**Строка(String) –** Кодируются в UTF-16,должны заключаться в кавычки.

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет ${name}`. Позволяют встраивать результат выражения в строку. Такие строки могут занимать несколько строк.

Свойство **length** содержит длину строки.

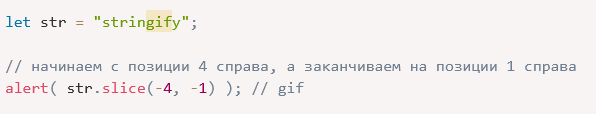
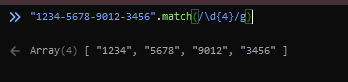
Получить символ по позиции можно с помощью:

* квадратных скобок []. Вернет undefind если на указанной позиции нет символа.
* **charAt(pos).**  Если на позиции нет символа, вернет пустую строку.
* **at.** Если на позиции нет символа, вернет пустую undefind. Позволяет указывать отрицательные индексы.

Перебрать

**Строки неизменяемы.**

Методы

* **St.toLowerCase()** – привести строку к нижнему регистру
* **St.toUpperCase()** – привести строку к верхнему регистру
* **st.indexOf(substr, pos)** – найти индекс подстроки substr начиная с позиции pos. -1 при отсутствии.
* **Str**.**lastIndexOf(substr, position) –** найти индекс последнего вхождения подстроки в строку.
* **str.includes(substr, pos) –** проверить, есть ли в строке подстрока.
* **str.startsWith(substr) –** проверить начинается ли строка определенной строкой.
* **str.endsWith(substr) -** проверить заканчивается ли строка определенной строкой.
* **str.slice(start [, end]) –** вернуть подстроку от start до (не включая) end. Для start и end можно задавать отрицательные значения.Тогда отсчет идет с конца строки  
  
* **str.substring(start [, end]) –** то же, что и slice, но можно задать start > end, тогда сработает так, если бы аргументы были поменяны местами. Не поддерживает отрицательные значения.
* **str.substr(start [, length]) -** Возвращает часть строки от start длины length. start может быть отрицательным.
* **str.localeCompare(str2) -** возвращает число, которое показывает, какая строка больше в соответствии с правилами языка. Имеет 2 доп. аргумента. Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".
* **str.codePointAt(pos) –** возвращает код символа, находящегося на позиции pos.
* **String.fromCodePoint(code) –** создает символ по его коду.
* **str.trim() –** убирает пробелы в начале и конце строки.
* **str.trimEnd(**
* **str.repeat(n) –** повторяет строку n раз.
* **str.split(‘x’) –** разбивает строку по указанному символу.
* **str.concat(str2, …)** – конкатенация строк
* **str.match(regexp) -** возвращает получившиеся совпадения при сопоставлении строки с регулярным выражением.  
  
* **str.matchAll(regexp) -** возвращает итератор по всем результатам при сопоставлении строки с регулярным выражением.
* **str.replace(regexp|substr, newSubStr|function[, flags]) –** вернуть новую строку, в которой первое выражение будет заменено на второе. Первым параметром может быть строка или регулярка. При указании строки заменено будет только первое вхождение. Второй параметр – строка для замены, либо функция, возвращающая строку для замены.  
  

**NUMBER**

**Число (number)** – представляет как целочисленные значений, так и числа с плавающей точкой.

Имеются “специальные числовые значения”:

* **Infinity** – математическая бесконечность, больше любого числа. Его можно получить в результате деления на 0, либо задать явно.
* **-Infinity** – отрицательная бесконечность.
* **NaN (Not a number)** – результат неправильной или неопределенной математической операции. Любая математическая операция с NaN возвращает NaN. NaN не равен ничему, даже самому себе. Можно проверить с помощью функции **isNan(value)**.

Функция **isFinite(value)** возвращает true, если число не является NaN/Infinity/-Infinity.

Методы **Number.isNaN** и **Number.isFinite** – это более «строгие» версии функций isNaN и isFinite. Они не выполняют приведение типов, и сразу возвращают false если не number.

Могут быть **записаны в разных системах счисления**

* Шеснадцатиричная 0xff
* Восьмеричная 0o377
* Двоичная 0b11111111

Через **метод toString(base)** можно представить число в виде строки в системе счисления от 2 до 36.

