1 };
* let o1 = Object.create(p);
* alert(Object.getPrototypeOf(o1)==p); // "true"
* let o2 = Object.create(null);
* alert(Object.getPrototypeOf(o2)==null); // "true"

**СЛУЧАЙ 3.**Если объект создаётся оператором **new**, его прототипом будет значение свойства **prototype** конструктора.

* let proto = { x: 1 }; // это будет прототип
* function F() { };
* F.prototype = proto; // установили прототип у функции
* let obj = new F();
* alert(Object.getPrototypeOf(obj) == proto); // "true"

Сейчас было рассмотрено свойство prototype функции.

Любая функция – это объект, а значит у функции есть прототип (и он хранится в атрибуте [[**Prototype**]]).

Важно: прототип функции и значение её свойства prototype – это, вообще говоря, разные объекты.

Совпадения prototype и [[**Prototype**]] возможны.

**СЛУЧАЙ 4.**Свойство объекта \_\_**proto**\_\_ (**pronounced "dunder proto")** позволяет прочитать или изменить прототип объекта:

* let shape = {}, circle = {};
* // установка прототипа объекта
* circle.\_\_proto\_\_ = shape;
* // получение прототипа объекта
* alert(circle.\_\_proto\_\_ === shape); // true

**СЛУЧАЙ 5.**Метод **Object**.**setPrototypeOf**(), который появился в **ES2015**, устанавливает прототип указанного объекта в другой объект или **null**.

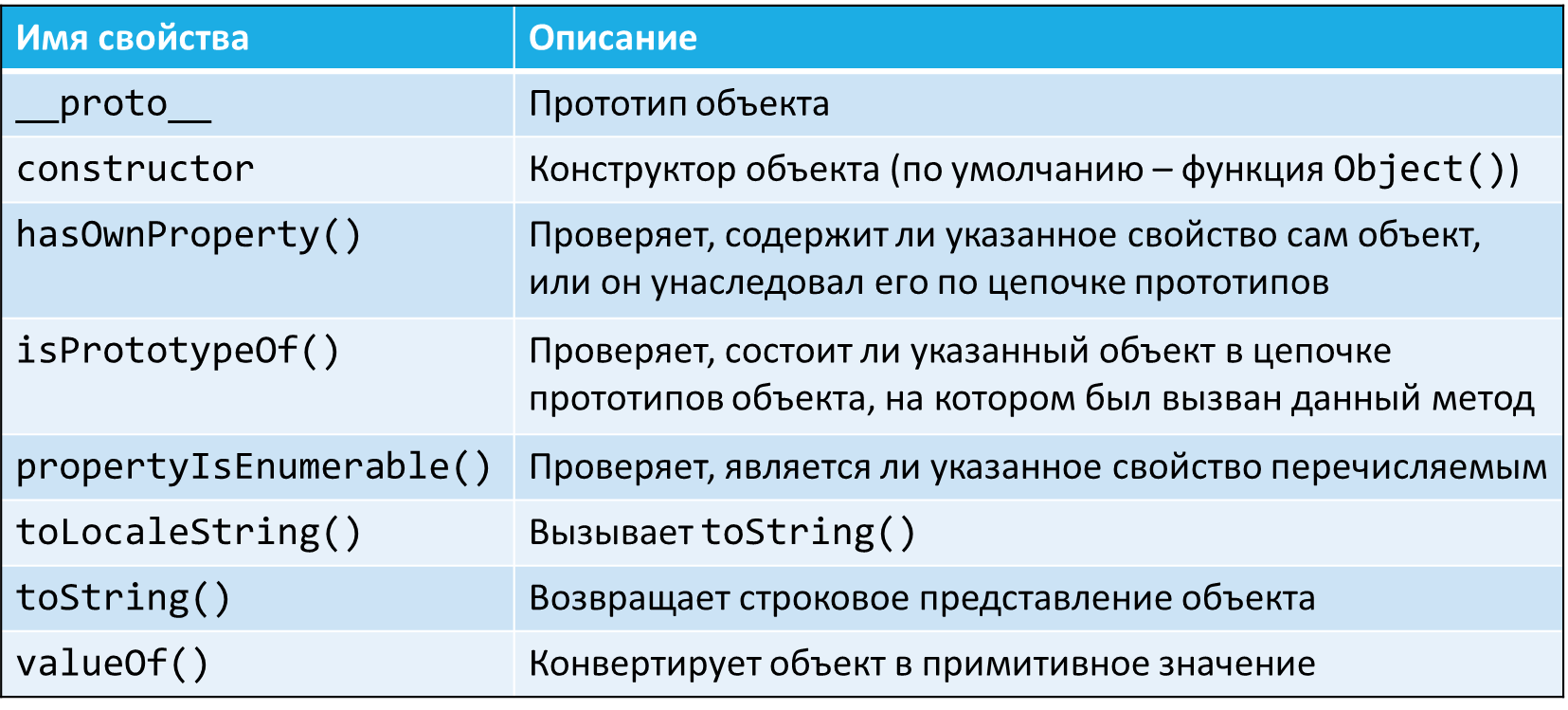
Этот метод не делает ничего, если его аргумент не является объектом или значением **null** (кстати, это справедливо и для \_\_**proto**\_\_).

# 58.Объект Object.prototype.Свойства Object.prototype

**Типовые** сценарии создания объекта ведут к тому, что Object.prototype завершает цепочку его прототипов (вызовы наподобие **Object**.**create**(null) не в счёт!).

Значит, у (обычного) объекта будут все свойства **Object**.**prototype**. А добавление свойства к Object.prototype отразится на всех (обычных) объектах.

**Свойства Object.prototype**

******

**Объект Object.prototype – атрибуты**

* Метаданные объекта Object.prototype:
* 1. [[Prototype]] = null;
* 2. [[Class]] = "Object";
* 3. [[Extensible]] = true;
* 4. У всех его свойств [[Enumerable]] = false.

# 59.Конструкторы.Свойство constructor

Использование оператора **new** и функции-конструктора – это один из способов создания объектов. Такой подход востребован, если необходимо породить набор объектов, обладающих одинаковой структурой.

***Конструкторы – типовой сценарий***

1. В теле функции-конструктора создаются и инициализируются свойства нового объекта.

2. Объект, который хранится в свойстве prototype функции-конструктора, может содержать общие данные (редко) и методы (часто) объектов.

Эти методы могут ссылаться на свойства нового объекта при помощи this – ведь вызываться они будут (скорее всего) в контексте созданного объекта.

|  |
| --- |
| * // при вызове этого как конструктора у нового объекта * // будут созданы и инициализированы свойства x и y * function Point(x, y) { * this.x = x; * this.y = y; * }; * // в прототипе два метода, второй работает со свойствами * // объекта (если вызывается как метод объекта) * Point.prototype = { * getType: function () { return "Point"; }, * sum: function () { return this.x + this.y; } * // создадим один объект * let p1 = new Point(3, 5); * alert(p1.getType()); // => "Point" * alert(p1.sum()); // => "8" * // создадим второй объект * let p2 = new Point(1, 1); * alert(p2.sum()); // => "2" |

По традиции на имя конструктора ссылаются как на имя класса. Например, класс **Array** – это все объекты, которые созданы при помощи конструктора **Array()** и разделяют общий прототип **Array.prototype**.

Имя класса используется оператором instanceof:

* p instanceof Point
* // ищем в цепочке прототипов p объект Point.prototype

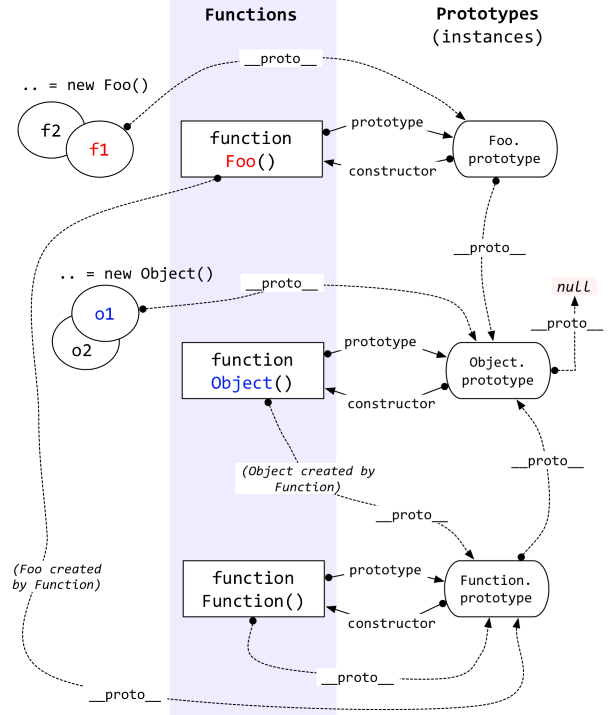
**Свойство constructor**

По умолчанию у любой функции в свойстве **prototype** находится такой объект, у которого **constructor** указывает на функцию:

* let F = function() {}; // функция
* let p = F.prototype; // прототип по умолчанию
* let c = p.constructor;
* c === F // => true: F.prototype.constructor === F

При описании прототипов связь свойства **constructor** с функцией желательно сохранить. Для этого можно не заменять **prototype** (как делали ранее), а дополнять его:

* Point.prototype.getType = function () {
* return "Point";
* };
* Point.prototype.sum = function () {
* return this.x + this.y;
* };



# 60.Закрытые поля в литерале/конструкторе

Моделирование *закрытых полей* (private fields) основано на использовании **замыканий**.

В случае *литерала объекта* закрытое поле объявляется как локальная переменная немедленно вызываемой функции, возвращающей объект. В этом объекте создаются методы для работы с локальной переменной.

* let person = (function () {
* let $age = 0; // закрытое поле
* // возвращаем объект
* return {
* // содержит два метода для работы с закрытым полем
* getAge: function () { return $age; },
* growOlder: function () { $age++; }
* };
* })();

Для имитации закрытых полей в конструкторе используются локальные переменные и аргументы конструктора.

В настраиваемом конструктором объекте задаются методы для работы с закрытыми полями (такие методы называются *привилегированными*).

* function Person(age) {
* // закрытое поле
* let $age = age;
* // два привилегированных метода
* this.getAge = function () { return $age; };
* this.growOlder = function () { $age++; };
* }

Привилегированные методы можно применить для создания *свойства с методами доступа*:

* function Person(age) {
* let $age = age;
* let meta = {
* get: function () { return $age; },
* set: undefined,
* enumerable: true,
* configurable: true };
* Object.defineProperty(this, "age", meta);
* }

# 61.Описание конструктора в модуле

Тело немедленно вызываемой функции (**НВФ**) можно использовать как модуль для описания конструктора и настройки прототипа объекта.

В этом случае **НВФ** возвращает конструктор.

Это даёт дополнительный уровень изоляции, позволяет скрыть вспомогательные функции и переменные.

* let Person = (function () {
* function InnerPerson(name) {
* this.name = name;
* }
* InnerPerson.prototype.greet = function () {
* return "Hello, my name is " + this.name;
* };
* return InnerPerson;
* })();

В случае использования модулей легко становится добавить закрытое поле, разделяемое между всеми объектами одного класса (словно это поле описано в прототипе).

* let Person = (function () {
* let $age = 25; // все объекты разделят это поле
* function InnerPerson(name) { this.name = name; }
* // в прототипе - метод работы с полем
* InnerPerson.prototype.getAge = function() {
* return $age;
* };
* return InnerPerson;
* })();

# 62.Примеси (mixins). Примеси-функция копирования. Примеси-множественное наследование

Если один объект (*получатель*) приобретает свойства другого объекта (*поставщика*) путём прямого копирования свойств, без изменения цепочки прототипов, то новые свойства называют *примесью*.

Для создания примесей обычно используют вспомогательную функцию копирования.

* function mixin(receiver, supplier) {
* for (var property in supplier) {
* if (supplier.hasOwnProperty(property)) {
* receiver[property] = supplier[property];
* }
* }
* return receiver;
* }

Примеси позволяют моделировать (с оговорками) множественное наследование:

* var person = {};
* person = mixin(person, { name: "Alex" });
* person = mixin(person, { age: 100 });
* alert(person.name); // => "Alex"
* alert(person.age); // => "100"

**Наследование**

Пусть имеется класс *Person*:

* function Person(name) {
* this.name = name;
* }
* Person.prototype.greet = function () {
* return "Hello, my name is " + this.name;
* };

Необходимо связать класс *Person* с классом *Developer*:

* function Developer(name, skills) {
* this.name = name;
* this.skills = skills || [];
* }

**ПЕРВЫЙ ВАРИАНТ:**

* Создадим при помощи конструктора произвольный объект Person и присвоим его Developer.prototype.
* При этом желательно корректно установить свойство constructor в объекте Developer.prototype.
* Также Developer.prototype можно дополнить новыми элементами (или переопределить существующие).
* function Developer(name, skills) {
* this.name = name;
* this.skills = skills || [];
* }
* Developer.prototype = new Person();
* Developer.prototype.constructor = Developer;
* Developer.prototype.getSkills = function () {
* return this.skills.join();
* };
* let d = new Developer("Alex", ["C#", "JS"]);
* alert(d.greet()); // "Hello, my name is Alex"
* alert(d.getSkills()); // "C#,JS"
* alert(d instanceof Developer); // "true"
* alert(d instanceof Person); // "true"

**ВТОРОЙ ВАРИАНТ:**

* Можно вместо вызова конструктора Person поместить в Developer.prototype результат вызова Object.create(Person.prototype):
* Developer.prototype = Object.create(Person.prototype);
* Developer.prototype.constructor = Developer;
* До ECMAScript 5 вместо Object.create() применялось решение на основе *промежуточного конструктора*:
* var F = function () { };
* F.prototype = Person.prototype;
* Developer.prototype = new F();
* Developer.prototype.constructor = Developer;

**Наследование:**

* В конструкторе Developer, по сути, дублируется код конструктора Person (установка имени).
* Избежать этого можно, использовав в конструкторе Developer косвенный вызов конструктора Person:
* function Developer(name, skills) {
* Person.apply(this, arguments);
* this.skills = skills || [];
* }
* Косвенные вызовы могут использоваться и для работы с методами прототипа базового класса:
* Developer.prototype.toString = function () {
* let s = Person.prototype.greet.call(this);
* return s + " my skills: " + this.skills.join();
* };

**Контекст вызова конструктора:**

Любой конструктор может быть вызван как обычная функция (без **new**). Различать контекст вызова конструктора поможет следующая проверка:

* function Person(age) {
* if (this instanceof Person) {
* // функция вызвана как конструктор
* } else {
* // функция вызвана как функция (без new)
* }
* }
* Почему работает описанная проверка?
* При вызове **new** **Person**() выполняются следующие шаги:
* 1. Создаётся объект.
* 2. В прототип созданного объекта записывается значение Person.prototype.
* 3. И лишь затем объект передаётся в тело конструктора.
* // создание объекта независимо от способа вызова
* function Person(age) {
* if (this instanceof Person) {
* // код конструктора
* this.age = age;
* }
* else {
* // фабрика объектов
* return new Person(age);
* }
* }

# 63.Особенности конструкции class

В **ES2015** появился новый синтаксис описания классов:

* class *Название* {
* constructor
* *методы*
* }

По сути, конструкция class – «синтаксический сахар» для задания конструктора и методов прототипа.

*Пример* работы с классом:

* class Point {
* constructor(x, y) {
* this.x = x;
* this.y = y;
* }
* toString() { return `(${this.x}, ${this.y})`; }
* }
* let p = new Point(3, 5);
* alert(p.toString());

**ОСОБЕННОСТИ КОНСТРУКЦИИ CLASS**

* Конструктор нельзя вызвать без new, иначе TypeError.
* Объявление класса ведёт себя как let. Оно видно только в текущем блоке и только в коде, который находится ниже объявления.
* Методы внутри class являются методами объекта (см. *методы объекта в литерале объекта*).
* Все методы работают в строгом режиме.
* Все методы **не** перечислимы в цикле for/in.

Классы не могут содержать обычных свойств. Но в классе можно описать *методы* чтения и записи свойств.

Также для имён методов разрешено использовать выражения в квадратных скобках (как в литералах объектов).

* class User {
* constructor(firstName, lastName) {
* this.firstName = firstName;
* this.lastName = lastName;
* }
* get fullName() { return `${this.firstName} ${this.lastName}`; }
* ["test".toUpperCase()]() { alert("PASSED!"); }
* };
* let u = new User("Alex", "Volosevich");
* alert(u.fullName);
* u.TEST();

В **ES2015** статические элементы «новых» классов объявляются с модификатором static.По сути, речь идёт о добавлении свойств к объекту, описывающему функцию-конструктор.

* class Point {
* constructor(x, y) {
* this.x = x;
* this.y = y;
* }
* static createPoint(x) { return new Point(x, 0); }
* static get Zero() { return new Point(0, 0); }
* toString() { return `(${this.x}, ${this.y})`; }
* }
* // работа со статическими элементами
* alert(Point.createPoint(1).toString());
* alert(Point.Zero.toString());

# 64.Наследование классов (ES2015)

Конструкция class позволяет указать **базовый класс** при помощи ключевого слова extends. При этом **наследник** получает доступ к методам базового класса.

В конструкторе и методах наследника доступ к предку можно получить при помощи ключевого слова super.

**Внимание**: в конструкторе наследника **должен** быть вызов конструктора предка через super(), и этот вызов должен предварять обращение к **this**.

После **extends** обычно пишут класс. Но на самом деле там допустимо выражение, возвращающее класс.

* // это будет базовый класс
* class Point {
* constructor(x, y) {
* this.x = x;
* this.y = y;
* }
* toString() { return `(${this.x}, ${this.y})`; }
* }
* // класс-наследник
* class ColorPoint extends Point {
* constructor(x, y, color) {
* super(x, y);
* this.color = color;
* }
* toString() {
* return super.toString() + ' in ' + this.color;
* }
* }

Конструкция super работает не только в классах, но и в литералах объектов (но только для *методов объектов*).

* let animal = {
* walk() { alert("I'm walking"); }
* };
* let rabbit = {
* \_\_proto\_\_: animal, // установка прототипа
* walk() { super.walk(); }
* };
* rabbit.walk();

Как и почему работает super в методах объектов?

Напомним, что метод объект (и только он) имеет внутренний атрибут [[**HomeObject**]] – ссылка на объект, которому метод принадлежит.

Запись **super**.**walk**() эквивалентна [[**HomeObject**]].\_\_**proto\_\_.walk**()

**КЛАСС БЕЗ ЯВНОГО КОНСТРУКТОРА**

В конструкции class можно не задавать constructor. В классах без предка это эквивалентно объявлению

* constructor() {}

В класса-наследниках отсутствие конструктора равносильно такому объявлению:

* constructor(...args) {
* super(...args);
* }

# 65.Итерируемые объекты. Создание итерируемого объекта

*Итерируемые* объекты предоставляют механизм для перебора своих элементов (например, в цикле). Примеры итерируемых объектов: массив, строка.

Для перебора итерируемых объектов начиная с **ECMAScript** **2015** используется новый цикл for/of:

* for (*переменная* of *итерируемый объект*)
* *инструкция*

Примеры работы цикла **for**/**of**:

* let arr = [1, 2, 3]; // это итерируемый объект
* for (let value of arr) {
* alert(value); // 1, затем 2, затем 3
* }
* for (const char of "Hello") {
* // выведет по одной букве: H, e, l, l, o
* alert(char);
* }

**СОЗДАНИЕ ИТЕРИРУЕМОГО ОБЪЕКТА**

Чтобы сделать объект итерируемым, нужно добавить в него свойство-функцию (или метод объекта) с именем **Symbol.iterator** (это well-known symbol).

Функция должна возвращать особый объект-*итератор*. У него должен быть метод next(), который при каждом вызове возвращает объект со свойствами value (очередное значение) и done (окончен ли перебор).

Интегрируемый объект – *Пример:*

* let range = { from: 1, to: 5 }
* range[Symbol.iterator] = function () {
* let current = this.from;
* let last = this.to;
* // метод Symbol.iterator() должен вернуть объект с методом next()
* return {
* next() {
* if (current <= last) return { done: false, value: current++ };
* else return { done: true };
* }
* }
* };
* // первый способ использования
* for (let num of range) {
* alert(num); // печатает 1, затем 2, 3, 4, 5
* }
* // второй способ использования
* let iterator = range[Symbol.iterator]();
* while (true) {
* let step = iterator.next();
* if (step.done) break;
* alert(step.value); // 1, затем 2, 3, 4, 5
* }

*Примечания:*

* 1. Оператор ... работает с итерируемыми объектами (в спецификации сказано «с массивами»).
* 2. Возможны бесконечные итераторы. *Пример* – итератор генерирует случайные числа. Цикл for/of по такому итератору тоже будет бесконечным, его нужно прерывать через break.

# 66.Генераторы. Композиция генераторов

***Генератор*** – особая разновидность функции.

Генератор может приостановить своё выполнение, вернуть промежуточный результат и возобновить выполнение позже, в произвольный момент времени.

Для чего нужны? Упростить реализацию итераторов!

Для объявления генератора используется новая синтаксическая конструкция **function\*** (функция со звёздочкой). Её называют *функция-генератор* (**generator** **function**). Звёздочку ставим или после function, или перед именем функции (хорошо для методов объектов).

Для возврата промежуточного результата в функции-генераторе используется оператор yield (инструкция return тоже может использоваться – в особых случаях).*Пример* простейшего генератора:

* function\* numbers() {
* yield 1;
* yield 2;
* yield 3;
* }

**Запуск генератора:**

* Вызов функции-генератора возвращает итератор:
* var iter = numbers();
* let step = iter.next();
* alert(`${step.value} ${step.done}`); // 1 false
* step = iter.next();
* alert(`${step.value} ${step.done}`); // 2 false
* step = iter.next();
* alert(`${step.value} ${step.done}`); // 3 false
* step = iter.next();
* alert(`${step.value} ${step.done}`); // undefined true
* // или так
* for (let num of numbers()) alert(num);

**КОМПОЗИЦИЯ ГЕНЕРАТОРОВ**

* Если один генератор включает вызовы других генераторов, получаем *композицию генераторов*.
* Выражение вида
* yield\* *вызов-генератора*()
* делегирует получение значений другому генератору.

*Пример:*

* function\* range(start, end) {
* for (let i = start; i <= end; i++) yield i;
* }
* function\* sequence() {
* yield\* range(0, 3);
* yield\* range(7, 9);
* }
* for(let num of sequence()) {
* alert(num); // 0 1 2 3 7 8 9
* }

# 67.Передача значений в генератор. Прекращение работы генератора

* let result = yield value;

Этот вызов делает следующее:

* Возвращает **value** во внешний код, приостанавливая выполнение генератора.
* Внешний код может обработать значение и затем вызвать метод **next()** **с аргументом**: **generator.next(arg).**
* Генератор продолжит выполнение, аргумент метода **next()** будет возвращён как результат **yield** и записан в **result**.

*Пример:*

* function\* gen() {
* let ask1 = yield "2 + 2?";
* alert(ask1); // 4
* let ask2 = yield "3 \* 3?";
* alert(ask2); // 9
* }
* let generator = gen();
* alert( generator.next().value ); // "2 + 2?"
* alert( generator.next(4).value ); // "3 \* 3?"
* alert( generator.next(9).done ); // true

1.Первый **.next()** начинает выполнение. Оно доходит до первого yield.

2.Результат возвращается во внешний код.

3.Второй **.next(4)** передаёт 4 обратно в генератор как результат первого yield и возобновляет выполнение.

4.Выполнение доходит до второго yield, который станет результатом **.next(4).**

5.Третий **.next(9)** передаёт 9 в генератор как результат второго yield и возобновляет выполнение, которое завершается окончанием функции, так что done: true.

**ПРЕКРАЩЕНИЕ РАБОТЫ ГЕНЕРАТОРА**

Пусть **\*gen()** – это генератор, и **let generator = gen()**.

Вызов **generator.return(arg)** заставит генератор остановиться на следующем **yield**, вернув значение **arg**.

Если инструкция return встретилась в самом генераторе, она также прекращает его работу. Значение, возвращённое return, не учитывается в цикле **for/of.**

*Пример:*

* function\* numbers() {
* yield 1;
* yield 2;
* yield 3;
* }
* let g = numbers();
* let step1 = g.next();
* alert(JSON.stringify(step1)); // {value: 1, done: false}
* let step2 = g.return(10);
* alert(JSON.stringify(step2)); // {value: 10, done: true}
* let step3 = g.next();
* alert(JSON.stringify(step3)); // {done: true}

**ПЕРЕДАЧА В ГЕНЕРАТОР ИСКЛЮЧЕНИЯ**

Пусть **\*gen()** – это генератор, и **let generator = gen().**

Вызов **generator.throw(err)** передаст в генератор исключение err. Это исключение можно отловить в генераторе при помощи **try-catch**.

* function\* gen() {
* try {
* let result = yield "2 + 2?";
* alert("Never see, if exception");
* }
* catch(e) {
* alert(e); // выведет ошибку
* }
* }
* let generator = gen();
* let question = generator.next().value;
* generator.throw(new Error("No answer"));

# 68.Коллекция Map. Свойства и методы Map

**Map** – это коллекция для хранения пар «*ключ-значение*». Ключ может быть абсолютно любым. Ключи сохраняются как есть, без преобразования типов (напоминание: в объекте ключами могут быть только строки и **symbol**).

Проверка ключей на равенство выполняется при помощи **Object.is()** (в частности, **NaN** считается равным **NaN**).

***Свойства и методы Map***



*Пример:*

* let map = new Map();
* map.set("1", "str"); // ключ - строка
* map.set(1, "num"); // ключ - число
* map.set(true, "bool"); // ключ - булевое значение
* alert(map.size); // 3 – количество пар
* // в обычном объекте это было бы одно и то же
* alert(map.get(1)); // num
* alert(map.get("1")); // str
* map.delete("1");
* alert(map.has("1") ? "Yes" : "No"); // No

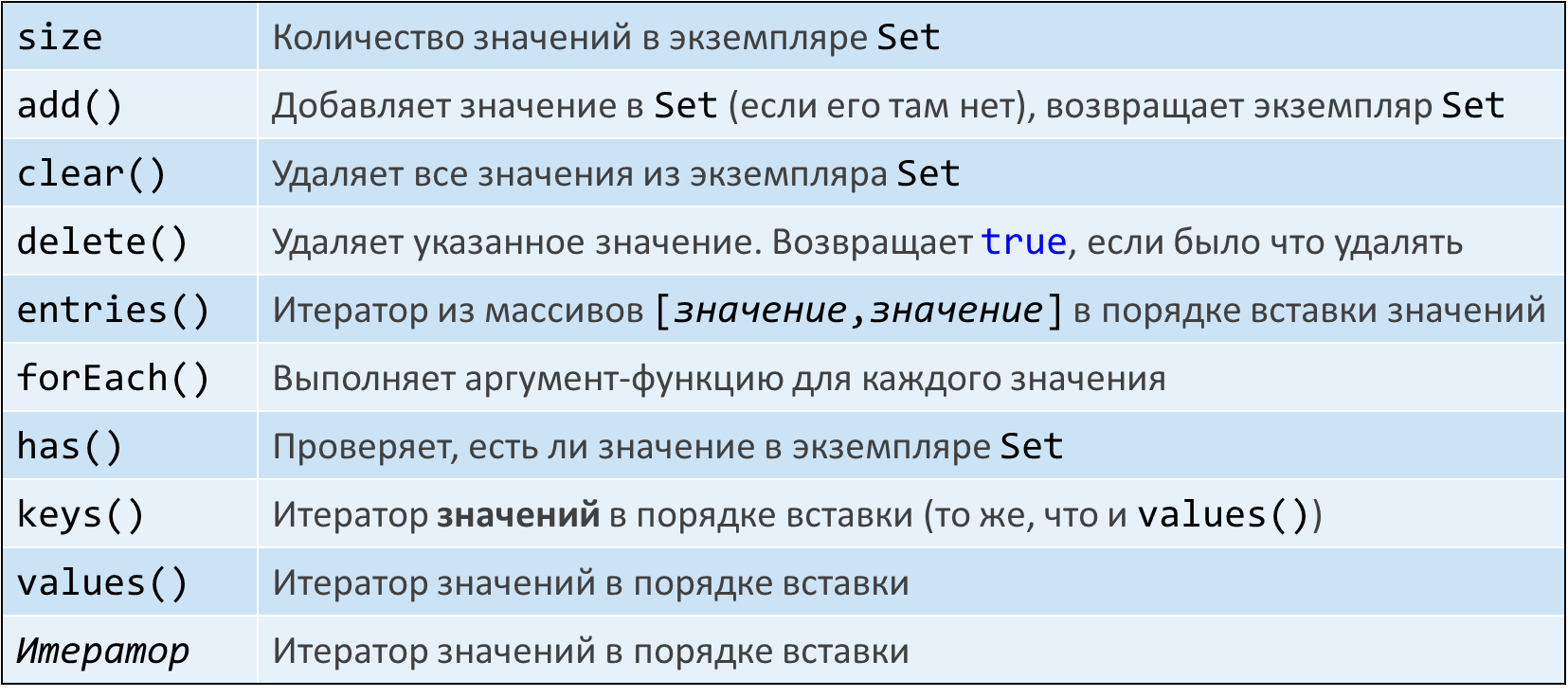
# 69.Коллекция Set. Свойства и методы Set

**Set** – множество неповторяющихся значений.

Коллекция **Set** имеет схожесть с **Map**: ключом может быть абсолютно произвольное значение, которое сохраняется без преобразования типов.

Проверка ключей на равенство в **Set** осуществляется аналогично проверке в **Map**.

**Свойства и методы Set**



*Пример:*

* // конструктору Set можно передать итерируемый объект
* let set = new Set([1, "2", false]);
* set.add(NaN).add(1);
* // 1, 2, false, NaN
* set.forEach((value, key, set) => alert(key));
* set.delete(false);
* // 1, 2, NaN
* for (let value of set.values()) alert(value);

# 70.Коллекции WeakMap и WeakSet. Применение WeakSet/WeakMap

WeakSet – особый Set, не препятствующий сборщику мусора удалять свои элементы. То есть, если некий объект присутствует **только** в WeakSet, он удаляется из памяти.

Аналогично ведёт себя WeakMap для Map (объект – ключ).

**Важно**: ключи в WeakMap и значения в WeakSet – **только объекты** (иначе TypeError).

*Примечание:* У нас есть *элементы* на странице. Мы для элементов хотим хранить *обработчики* событий.

Поместим пару *элемент*-*обработчик* в WeakMap (элемент будет ключом).

Если удалить элемент на странице, сборщик мусора автоматически удалит и обработчик для этого элемента из WeakMap. То есть, нам не надо вручную удалять вспомогательные данные, когда удалён основной объект.

**ПРИМЕНЕНИЕ WEAKSET/WEAKMAP**

У нас есть *элементы* на странице. Мы для элементов хотим хранить *обработчики* событий.

Поместим пару *элемент*-*обработчик* в **WeakMap** (элемент будет ключом).

Если удалить элемент на странице, сборщик мусора автоматически удалит и обработчик для этого элемента из **WeakMap**. То есть, нам не надо вручную удалять вспомогательные данные, когда удалён основной объект.

# 71.Ограничения WeakSet/WeakMap

* + Нет свойства **size**
  + Нельзя перебрать элементы (ключи, значения)
  + Нет метода **forEach**()
  + Нет метода **clear**()

Ограничения связаны с тем, что с **Weak**-коллекциями в **независимом потоке** работает сборщик мусора.

# 72.Буферные массивы. Буферные массивы – создание

Классические массивы **ECMAScript** обладают рядом особенностей: динамическое изменение размера, хранение элементов разных типов.

Эти особенности порождают очевидные проблемы: **низкая скорость работы** и **большое потребление памяти.**

Для решения этих проблем **ECMAScript 2015** предлагает *буферные массивы* и *типизированные массивы*.

***Буферный массив*** – коллекция байт в памяти. Каждый байт рассматривается как элемент буферного массива.

**Размер** буферного массива определяется при его создании и **не может изменяться** динамически.

В момент создания буферного массива все его элементы инициализируются значением 0.

**БУФЕРНЫЕ МАССИВЫ – СОЗДАНИЕ**

Буферный массив создаётся конструктором **ArrayBuffer** с указанием неотрицательного размера в байтах:

* let buffer = new ArrayBuffer(80);

Но напрямую работать с буферным массивом нельзя! Для обслуживания массива используются либо объекты **DataView**, либо типизированные массивы.

# 73.Объект DataView. DataView – методы чтения и записи

Объект **DataView** создаётся на основе существующего буферного массива. Опционально можно указать смещение в буфере и захватываемую длину буфера.

* let dv = new DataView(buffer);

Объект **DataView** предоставляет методы чтения и записи элементов числовых типов:

* // записали -10 со смещением 0 байт
* dv.setInt32(0, -10, false);
* alert(dv.getInt32(0, false));

**DATAVIEW – МЕТОДЫ ЧТЕНИЯ И ЗАПИСИ**

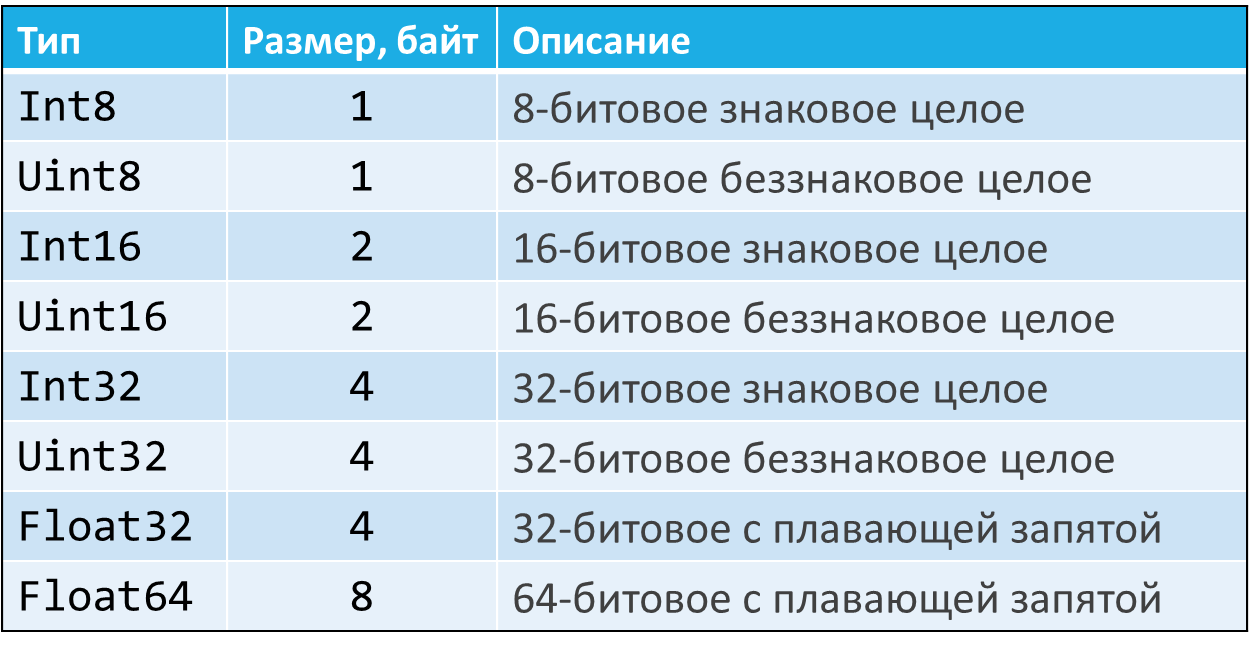
Любой метод записи **DataView** получает три аргумента:

* + смещение в буфере
  + записываемое число
  + (\*) **true** (порядок байт **little-endian**) или **false (big-endian)**

Метод чтения получает два аргумента:

* + смещение в буфере
  + (\*) **true** (порядок байт **little-endian**) или **false (big-endian)**

**Типы, для которых существуют методы чтения и записи**

****

# 74.Типизированные массивы

*Типизированные массивы* позволяют работать с буферными массивами как с обычными массивами.

Для каждого типа из таблицы на предыдущем слайде существует свой массив. К примеру, массив **Int32Array**.

Типизированные массивы поддерживают многие методы обычных массивов **Array**. Для них доступна возможность итерирования.

*Пример работы с массивом* ***Int32Array****:*

* let buffer = new ArrayBuffer(80);
* // разные конструкторы
* let arr1 = new Int32Array(10);
* let arr2 = new Int32Array(arr1);
* let arr3 = new Int32Array(buffer); // + смещение и длина
* arr1[0] = 10;
* arr1[1] = 20;
* // выведет все десять элементов (10, 20, остальные 0)
* for(let x of arr1) alert(x);

Особо упомянем массив **Uint8ClampedArray**. Он весьма похож на **Uint8Array**. Однако эти массивы ведут себя по разному, если элементу присваивается значение за пределами диапазона **0..255**:

**Uint8Array** – используется восемь младших байт значения

**Uint8ClampedArray** – используется **0** (если значение меньше нуля) или **255** (если значение больше, чем **255**)

# 75.Асинхронное выполнение. Функция setTimeout()

***Асинхронный код*** – код, который выполняется в параллельном вычислительном потоке.

Когда такой код стоит использовать:

* + работа с сетью
  + работа с диском
  + фоновые вычисления
  + анимация интерфейса
  + . . .

Глобальный объект в браузерах и в **Node.js** имеет функцию **setTimeout**(), позволяющую вызвать любую функцию через заданный промежуток времени.

Аргументы **setTimeout():**

* + Функция для исполнения **f()**
  + Время отложенного запуска в миллисекундах
  + Аргументы функции **f()** (не поддерживается в **IE<9**)

*Пример:*

* function func(str) {
* alert(str);
* }
* alert("Start");
* setTimeout(func, 5000, "Hello");
* **setTimeout(func, 0, "Read about Run-to-completion semantics!");**
* alert("Stop");
* // setTimeout() возвращает идентификатор таймера
* let id = setTimeout(func, 2000, "Cancel");
* clearTimeout(id); // a clearTimeout() останавливает таймер
* alert("Clear");

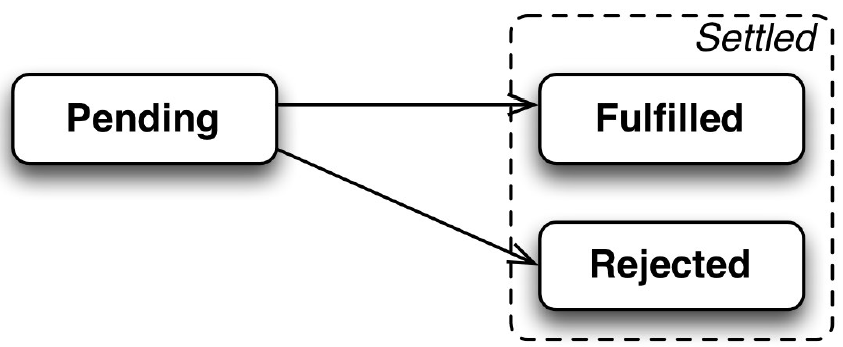
# 76.Promise. Состояние Promise. Схема использования

***Promise*** («промис») – объект, используемый для выполнения отложенных и асинхронных операций.

Задача Promise – облегчить **организацию** асинхронных операций.

Promise впервые были представлены в ES2015 – они сделаны по спецификации Promises/A+.

Promise хранит своё состояние. Вначале это **pending** (*ожидание*), затем – **fulfilled** (*выполнено успешно*) или **rejected** (выполнено с ошибкой).



Перевод **Promise** в состояние **fulfilled** или **rejected** выполняется инкапсулированной в Promise функцией (как правило, это некая асинхронная операция).

Для **Promise** можно задать функции-обработчики:

* **onFulfilled**: вызывается при переходе в состояние **fulfilled**
* **onRejected**: вызывается при переходе в **rejected**

**PROMISE – СХЕМА ИСПОЛЬЗОВАНИЯ**

1. Код, которому надо сделать что-то асинхронно, создаёт **Promise** и возвращает его.
2. Внешний код, получив **Promise**, навешивает на него обработчики.
3. По завершении процесса асинхронный код из пункта 1 переводит Promise в состояние **fulfilled** (с результатом) или **rejected** (с ошибкой). При этом во внешнем коде автоматически вызываются соответствующие обработчики.

Синтаксис создания:

* let promise = new Promise(function(resolve, reject) {
* // Эта функция будет вызвана автоматически.
* // В ней можно делать любые асинхронные операции,
* // а когда они завершатся, нужно вызвать:
* // resolve(результат) при успешном выполнении
* // reject(ошибка) при ошибке
* })

Универсальный метод для навешивания обработчиков:

* promise.then(onFulfilled, onRejected)

*Примеры:*

1.Создадим **Promise** с асинхронной операцией, которая через 1 секунду переведёт **Promise** в состояние **fulfilled** с результатом "**result**":

* let pr = new Promise((resolve, reject) => {
* setTimeout(() => { resolve("result") }, 1000);
* });

2.Установим для созданного **Promise** обработчики (в нашем случае, очевидно, сработает только первый):

* pr.then(
* result => {
* // result – это то, что мы передали в resolve()
* alert("Fulfilled: " + result);
* },
* error => {
* // error – это аргумент reject()
* alert("Rejected: " + error);
* }
* );

**НЮАНСЫ**

1. С помощью **then()** можно назначить оба обработчика или только один: **promise.then(onFulfilled)**
2. Чтобы поставить обработчик только на ошибку, вместо **then(null, onRejected)** можно писать **catch(onRejected)**
3. Если в функции внутри **Promise** происходит синхронный **throw**, то вызывается **reject().**
4. Когда **Promise** переходит в состояние «выполнен», это уже навсегда. То есть, после вызова **resolve**()/**reject**() Promise уже не может «передумать».

* let pr = new Promise((resolve, reject) => { resolve(5) });
* // Start – End – Result: 5
* alert('Start');
* pr.then(x => alert(`Result: ${x}`));
* alert('End');

# 77.Промисификация. Promise.all() и Promise.race()

***Промисификация*** – создание «обёртки», возвращающей **Promise**, для некоторого асинхронного функционала.

В качестве примера сделаем такую обёртку для запросов при помощи **XMLHttpRequest**.

Функция **httpGet**(**url**) возвращает **Promise**, который при успешной загрузке данных с **url** будет переходить в состояние **fulfilled** с этими данными, а при ошибке – в состояние **rejected** с информацией об ошибке.

*Пример:*

* function httpGet(url) {
* return new Promise(function (resolve, reject) {
* var xhr = new XMLHttpRequest();
* xhr.onload = function () {
* if (this.status == 200) {
* resolve(this.response);
* } else {
* var error = new Error(this.statusText);
* error.code = this.status;
* reject(error);
* }
* };
* xhr.onerror = () => { reject(new Error("Network Error")); };
* xhr.open("GET", url, true);
* xhr.send();
* });
* }

**ЦЕПОЧКИ PROMISE**

Метод **then**() вызывается у **Promise** и **возвращает Promise**, а значит, допускает цепочечный вызов.

Если функции-обработчики внутри **then**() сами возвращают Promise, то по цепочке передаётся результат этого **Promise**.

Если в цепочке вызовов **then**() генерируется ошибка, вызывается ближайший **then**() с обработчиком ошибок или ближайший **catch**().

* httpGet("/user/25") // делаем запрос о пользователе
* .then(response => {
* // из результата формируем объект, передаём дальше этот объект
* return JSON.parse(response);
* })
* .then(user => {
* // делаем ещё запрос
* return httpGet(`/orders/${user.name}`);
* })
* .then(response => {
* // мы получили не Promise, а его результат!
* let order = JSON.parse(response);
* alert(order.id); });

**Promise.all() и Promise.race()**

В классе **Promise** есть следующие статические методы:

* + **Promise**.**all**(**iterable**) – получает итерируемый объект из **Promise** и возвращает **Promise**, который ждёт, пока все переданные **Promise** завершатся, и переходит в состояние «выполнено» с массивом их результатов.
  + **Promise**.**race**(**iterable**) – получает итерируемый объект из **Promise** и возвращает **Promise**, который ждёт, пока первый из переданных **Promise** завершится.

# 78.Promise и генераторы

Комбинация **Promise** и генераторов позволяет писать «плоский» асинхронный код:

* + В генераторе **yield** возвращает **Promise**.
  + Вспомогательная функция **execute**() запускает генератор, и вызовами **next**() получает из него **Promise**. А когда очередной **Promise** выполнится, возвращает его результат в генератор следующим **next**().
  + Последнее значение генератора обрабатывается функцией **execute**() как окончательный результат.

*Примеры:*

*Пример* (a)

* // вспомогательная функция
* function execute(generator, yieldValue) {
* let next = generator.next(yieldValue);
* if (!next.done) {
* next.value.then(
* result => execute(generator, result), // рекурсия
* err => generator.throw(err)
* );
* } else {
* // обработаем конечный результат
* // обычно здесь вызов callback или что-то в этом духе
* alert(next.value);
* }
* }

*Пример* (b)

* function\* getOrderId() {
* let userResponse = yield httpGet("/user/25");
* let user = JSON.parse(userResponse);
* let response = yield httpGet(`/orders/${user.name}`);
* let order = JSON.parse(response);
* yield order.id;
* }
* // как это использовать:
* execute(getOrderId());

# 79.Асинхронные функции. Выражение await

***Асинхронные******функции*** (**async** **functions**) – нововведение **ES2017**, призванное упростить работу с **Promise**.

Асинхронные функции и новое *выражение* **await** позволяют писать «плоский» асинхронный код, подобно тому, как это позволяли делать генераторы.

**СИНТАКСИС**

Для создания используется ключевое слово **async**:

* Объявление асинхронной функции:

async function asyncFunc() {}

* Выражение с асинхронной функцией:

let asyncFunc = async function() {}

* Метод с асинхронной функцией:

let obj = { async asyncFunc() {} }

* Стрелочная асинхронная функция:

const asyncFunc = async () => {}

**РЕЗУЛЬТАТ АСИНХРОННОЙ ФУНКЦИИ**

***Асинхронные******функции*** не возвращают явно указанный в их теле результат. Вместо этого отдается объект-promise.

Чтобы получить реальный результат, нужно разрешить (**resolve**) promise при помощи **then**() (соответственно, реакция на исключение делается методом **catch**()).

*Пример:*

* async function mainQuestion() {
* return 42;
* }
* // использование
* let result = mainQuestion();
* alert(result instanceof Promise); // True
* alert('Start'); // Start – End – Result: 42
* mainQuestion().then(x => alert(`Result: ${x}`));
* alert('End');

**ВЫРАЖЕНИЕ AWAIT**

Внутри асинхронной функции может быть выполнен вызов другой асинхронной функции или любой функции, возвращающей **promise**.

Если этот вызов выполняется при помощи *выражения* **await**, то promise автоматически разрешается, а инструкции после await выполняются после разрешения этого promise (короче – получается плоский асинхронный код ☺).

*Пример:*

* function delayed(x) {
* return new Promise((resolve, reject) => {
* setTimeout(() => { resolve(x) }, 3000);
* });
* }
* async function add(a) {
* let b = await delayed(10);
* let c = await delayed(20);
* return a + b + c;
* }
* add(5).then(x => alert(x)); // печатает "35" через 6 секунд

*Сравнение:*

|  |  |
| --- | --- |
| **Было: цепочки Promise** | **Стало: вызовы при помощи await** |
| httpGet("/user/25") // делаем запрос о пользователе  .then(response => {   * // из результата формируем объект, передаём дальше этот объект   return JSON.parse(response);  })  .then(user => {   * // делаем ещё запрос   return httpGet(`/orders/${user.name}`);  })  .then(response => {   * // мы получили не Promise, а его результат!   let order = JSON.parse(response);  alert(order.id);  }); | async function getOrderId()  {  let userResponse = await httpGet("/user/25");  let user = JSON.parse(userResponse);  let response = await httpGet(`/orders/${user.name}`);  let order = JSON.parse(response);  return order.id;  } |

# 80.Прокси. Ловушки

***Прокси* (ECMAScript 2015)** – это особый объект, который может перехватывать обращения к другому объекту и модифицировать их при помощи *функций-ловушек*.

Синтаксис создания прокси:

* let proxy = new Proxy(target, handler);
  + **target** – обращения к этому объекту перехватываем;
  + **handler** – объект с «ловушками»: функциями-перехватчиками для операций с **target**.

**ЛОВУШКИ**

***Ловушки*** (**traps**) – функции со специальными именами.

Например, ловушка **get**() срабатывает при чтении свойства из **target**.

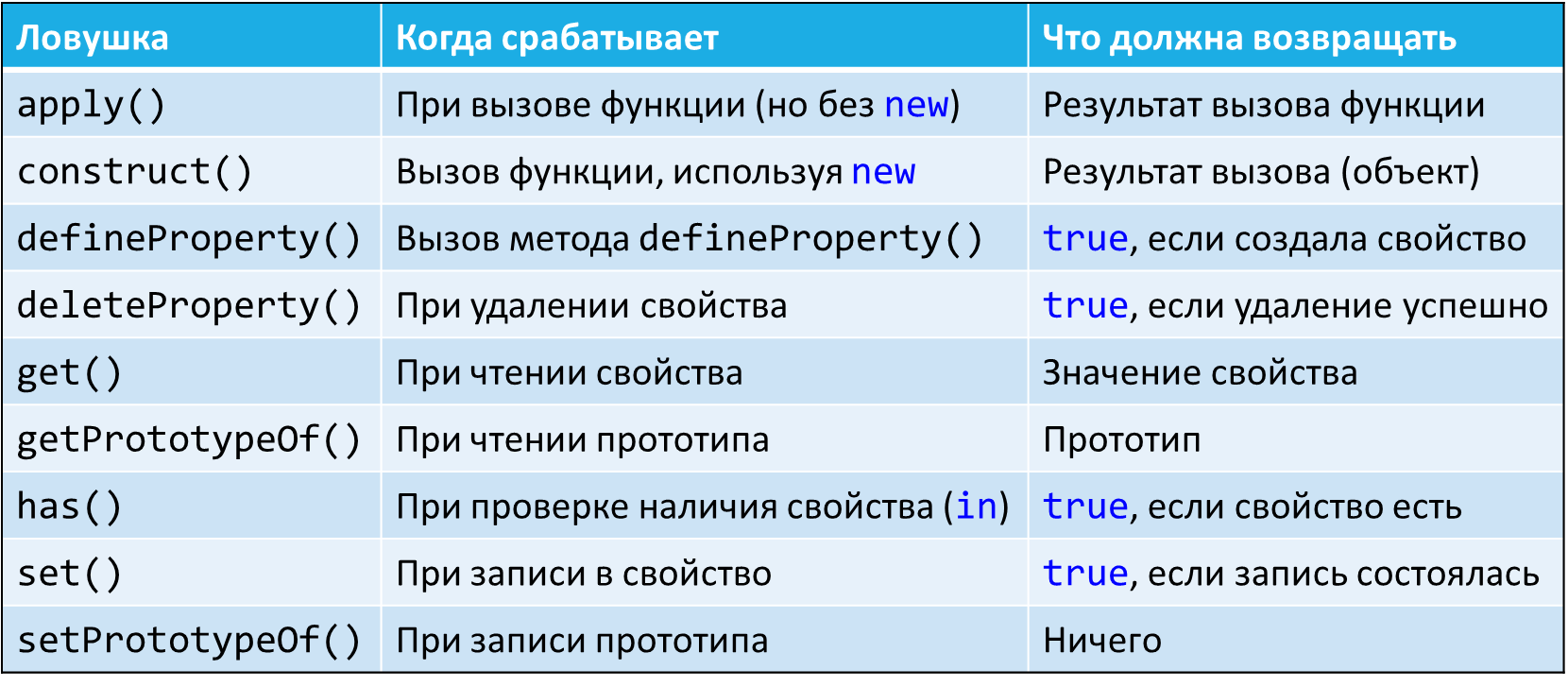
Аргументы ловушки **get**():

* + **target** – целевой объект (первый аргумент **new** **Proxy**())
  + **property** – строка с именем свойства
  + **receiver** – прокси либо его наследник (используется редко)

Пример прокси и ловушки get():

* let user = { name: "Alex" };
* let receiver = new Proxy(user, {
* get(target, property, receiver) {
* alert(target[property]);
* return target[property].toUpperCase();
* }
* });
* // выведет "Alex" при помощи alert(), а x = "ALEX"
* let x = receiver.name;

**Известные ловушки (почти все)**

******

# 81.Модули в ECMAScript 2017

С точки зрения **ECMAScript** **2017**, ***модулем*** считается отдельный файл с JS-кодом.

В этом файле ключевым словом **export** помечаются переменные и функции, которые могут быть использованы снаружи.

Другие модули могут подключать их через вызов **import**.

# 82.Экспортирование. Импортирование

Ключевое слово **export** указывается:

* + перед объявлением переменных, функций и классов;
  + отдельно, при этом в фигурных скобках указывается, что именно экспортируется.

В последнем случае при помощи **as** можно указать ***экспортируемое*** (т.е. внешнее) ***имя*** элемента.

**Внимание:** экспортируемый элемент должен иметь имя (анонимные функции экспортировать нельзя).

*Экспортирование – Пример:*

* // экспорт при объявлении
* export let one = 1;
* // отдельный экспорт + новое имя
* let two = 2;
* export {two as twice};
* // экспорт класса
* export class User {
* constructor(name) {
* this.name = name;
* }
* }

**ИПОРТИРОВАНИЕ**

Код подключает экспортированные значения из модуля при помощи ключевого слова **import**:

* import {one, twice} from "md";

Здесь "**md**" – это модуль (файл **md**.**js**); **one**, **twice** – импортируемые переменные.

В результате импорта появятся локальные переменные **one** и **twice** со значениями из модуля.

Импортирование-*Пример:*

* //------ lib.js ------
* export let counter = 3;
* export function incCounter() {
* counter++;
* }
* //------ main1.js ------
* import { counter, incCounter } from "lib";
* console.log(counter); // 3
* incCounter();
* console.log(counter); // 4
* // The imported value can't be changed
* counter++; // TypeError

При импорте можно изменить имя при помощи as:

* import {one as it1, twice as it2} from "md";

Можно импортировать все значения сразу в виде объекта вызовом **import** **\*** **as** **obj**:

* import \* as numbers from "md";

**ЭКСПОРТ ЭЛЕМЕНТА ПО УМОЛЧАНИЮ**

Для одного из экспортируемых элементов можно использовать сочетание **export** **default**:

* export default class User { . . .}

При импорте такого элемента не нужно использовать фигурные скобки:

* import User from "md";

**МОДУЛИ – ЗАМЕЧАНИЕ О СИНТАКСИСЕ**

И **export**, и **import** должны находится на верхнем уровне определения. Вложенность их в блок не допустима!

Импорты поднимаются наверх объявления:

* // это работает
* alert(one);
* import {one, twice} from "md";

**МОДУЛИ И СИСТЕМЫ СБОРКИ**

Важно: **ECMAScript** **2017** описывает, как импортировать и экспортировать значения из модулей, но он ничего не говорит о том, как эти модули искать, и загружать.

Поэтому **модули реально работают только в составе систем сборки!** Например, **Babel** в **REPL** для импорта модулей генерирует вызов функции **require**() – стандартная функция **Node.js** для загрузки модулей.

# 83.Предыстория XML: SGML. Необходимость стандарта XML

**XML** – расширяемый язык разметки,разработан и рекомендован соответствующей рабочей группой **W3C**  (Консорциумом Всемирной паутины) в **1996** г., стандартизован в **1998** г.

**ПРЕДЫСТОРИЯ XML: SGML**

*1980-е* - **SGML** (Standard Generalized Markup Language) *предшественник* ***HTML****,* ***XML*** *- стандартный обобщенный язык разметки*

* + разработан для МО США; задача - снизить расходы на передачу документации
  + четкая иерархическая структурированность информации;
  + расширяемость стандарта;
  + отделение информации от представления (использование **DTD** - **Document Type Definition**)
  + слишком сложен для реализации в **Web** (**Sounds** **Good**, **Maybe** **Later**)

**ПРЕДЫСТОРИЯ XML: HTML**

*1991 -* **HTML** (**HyperText** **Markup** **Language**) это *фиксированное подмножество* ***SGML****, ориентированное на* ***Web****-страницы*

* + чрезвычайно распространен - существует около 100 млрд. **HTML**-страниц!
  + трудности форматирования текста и невозможность отделить представление от данных (**CSS** решает эту проблему только частично)
  + недостаточная строгость стандарта и, как следствие, проблема структурирования текста

**ЗАЧЕМ НАМ ЕЩЕ ОДИН СТАНДАРТ?**

* **Необходимо** компактное, дешевое, простое и быстрое средство, аналогичное HTML, которое могло бы еще и расширяться, как SGML.
* **Необходимо** иметь единый формат для передачи данных между приложениями
* **Необходимо** создавать структурированные тексты с возможностью создания оглавлений, перекрестных ссылок и т.д.
* **Необходимо** уметь отображать нестандартную информацию (математические формулы, ноты и т.п.)

# 84.Цели XML. Основы синтаксиса XML

«**1.** **XML** должен стать языком прямого использования в **Internet**.»

«**2**. **XML** будет поддерживать большое количество приложений.»

«**3**.**XML** будет совместим с **SGML**.»

«**4**. Будет легче писать программы, обрабатывающие **XML**-документы.»

«**5**. Количество дополнительных функций в **XML** должно быть минимальным, а в идеале — нулевым.»

«**6**. **XML**-документы должны быть понятными и ясными для пользователя.»

«**7**. Разработка **XML** должна быть завершена в короткие сроки»

«**8**. Язык **XML** должен быть формальным и кратким. Спецификация **XML** должна быть четкой и краткой»

«**9**. Создание **XML**-документов должно быть максимально простым»

«**10**. Краткость команд разметки **XML** не имеет принципиального значения»

**МИФЫ ОБ XML:**

**XML** - это язык разметки (на самом деле - это метаязык для создания языков разметки)

**XML** - это только для **Web** (на самом деле, многие компании переходят на него с **SGML**)

**XML** - это подмножество **SGML** (раньше был, но теперь ввели схемы, пространства имен и т.д.)

**HTML** - это подмножество **XML** (такое описание в принципе возможно, но бессмысленно; кроме того, есть вопрос следования стандарту **HTML**)

**XML** - это еще один рекламный трюк (увы, нет)

**ОСНОВЫ СИНТАКСИСА XML**

* У каждого элемента должен быть открывающий и закрывающий тэг
* В документе должен быть ровно один корневой элемент
* Элементы могут быть вложены друг в друга, но не могут пересекаться (т.е. деревянная структура)
* Все названия элементов чувствительны к регистру
* Некоторые символы запрещены (надо использовать специальные последовательности)
* Значения атрибутов должны быть взяты в кавычки

# 85. Структура тэгов XML

Должен быть открывающий и закрывающий тэг:

<duration course=".NET">1.5</duration>

Здесь duration - это название тэга, course – это атрибут, а 1.5 – значение (содержимое элемента).

Должны соблюдаться правила вложенности:

<P>Права <B>одного <I>человека</B> важнее</I>

прав коллектива - **неправильно!!!**

Вместо этого правильно так:

<P>Права <B>одного <I>человека</I></B>

<I>важнее</I> прав коллектива

**КОРНЕВОЙ ТЭГ**

Следующий *Пример* некорректен:

**<!-- WRONG! -->**

**<a> ... </a>**

**<b> ... </b>**

Вместо этого должно быть:

**<root>**

**<a> ... </a>**

**<b> ... </b>**

**</root>**

# 86.Описания структуры документа DTD XML

**В XML документе можно выделить 2 основные части:**

1. Описания структуры документа **DTD** –(Document Type Definition) *(для определения логической структуры документов использовался набор формальных правил, называемый DTD — декларацией типа документа****)***
2. Непосредственно содержание документа

В первой части мы можем использовать:

* Инструкции XML – процессора
* Объявление элементов структуры документа (*являются лишь контейнерами для хранения данных*)
* Атрибуты для каждого элемента
* Cущности

**DTD**-блок может внедряться как в сам документ, так и находиться во внешнем файле.

В **XML** – документе, как и в любой объектной иерархии существует некий корневой элемент, от которого наследуются все остальные.

Содержимое **XML** – документа (2-ая часть) форматируется при помощи тэгов, которые определяются в описании типа документа.

* Наименования тэгов полностью совпадают с наименованием элементов
* Параметры тэгов позволяют устанавливать значения атрибутов элементов

# 87.Инструкция XML – процессора.

В качестве первой строки каждого **XML** – документа должна использоваться исполняемая инструкция, предназначенная для **XML** – процессора:

<?xml version=“1.0”?>

**xml** – ключевое слово для каждой исполняемой инструкции;

**version** - параметр инструкции, указывающий на то, что будет использоваться первая версия стандарта **XML**.

**Пролог XML** – документа – блок исполняемых инструкций.

Инструкция для указания конкретной кодировки, которая будет использоваться:

<?xml encoding=“UTF-8”?>

**encoding** – параметр инструкции;

**UTF-8** – одна из наиболее часто используемых кодировок.

**Заголовок чувствителен к регистру символов!**

# 88. Элементы XML-документа.

Метаязык **XML** предназначен для создания конкретных языков разметки

Для конкретных языков разметки определены наилучшим образом подходящие для описываемой информации

* + наборы элементов;
  + взаимосвязь между элементами
  + атрибуты элементов.

Для конкретных языков разметки нужно описывать их синтаксис

Описание синтаксиса позволяет формальным образом проверять правильность составления **XML**-документов.

В **XML**-технологии имеется два языка для описания синтаксиса конкретных языков разметки:

* + язык **Document Type Defenition** (**DTD**) – описывает структуру документа с помощью декларативных правил
  + разработан совместно с метаязыком SGML и не использует синтаксис языка XML;
  + языка **XML Schema** – описывает структуру документа с использованием синтаксиса языка **XML** (специально разработан для метаязыка **XML**).

Формальное описание синтаксиса конкретного языка не является обязательным.

Если с помощью конкретного языка разметки требуется разработать только несколько однотипных **XML**-документов, то формально описывать его синтаксис не обязательно.

Если с помощью конкретного языка разметки

* + создается много **XML**-документов, или
  + данный язык разметки используется разными пользователями,

то ***формальное описание синтаксиса желательно***.

# 89.Атрибуты элементов XML +

# (91.Модификаторы атрибутов XML)

**ОБЪЯВЛЕНИЕ АТРИБУТОВ**

В валидном **XML**-документе должны быть полностью описаны все атрибуты, которые предполагается использовать для элементов документа.

Все атрибуты, ассоциированные с определенным элементом, описываются с помощью специального типа **DTD**-разметки, называемого объявлением списка атрибутов.

В объявлении атрибутов задаются:

* + **имена** атрибутов, ассоциированных с элементом.
    - В валидном документе можно включить в начальный тэг элемента только те атрибуты, которые определены для элемента;
  + **тип данных** каждого атрибута;
  + **обязательность** для каждого атрибута.

Если атрибут необязателен, то указывается, что должен делать процессор, если атрибут опущен

Например, можно задать значение атрибута по умолчанию, которое будет использовать процессор.

**ФОРМА ЗАПИСИ ОБЪЯВЛЕНИЯ СПИСКА АТРИБУТОВ**

Объявление списка атрибутов имеет следующую общую форму:

<!ATTLIST [ИмяЭлемента] [ОпределенияАтрибутов] >

* + [**ИмяЭлемента**] – имя элемента, связанного с атрибутом или атрибутами;
  + [**ОпределенияАтрибутов**] – это одно или несколько определений атрибутов, каждое из которых определяет один атрибут.

Каждое определение атрибута является последовательностью следующих трех значений:

[ИмяАтрибута] [ТипАтрибута] [ДопОписание]

* + [**ИмяАтрибута**] – имя атрибута;
  + [**ТипАтрибута**] – тип атрибута, т.е. виды значений, которые могут быть присвоены атрибуту;
  + [**ДопОписание**] – это объявление по умолчанию, которое указывает на обязательность атрибута и содержит другую информацию.

Если для данного типа элемента будет задано более одного объявления списка атрибутов, содержания двух объявлений объединяются.

Если атрибут с заданным именем объявлен для одного и того же элемента несколько раз, первое объявление используется, а последующие – игнорируются.

Множественные объявления списков атрибутов может возникнуть, если документ имеет как внутренние, так и внешние **DTD**.

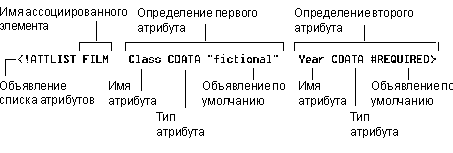
*Пример определения атрибутов:*

Пусть тип элемента с именем **FILM** объявлен следующим образом:

<!ELEMENT **FILM** (TITLE, (STAR | NARRATOR | INSTRUCTOR) )>

Тогда можно объявить два атрибута (**Class** и **Year**) для элемента **FILM**:

<!ATTLIST **FILM** Class CDATA "fictional" Year CDATA #REQUIRED>



Атрибуту **class** можно присваивать любую строку в кавычках (ключевое слово **CDATA**);

* + если в **XML**-документе данному атрибуту значение не задается, то ему будет автоматически присваиваться значение по умолчанию "fictional".

Атрибуту year можно присвоить любую строку в кавычках;

* + однако, этому атрибуту должно быть обязательно присвоено значение в каждом элементе **film** (ключевое слово #**REQUIRED**), поэтому значение по умолчанию не задается.

*Пример описания XML-документа:*

<?xml version="1.0" encoding="windows-1251" ?>

<!DOCTYPE FILM

[

<!ELEMENT FILM (TITLE, (STAR | NARRATOR | INSTRUCTOR) )>

<!ATTLIST FILM Class CDATA "fictional" Year CDATA #REQUIRED>

<!ELEMENT TITLE (#PCDATA)>

<!ELEMENT STAR (#PCDATA)>

<!ELEMENT NARRATOR (#PCDATA)>

<!ELEMENT INSTRUCTOR (#PCDATA)>

]

>

<FILM Year="1948">

<TITLE>The Morning After</TITLE>

<STAR>Morgan Attenbury</STAR>

</FILM>

**ТИП АТРИБУТА**

Тип атрибута является вторым необходимым компонентом в определении атрибута.

Он задает вид значений, которые вы можете присваивать атрибуту внутри документа.

Тип атрибута может задавать тремя различными способами:

* + **Строковый тип** - атрибуту может быть назначена любая строке в кавычках (литералу). Для задания такого типа используется ключевое слово **CDATA**.

*Пример:*

<!ATTLIST FILM Class CDATA "fictional">

* + **Маркерный тип** - можно задать значения, на которые наложен ряд ограничений.
  + **Нумерованный тип** – можно задать одно или список определенных значений.

**ЗАДАНИЕ МАРКЕРНОГО ТИПА АТРИБУТА**

Значение, присваиваемое маркерному типу атрибута, должно удовлетворять дополнительному ограничению, которое задается в описании атрибута с помощью ключевых слов:

* + **ID** – идентификатор (уникальное значение);
  + **IDREF** – ссылка на идентификатор;
  + **IDREFS** – набор ссылок на идентификаторы;
  + **ENTITY** – не анализируемая сущность;
  + **ENTITIES** – набор не анализируемых сущностей.

***МАРКЕРНЫЙ ТИП ID***

**ID** - для каждого элемента атрибут должен иметь уникальное значение.

Значение должно начинаться с буквы или символа подчеркивания **(\_),** за которыми могут идти или не идти другие буквы, цифры, символы точки **(.),** тире **(–)** или символы подчеркивания.

Данный тип элемента может иметь только один атрибут типа **ID**, а в объявлении значения атрибута по умолчанию должно фигурировать #**REQUIRED** или #**IMPLIED**.

***Маркерные типы IDREF и IDREFS,* *ENTITY и ENTITIES ,* *NMTOKEN и NMTOKENS***

1. **IDREF** - значение атрибута должно совпадать со значением атрибута элемента типа **ID** внутри документа.

Другими словами, этот тип атрибута является ссылкой на уникальный идентификатор другого атрибута.

1. **IDREFS** - этот тип атрибута похож на тип **IDREF**, но при этом значение может включать ссылки на несколько идентификаторов – разделенных пробелами – внутри строки в кавычках.
2. **ENTITY** - значение атрибута должно совпадать с именем сущности (примитива), объявленного в **DTD**.

Примитив не обрабатывается синтаксическим анализатором и ссылается на внешний файл, обычно содержащий не XML-данные.

1. **ENTITIES** - значение атрибута может содержать имена нескольких сущностей (не анализируемых примитивов)

Они разделяются пробелами – внутри строки в кавычках.

1. **NMTOKEN** - значением атрибута должно быть элементарное имя (**name** **token**),
   * состоит из одной или нескольких букв, цифр, точек (.), тире (–) или символов подчеркивания (\_).
   * может также содержать двоеточие (:), но не на первом месте.
2. **NMTOKENS** - атрибут может содержать несколько элементарных имен
   * имена разделяются пробелами – внутри строки в кавычках.

**ПЕРЕЧИСЛЯЕМЫЙ ТИП АТРИБУТОВ**

Значение атрибута должно совпадать с одним из имен, приведенных в списке типов атрибутов.

Эти имена могут иметь одну из следующих двух форм записи.

* + Открывающая скобка, вслед за которой идет список элементарных имен, разделенных символами **|** , после чего следует закрывающая скобка.
  + Ключевое слово **NOTATION**, за которым идет пробел, затем открывающая скобка, затем список имен нотаций, разделяемых символами **|** , после чего следует закрывающая скобка.
    - Каждое из этих имен должно точно соответствовать имени нотации, объявленному в **DTD**.
    - Нотация описывает формат данных или идентифицирует программу, применяемую для обработки определенного формата.

**ДОПОЛНИТЕЛЬНОЕ ОПИСАНИЕ АТРИБУТА**

Третий (последний) обязательный компонент в определении атрибута указывает:

* + является ли атрибут обязательным,
  + если нет, то задает, что должен предпринимать процессор в случае, когда атрибут опущен.

Так, объявление должно обеспечить значение атрибута по умолчанию, которое будет использовать процессор в том случае, если атрибут отсутствует.

**ОБЪЯВЛЕНИЕ ЗНАЧЕНИЙ ПО УМОЛЧАНИЮ**

Объявление обязательности атрибута и его значения по умолчанию может иметь следующие четыре формы:

1. #**REQUIRED** – требуется обязательно задавать значение атрибута для каждого элемента.
2. #**IMPLIED** - можно либо включить, либо опустить атрибут, если атрибут опущен, то никакое значение по умолчанию процессору не передается.
3. **AttValue** (значение атрибута по умолчанию) - можно либо включить, либо опускать атрибут для элемента.
   1. Если атрибут опускается, то процессор использует значение по умолчанию, как если бы атрибут был включен и задано его значение.
4. #**FIXED** **AttValue** (значение атрибута по умолчанию) - можно либо включать, либо опускать атрибут для элемента ассоциированного типа.
   1. если атрибут опускается, то процессор будет использовать значение, заданное по умолчанию;
   2. если атрибут включается, то ему должно быть задано значение по умолчанию (у атрибута м.б. только одно значение).

**ВКЛЮЧЕНИЕ В XML-ДОКУМЕНТ ВНЕШНИХ ДАННЫХ**

Хотя метаязык **XML** в основном разработан для создания языков описания текстовой информации.

Но он также предоставляет средство взаимодействия с другими данными

* + например, двоичными данными, такими, как растровые изображения.

Для обработки данных других типов нужно

* + задать их тип с помощью ***нотации*** и
  + включить их в **XML**-документ с помощью ***сущности***.

Кроме не текстовых типов можно использовать и текстовые данные специальных форматов

* + например, даты.

**НОТАЦИИ**

Под ***нотацией*** понимается *специальная метка*, сообщающая **XML**-процессору об используемом типе данных.

Одним из применений нотаций является описание текстовых данных, имеющих особый формат, например, дат.

Нотация описывает определенный формат данных.

Для этого указывается

* + адрес описания формата,
  + адреса программы, которая может обрабатывать данные в этом формате,
  + либо просто описание формата.

Нотация может использоваться,

* + для описания формата общей внешней не анализируемой сущности
  + либо можете присвоить нотацию атрибуту, который имеет перечисляемый тип **NOTATION**.

**ОБЪЯВЛЕНИЕ НОТАЦИИ**

Объявление нотации имеет следующую форму записи:

<!NOTATION [ИмяНотации] SYSTEM [СистемЛитерал]>

* + [**ИмяНотации**] – имя нотации;
  + [**СистемЛитерал**] – любое известное описание формата
  + ограничен одинарными (') или двойными (") кавычками;
  + должен быть уникальным;
  + должен сообщать **XML**-процессору достаточно информации для обработки данных;
  + сообщает **XML**-процессору, как отображать или обрабатывать включаемые в XML-документ данные.

Объявление нотации создает метку, которая используется вместе

* + с объявлением атрибута или
  + не анализируемой внешней сущностью.

**ОБЪЯВЛЕНИЕ СИСТЕМНОЙ НОТАЦИИ С ПОМОЩЬЮ ССЫЛКИ НА ОБРАБАТЫВАЮЩУЮ ПРОГРАММУ**

С помощью указания **URI** программы, которая может обрабатывать или отображать формат данных.

*Пример:*

<!NOTATION bmp SYSTEM "Pbrush.exe">

<!NOTATION gif SYSTEM http://bogus.com/ShowGif.exe">

# 90. Сущности XML.

В языке **XML** ***сущности*** (примитивы, **entity**) – это

* + средство встраивания в **XML**-документ многократно повторяющихся блоков текста,
  + способ встраивать различных, не текстовых типов данных.

Сущностями могут быть

* + часто используемые блоки текста, что позволяет быстро вставлять их в нужные места.
  + внешние файлы, чтобы иметь возможность включать их содержание в **XML**-документ;
    - в таких файлах могут содержаться текстовые или двоичные данные.

Определение сущностей выполняется аналогичного тому, как объявляются элементы и атрибуты конкретного языка.

**ТЕКСТОВЫЕ СУЩНОСТИ**

В DTD могут описываться следующие типы сущностей:

**Текстовая сущность** это простая текстовая строка, для которой задается имя, на*Пример:*

<!ENTITY abc "The ABC Group">

Для вставки данной строки в **XML**-документе используется ссылка на нее, в виде «&abc;». **Внешняя текстовая сущность** содержит текст из внешнего источника.

*Примеры описания:*

* + Задание внешней общей сущности посредством его формального открытого идентификатора:

<!ENTITY man PUBLIC -"//Acme Gadgets//TEXT Manual 23//EN"

"http://www.acme-gadgets.com/manuals/prod23.htm">

**НЕ АНАЛИЗИРУЕМЫЕ ВНЕШНИЕ СУЩНОСТИ**

Сущности, используемые для импорта не текстовых данных, называется ***не анализируемыми сущностями*** (unparsed entity).

Объявления не анализируемой и текстовых сущностей аналогичны, за исключением

* + ключевого слова NDATA и
  + типа ***нотации***, следующих за системным или открытым идентификатором.

**ВСТАВКА ССЫЛОК НА СУЩНОСТИ**

Не анализируемые сущности непосредственно в **XML**-документ не встраиваются.

Вместо этого ссылки на них передаются в элемент с помощью атрибутов, имеющих тип **ENTITY** или **ENTITIES**.

Вставка содержимого (замещающий текст) сущности (примитива) в документ осуществляется с помощью ссылок на сущности.

Ссылки на сущность имеет следующий общий вид: “&**ИмяСущности**;”.

# 92. Особые символы XML.

Некоторые символы необходимо заменять на подстановочные строки:

<P>Было предложено соотношение A<B, но оно не

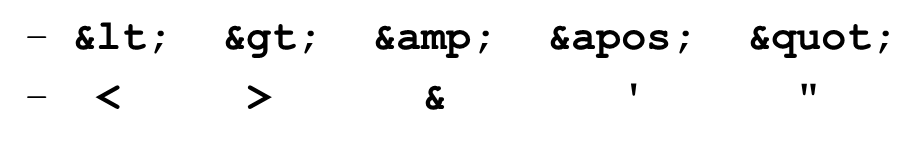
подошло.</P>

Должно быть:

<P>Было предложено соотношение A&lt;B, но оно

не подошло.</P>

Пять подстановочных строк:

****

# 93. Корректно сформированные XML-документы.

***Правильные (корректно сформированные, well-formed) XML документы*** соответствуют основным правилам **XML** документов.

Каждый **XML**-документ должен быть правильным, т.е. отвечать минимальным требованиям по составлению **XML**-документа.

Если документ не является правильным, он не может считаться **XML**-документом.

Правильный **XML**-документ также может быть действительным (валидным).

Действительные **XML**-документы соответствуют более строгим критериям.

Действительным (валидным, **valid**) называется правильный (**well**-**formed**) документ, отвечающий двум дополнительным требованиям:

* + пролог документа должен содержать указание на описание синтаксиса данного документа
  + На*Пример:* определение типа документа (**DTD**), задающее структуру документа;
  + остальной документ должен отвечать структуре, заданной в описании синтаксиса.

# 94. Определение типа документа (DTD) XML.

Язык **Document Type Definition** (**DTD)** используется для описания схем (синтаксиса) конкретных языков разметки.

Данный язык был унаследован от языка **SGML**.

Описание конкретного языка разметки с помощью языка **DTD** может

* + включаться в **XML**-документ (внутреннее **DTD**-описание)
  + или содержаться в отдельном файле (внешнее **DTD**-описание).

Для включения **DTD**-описания в начало **XML**-документа записывается элемент

<!DOCTYPE … >

Если **DTD**-описание языка разметки включается в **XML**-документ, то элемент **DOCTYPE** записывается следующим образом:

<!DOCTYPE «имя\_корневого\_элемента»

[

«объявление1»

…

«объявлениеN»

]>

**ССЫЛКА НА ОПИСАНИЕ СИНТАКСИСА ДОКУМЕНТА**

Если в этом документе используется внешнее **DTD**-описание (обычно содержащееся в файле с расширением **dtd**), то **DOCTYPE** будет иметь следующий вид:

<!DOCTYPE [имя\_корневого\_элемента] [тип] [идентификатор] [адрес]>

* [имя\_корневого\_элемента] – указывает имя корневого элемента конкретного языка разметки (оно должно в точности соответствовать имени корневого элемента, записанного в XML-документе);
* [тип] – может принимать одно из двух значений:
* SYSTEM – **DTD**-описание является закрытым, не для общего распространения,
* PUBLIC – **DTD**-описание является открытым для общего использования;
* [идентификатор] – формальный открытый идентификатор для типа PUBLIC;
* [адрес] – адрес файла или **URL**.

**ССЫЛКА НА ФАЙЛ С DTD-ОПИСАНИЕМ**

Если описание конкретного языка разметки, содержится в файле **BookDTD**, то **XML**-документ должен включать следующую запись:

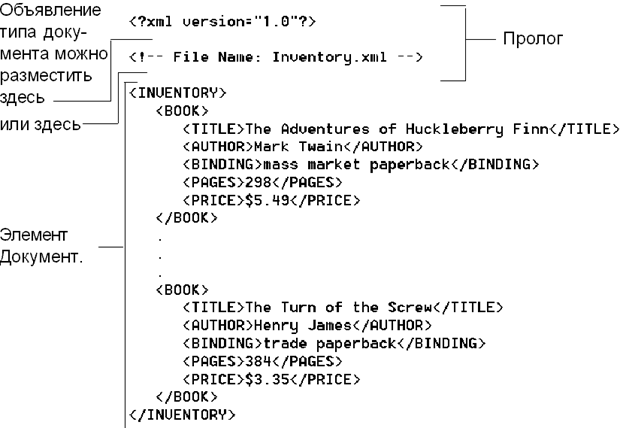
<!DOCTYPE Библиотека SYSTEM BookDTD.dtd>

**ССЫЛКА НА DTD-ОПИСАНИЕ ОПУБЛИКОВАННОЕ В СЕТИ**

Ссылка на открытое DTD-описание для конкретного языка XHTML записывается следующим образом:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

**ОБЪЯВЛЕНИЕ ТИПА ДОКУМЕНТА**

****

**ФОРМА ЗАПИСИ DOCUMENT TYPE DEFINITION (DTD)**

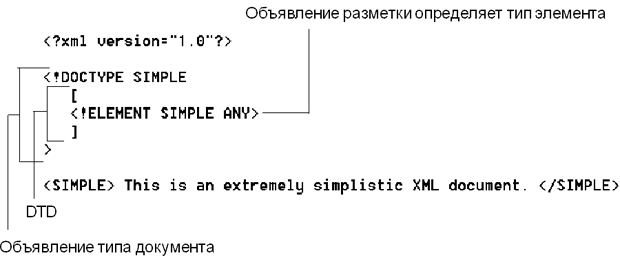
Объявление типа документа имеет следующую обобщенную форму записи:

**<!DOCTYPE Имя DTD>**

* + **Имя** – название корневого тэга XML документа.
  + **DTD** – определение типа документа.

**DTD** состоит из символа левой квадратной скобки “[” , после которой следует ряд объявлений разметки, заканчивающихся правой квадратной скобкой “]”. Объявления разметки описывают логическую структуру документа; т.е. задают элементы документа, атрибуты и другие компоненты.

*Пример* валидного **XML**-документа, содержащего **DTD** с единственным объявлением разметки, которое определяет один тип элемента в документе, **SIMPLE**:



**ОБЪЯВЛЕНИЯ ЯЗЫКА DTD**

Описание конкретного языка разметки на языке **DTD** состоит из ***объявлений***, которые описывают логическую структуру документа.

Имеются следующие основные типы объявлений:

* + **объявления типов элементов** – определяют типы элементов, которые может содержать документ, а также их содержимое и порядок следования элементов;
  + **объявления списков атрибутов –** задают имена атрибутов, которые могут быть использованы с определенным типом элемента, а также типы данных и устанавливаемые по умолчанию значения этих атрибутов;
  + **объявления сущностей –** используются для описания часто используемых фрагментов текста или для встраивания в **XML**-документ не текстовых типов данных;
  + **объявления нотаций –** могут использоваться для описания форматов внешних данных.

**ОБЪЯВЛЕНИЕ ТИПОВ ЭЛЕМЕНТОВ**

В валидном **XML**-документе требуется полностью объявить тип каждого элемента, который используется в документе, в объявлении типа элемента внутри **DTD**.

В объявлении типа элемента задается

* + имя элемента и
  + его допустимое содержимое (часто описывающее порядок размещения дочерних элементов), в частности, указывает типы и порядок следования элементов, которые данный тип элемента может содержать.

Объявление типа вложенного элемента имеет следующую общую форму:

<!ELEMENT [имя] [содержание]>

* + **Имя**  – имя объявляемого типа элемента;
  + **опись\_содержимого** – это описание содержимого, определяющего, что может содержать элемент.

**МОДЕЛИ СОДЕРЖИМОГО ЭЛЕМЕНТА**

Содержимое элемента (то, что находится между открывающим и закрывающим тэгами элемента) может быть описано четырьмя различными способами (с использованием разных моделей содержимого):

**Пустое содержимое** (**EMPTY**) – элемент должен быть пустым – т.е. не должен иметь содержимого.

**Любое содержимое** (**ANY**) – указывает, что элемент может иметь любое допустимое для этого типа содержимое (наиболее неопределенный тип описания содержимого и позволяет создавать типы элементов без ограничений на их содержимое).

**Дочернее содержимое** – элемент может содержать только вложенные элементы, но не может символьные данные (будет рассматриваться далее).

**Смешанное содержимое** – элемент может содержать комбинацию дочерних элементов определенного типа и символьных данных (самый общий вариант описания с ограничением содержимого элементов).

**Пустое содержимое:**

EMPTY - **пустое содержимое,** элемент должен быть пустым – т.е. не должен иметь содержимого.

<!ELEMENT image EMPTY>

В этом случае элементы **image**, могут записываться в **XML**-документе следующим образом:

<image></image> или <image />

**Любое содержимое:**

**ANY** - **любое содержимое,** указывает, что элемент может иметь любое допустимое для этого типа содержимое

Наиболее неопределенный тип описания содержимого и позволяет создавать типы элементов без ограничений на их содержимое.

<!ELEMENT misc ANY>

**Задание дочернего содержимого элемента:**

Элемент с «дочерним содержимым» могут непосредственно содержать только определенные дочерние элементы, но не символьные данные.

*Пример* описания элемента **book** имеющего дочернее содержимое:

<!ELEMENT book (title, author)>

В данном случае

* + элементы **book** должны иметь только один дочерний элемент **title**,
  + за которым следует ровно один дочерний элемент **author**.

***Пример* объявления типов элементов:**

Например, объявление типа элемента с именем TITLE, для содержимого которого могут использоваться только символьные данные (дочерние элементы не допускаются):

**<!ELEMENT TITLE (#PCDATA)>**

Объявление для типа элемента с именем GENERAL, содержимое которого может быть любым:

**<!ELEMENT GENERAL ANY>**

**ЗАДАНИЕ ДОЧЕРНЕГО СОДЕРЖИМОГО ЭЛЕМЕНТА**

Элемент может непосредственно содержать только определенные дочерние элементы, но не символьные данные.

В тексте документа дочерние элементы можно разделять символами пробела, табуляции, возврата каретки или перевода строки, чтобы улучшить восприятие документа, но процессор будет игнорировать эти символы и не передаст их приложению.

*Пример* ***XML****-документа, описывающего книгу:*

<?xml version="1.0" encoding="windows-1251" ?>

<!DOCTYPE BOOK

[

<!ELEMENT BOOK (TITLE, AUTHOR)>

<!ELEMENT TITLE (#PCDATA)>

<!ELEMENT AUTHOR (#PCDATA)>

]

>

<BOOK>

<TITLE>The Scarlet Letter</TITLE>

<AUTHOR>Nathaniel Hawthorne</AUTHOR>

</BOOK>

**МОДЕЛИ СОДЕРЖИМОГО ЭЛЕМЕНТА**

**Последовательная модель содержимого**

* + Элемент должен иметь строгую заданную последовательность дочерних элементов.

**Выборочная модель содержимого**

* + Выборочная форма модели содержимого указывает, что элемент может иметь только один (любой) из набора допустимых дочерних элементов

**ПОСЛЕДОВАТЕЛЬНАЯ МОДЕЛЬ СОДЕРЖИМОГО (SEQUENCE)**

Элемент должен иметь строгую заданную последовательность дочерних элементов.

Имена типов дочерних элементов должны отделяться запятыми.

Неправильный порядок, пропуск дочернего элемента или использование одного и того же типа дочернего элемента более одного раза также недопустимо.

*Пример последовательной модели содержимого:*

<!DOCTYPE MOUNTAIN

[

<!ELEMENT MOUNTAIN (NAME, HEIGHT, STATE)>

<!ELEMENT NAME (#PCDATA)>

<!ELEMENT HEIGHT (#PCDATA)>

<!ELEMENT STATE (#PCDATA)>

]>

**ВЫБОРОЧНАЯ МОДЕЛЬ СОДЕРЖИМОГО (CHOICE)**

Элемент может иметь ***только один*** (любой) из набора допустимых дочерних элементов, разделяемых символом “|”.

*Пример:*

<!DOCTYPE FILM

[

<!ELEMENT FILM (STAR | NARRATOR | INSTRUCTOR)>

<!ELEMENT STAR (#PCDATA)>

<!ELEMENT NARRATOR (#PCDATA)>

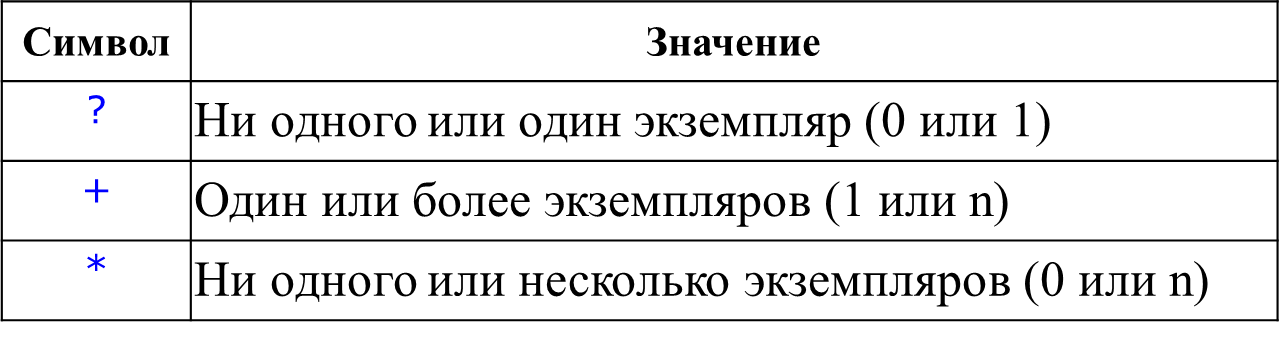
<!ELEMENT INSTRUCTOR (#PCDATA)>

]>

**УТОЧНЕНИЕ ФОРМЫ СОДЕРЖИМОГО**

В любой форме модели содержимого после имени элемента можно задать символы: знаки **вопроса (?), плюса (+)** и **звездочки (\*).**

Смысл использования этих знаков поясняется в следующей таблице:



*Пример:*

<!ELEMENT MOUNTAIN (NAME+, HEIGHT?, STATE)>

Это означает

* + можно включить один или более дочерний элемент **NAME**,
  + дочерний элемент **HEIGHT** является не обязательным,
  + дочерний элемент **STATE** является обязательным.

**УТОЧНЕНИЕ ВСЕЙ МОДЕЛИ СОДЕРЖИМОГО**

Символы ?, + или \* также могут использоваться для модификации всей модели содержимого.

Для этого они записываются непосредственно после закрывающих скобок.

*НаПример:*

<!ELEMENT FILM (STAR | NARRATOR | INSTRUCTOR)+>

дает возможность включить один или несколько дочерних элементов любого из этих трех типов в любом порядке.

**СЛОЖНЫЕ МОДЕЛИ СОДЕРЖИМОГО**

Формируются путем

* + вложения выборочной модели содержимого внутрь последовательной модели,
  + либо последовательной модели в выборочную модель.

*НаПример:*

<!DOCTYPE FILM [

<!ELEMENT FILM (TITLE, CLASS, (STAR | NARRATOR | INSTRUCTOR) )>

<!ELEMENT TITLE (#PCDATA)>

<!ELEMENT CLASS (#PCDATA)>

<!ELEMENT STAR (#PCDATA)>

<!ELEMENT NARRATOR (#PCDATA)>

<!ELEMENT INSTRUCTOR (#PCDATA)>

]>

**ЗАДАНИЕ СМЕШАННОГО СОДЕРЖИМОГО**

Элемент может включать символьные данные и дочерние элементы.

Если задан один или несколько типов дочерних элементов, то элемент может содержать

* + любые из этих дочерних элементов
  + в любом порядке и
  + с любым количеством вхождений (нуль и более).

При смешанном содержимом

* + можно задавать типы дочерних элементов,
  + нельзя задавать порядок или количество вхождений дочерних элементов,
  + нельзя задавать обязательность включения для определенных типов дочерних элементов.

Для объявления типа элемента смешанного содержимого, можно воспользоваться одной из следующих форм модели содержимого:

1. **#PCDATA** - только символьные данные (может и не иметь никаких символов – т.е. быть пустым).
2. Символьные данные с необязательными дочерними элементами.
   * после ключевого слова **PCDATA** записывается один или несколько дочерних элементов
   * разделяя их символами **|** и помещая звездочку **(\*)** в конце всей модели содержимого.

*Пример смешанного содержимого:*

Следующее объявление указывает, что элемент **TITLE** может содержать символьные данные плюс ни одного или несколько дочерних элементов **SUBTITLE**:

<!ELEMENT TITLE (#PCDATA | SUBTITLE)\*>

# 95. Проблемы DTD XML.

Недостатки **XML** **DTD**:

* Отличный от **XML** синтаксис языка. Это вызывает множество проблем, таких как, например, проблемы с кодировкой или невозможность отслеживать ошибки.
* Нет проверки типов данных. В **DTD** есть только один тип – строка.
* В DTD нет пространств имен. Нельзя поставить в соответствие документу два и более **DTD** описаний.

# 96. Схема XML. Метод разрешения неоднозначности имен при соединении двух файлов.

**СИНТАКСИС – СХЕМА**

В **XML**-технологии вместо термина «синтаксис» чаще используется термин «схема».

Отсюда и название языков:

* + **XML** **Schema**,
  + **RDF** **Schema**.

**ЧТО ВКЛЮЧАЮТ СХЕМЫ КОНКРЕТНЫХ ЯЗЫКОВ РАЗМЕТКИ?**

Множество допустимых элементов;

* + если специально не определено, то нельзя использовать другие имена элементов, кроме содержащихся в этом множестве (словарь языка);

Шаблон каждого элемента, который определяет,

* + какие элементы или данные могут находиться внутри элемента,
  + в каком порядке,
  + сколько их может быть,
  + являются ли они обязательными;

Набор разрешенных атрибутов;

* + каждое объявление атрибута определяет его имя, тип данных,
  + значения атрибутов по умолчанию (если они есть) и
  + обязательность из задания.

# 97. XML и стандарт XSLT

**XSL** позволяет описывать внешний вид (форматирование) **XML**-документа

Реально **XSL** состоит из двух документов: первый относится к преобразованию документов **XML**, а второй – к интерпретации форматирования

Язык преобразования описан в стандарте **XSLT** (**XSL** **Transformations**)

Являясь очень мощным средством оформления **HTML**- страниц, **CSS**- таблицы, тем не менее, не могут применяться в **XML**-документах, т.к. набор тэгов в этом языке не ограничен и использование статических ссылок на форматируемые объекты документа в этом случае невозможно.

Поэтому для форматирования **XML**- элементов был разработан новый язык разметки, являющийся подмножеством **XML**, и специально был предназначен для форматирования **XML**- элементов

**ПРИНЦИП ОБРАБОТКИ XML- ДОКУМЕНТОВ СТИЛЕВЫМИ ТАБЛИЦАМИ ЗАКЛЮЧАЕТСЯ В СЛЕДУЮЩЕМ:**

при разборе **XSL**-документа программа-анализатор обрабатывает инструкции этого языка и каждому элементу, найденному в **XML**- дереве ставит в соответствие набор тэгов, определяющих форматирование этого элемента

мы задаем шаблон форматирования для **XML**- элементов, причем сам этот шаблон может иметь структуру соответствующего фрагмента **XML**-документа.

Инструкции **XSL** определяют точное месторасположение элемента **XML** в дереве, поэтому существует возможность применять различные стили оформления к одинаковым элементам, в зависимости от контекста их использования.

**СТРУКТУРА XSL- ТАБЛИЦ**

**XSL**- документ представляет собой совокупность правил построения, каждое из которых выделено в отдельный блок, ограниченный тэгами **<rule>** и **</rule>**;.

Правила определяют шаблоны, по которым каждому элементу **XML** ставится в соответствие последовательность **HTML**- тэгов, т.е. внутри них содержатся инструкции, определяющие элементы **XML**- документа и тэги форматирования, применяемые к ним.

Элементы **XML**, к которым будет применяться форматирование, обозначаются в **XSL** дескриптором **<target-element/>**;.

Для указания элемента с конкретным названием (название элемента определяется тэгами, его обозначающими), т.е. определения класса элемента, можно использовать атрибут **type**="<имя\_элемента>"

**ПРИОРИТЕТЫ ПРАВИЛ**

(в порядке убывания приоритета):

* правила, помеченные специальным тэгом <importance>
* правила с наибольшим значением атрибута id, если он определен
* правила с наибольшим значением атрибута class, если он определен
* правила, имеющие наибольшую вложенность, определяемую тэгом <element>
* правила, использующие атрибут type совместно с <target-element>
* правила, в КОТОРЫХ отсутствует атрибут type в <target-element> или <element>
* правила с более высоким приоритетом, задаваемым атрибутом priority тэга <rule>
* правила с наибольшим значением квалификаторов <only>, <position>, <attribute>

**ФИЛЬТРАЦИЯ ЭЛЕМЕНТОВ**

Одним из самых мощных средств **XSL** является возможность сортировки и выборки элементов, выделяемых из общего дерева элементов документа.

Для этого используется элемент **<select-elements>**;, который заменяет <**children**/> в правилах, определяя те элементы, которые следует обработать в процессе рекурсивного обхода.

В примере будут обработаны только элементы <author>:

|  |
| --- |
| <rule>  <target-element type=" staff"/>  <div>  <select-elements>  <target-element type = "author"/> </select-elements>  </div>  </rule> |

**ПРАВИЛА СТИЛЕЙ**

Для определения правила стилевого оформления необходимо воспользоваться элементом **<style-rule>**;, который используется точно также, как и <**rule**>, но инструкции, содержащиеся в нем, никак не влияют на структуру выходного документа. Поэтому все команды внутри этого правила должны описываться в рамках элемента <**apply**>.

*Пример:*

|  |
| --- |
| <style-rule>  <target-element type ="flower"/>  <apply color ="red"/>  </style-rule>  <rule> <target-element type="flower"/>  <div font-style="italic";>  <children/>  </div>  </rule> |

*Если бы мы не определили правила <rule>, то в выходной документ не было бы помещено никакой информации, т.к. элемент <style-rule> только определяет параметры стилевого оформления, не предпринимая никаких других действий.*

# 98. XLink & XPointer. Семантика обработки.

Иногда возникает ситуация, когда объект связан с другим объектом, но эта связь не является иерархической

**XLink** позволяет создавать в **XML**-документе ссылки и задавать семантику его обработки (кроме того, поддерживает встроенные ссылки)

**XPointer** позволяет ссылаться на фрагменты документов с гораздо большей степенью детализации, чем была доступна в **HTML**

# 99. Разработка отраслевых схем XML. (нет в презентациях и лекциях)

# 100. XML-компоненты в .NET. (нет в презентациях и лекциях)

# 101. Описание XmlReader и XmlWriter.

Класс чтения **XML** предоставляет интерфейс программирования, используемый вызывающими программами для подключения к **XML**-документам и чтения требуемых данных.

С точки зрения классов чтения, **XML**-документ — это не размеченный текстовый файл, а сериализованный набор узлов.

***Свойства классов, производных от XmlReader***

|  |  |
| --- | --- |
| Свойство | Описание |
| AttributeCount | Сообщает число атрибутов для текущего узла |
| BaseURI | Сообщает базовый URL текущего узла |
| CanResolveEntity | Сообщает значение, указывающее, может ли класс чтения разрешать сущности |
| Depth | Сообщает глубину текущего узла в XML-документе |
| EOF | Указывает, достиг ли класс чтения конца потока |
| HasAttributes | Указывает, имеются ли у текущего узла атрибуты |
| HasValue | Указывает, может ли быть у текущего узла значение |
| IsDefault | Указывает, является ли текущий узел атрибутом, сгенерированным на основе значения по умолчанию, определенному в DTD или схеме |
| IsEmptyElement | Указывает, является ли текущий элемент пустым — без атрибутов и значения |
| Item | Индексное свойство, возвращающее значение заданного атрибута |
| LocalName | Сообщает имя текущего узла, из которого удалены префиксы |
| Name | Сообщает полностью определенное имя текущего узла |
| NamespaceURI | Сообщает URI пространства имен текущего узла; имеет смысл только для узлов Element и Attribute |
| NameTable | Сообщает имя объекта таблицы имен, связанного с классом чтения |
| NodeType | Сообщает тип текущего узла |
| Prefix | Сообщает префикс пространства имен, связанный с текущим узлом |
| QuoteChar | Сообщает, какой символ обрамляет значения атрибутов |
| ReadState | Сообщает состояние класса чтения; имеет перечислимый тип ReadState |
| Value | Сообщает текстовое значение текущего узла |
| XmlLang | Указывает область xml:lang, к которой относится текущий узел |
| XmlSpace | Указывает текущую область xml:space; имеет перечислимый тип XmlSpace (Default, None или Preserve) |

***Методы классов, производных от XmlReader***

|  |  |
| --- | --- |
| Метод | Описание |
| Close | Закрывает класс чтения и присваивает переменной, описывающей внутреннее состояние, значение Closed |
| GetAttribute | Получает значение заданного атрибута; позволяет обращаться к атрибуту по индексу, локальному имени или полному (с путем) имени |
| IsStartElement | Указывает, является ли текущий узел контента начальным тэгом |
| LookupNamespace | Возвращает URI пространства имен, которому сопоставляется данный префикс |
| MoveToAttribute | Перемещает указатель на заданный атрибут; атрибут можно указывать с помощью индекса, локального имени или полного (с путем) имени |
| MoveToContent | Перемещает указатель на следующий узел контента или в конец файла; метод сразу же возвращает управление, если текущий узел уже является узлом контента, на*Пример* текстом, CDATA, Element, EndElement, EntityReference или EndEntity |
| MoveToElement | Перемещает указатель обратно на узел элемента, к которому относится текущий узел атрибута; имеет смысл, только когда текущим узлом является атрибут |
| MoveToFirstAttribute | Перемещает указатель на первый атрибут текущего узла Element |
| MoveToNextAttribute | Перемещает указатель на следующий атрибут текущего узла Element |
| Read | Читает текущий узел и перемещает указатель вперед |
| ReadAttributeValue | Анализирует значение атрибута и заносит результаты в один или несколько узлов Text или EntityReference |
| ReadElementString | Читает и возвращает текст из чисто текстового элемента |
| ReadEndElement | Проверяет, является ли текущий узел контента замыкающим тэгом, и перемещает указатель на следующий узел; генерирует исключение, если узел не является замыкающим тэгом |
| ReadInnerXml | Читает и возвращает содержимое (контент) текущего узла (без самого узла), в том числе разметку |
| ReadOuterXml | Читает и возвращает все содержимое текущего узла (вместе с самим узлом), в том числе разметку |
| ReadStartElement | Проверяет, является ли текущий узел элементом, и перемещает указатель на следующий узел; генерирует исключение, если узел не является начальным тэгом |
| ReadString | Читает содержимое элемента или текстового узла как строку; метод объединяет весь текст, встречающийся до ближайшего элемента разметки; в случае узлов атрибутов этот метод считывает значение атрибута |
| ResolveEntity | Раскрывает и разрешает текущий узел ссылки на сущность |
| Skip | Пропускает дочерние элементы текущего узла |

**КЛАСС XMLTEXTWRITER**

Класс записи XML (XML writer) выводит XML-данные в потоки или файлы с перемещением только вперед.  XmlWriter — абстрактный базовый класс для всех классов записи XML.

***Состояния класса чтения XML***

|  |  |
| --- | --- |
| Состояние | Описание |
| Attribute | Выполняется запись атрибута |
| Closed | Вызван метод Close, и класс записи больше не позволяет выполнять операции записи |
| Content | Записывается содержимое узла |
| Element | Записывается начальный тэг элемента |
| Prolog | Записывается пролог синтаксически корректного XML-документа |
| Start | Класс записи находится в начальном состоянии и ждет, когда начнется запись |