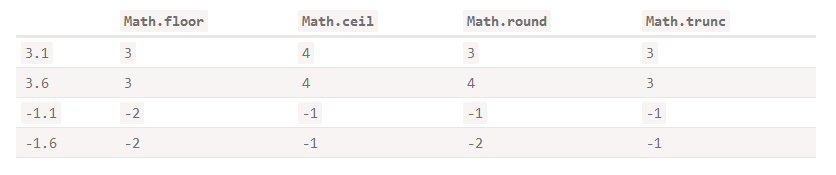
Числа могут быть представлены в экспоненциальной форме





**Округление:**

* **Math.floor** – округление в меньшую сторону
* **Math.ceil** – округление в большую сторону
* **Math.round** – округление до ближайшего целого
* **Math.trunc** – отбрасывание дробной части
* **num.toFixed(n)** – округляет число до n знаков после запятой и возвращает строковое представление результата.
* **num.toPrecision([precision])** – возвращает строку, представляющую число в записи с фиксированной запятой или в экспоненциальной записи, округленное до precision значащих цифр. если заданного количества разрядов недостаточно для точного отображения целой части числа, значение может быть возвращено в экспоненциальной.  
  



Чтобы вызвать метод непосредственно на литерале числа, нужно использовать 2 точки, так как js думает, что после первой точки начинается десятичная часть числа. Либо можно обернуть число в круглые скобки.

12..toString(16)

(12).toString(16)

Для явного преобразования можно использовать унарный плюс, либо Number(). Но они будут возвращать NaN если строка не является в точности числом.

Преобразование значений с помощью функции **Number()** осуществляется по следующим правилам:

* Числа возвращаются без каких либо изменений.
* **true** преобразуется в **1**.
* **false** преобразуется в **0**.
* Значение **null** преобразуется в **0**.
* Значение **undefined** преобразуется в **NaN**.

Преобразование строк также осуществляется по своим правилам:

* Обрезатся пробелы, **\n**,  **\t**, **\v** и др., которые находятся в начале или в конце строки.
* Строка может содержать унарный **-** или **+** либо без знака. Начальные нули игнорируются, например **"-007"** преобразуется в -**7**.
* Если строка представляет собой число в шестнадцатеричном формате, она преобразуется в соответствующее целое десятичное число. Например +”0xaff” => 2815.
* Пустая строка преобразуется в **0** (только в мат. операциях, через parse она становится NaN).
* Если строка содержит что-то отличное от предыдущих вариантов, например инкремент (декремент), она преобразуется в **NaN**.

Неявное приведение значения к числовому типу в JavaScript применяется, чаще чем преобразование в строку или в логическое значение.

Функции parseInt и parseFloat могут распарсить такое: “12.11dasdad” -> 12.11; Пробелы в начале строке игнорируются.

Такое не могут “dasd10” -> NaN. Они так же позволяют вторым параметром указать систему счисления.

Объект **Math** содержит различные математические функции и константы.

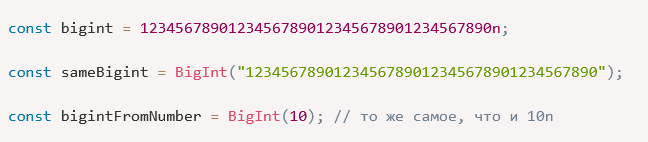
**Math.random()** - Возвращает псевдослучайное число в диапазоне от 0 (включительно) до 1 (но не включая 1).

**Math.max(a, b, c...) / Math.min(a, b, c...)** - Возвращает наибольшее/наименьшее число из перечисленных аргументов.

**Math.pow(n, power)** - Возвращает число n, возведённое в степень power

**BigInt** – был добавлен, чтобы дать возможность работать с целыми числами произвольной длины. (number ограничен значением (253-1)).

Чтобы создать значение типа BigInt, необходимо добавить **n** в конец числового литерала или вызвать функцию **BigInt**, которая создаст число типа BigInt из переданного аргумента. Аргументом может быть число, строка и др.

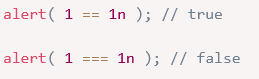


Свойство Number.MAX\_VALUE представляет максимальное числовое значение, представимое в JavaScript. Значение больше представляются как бесконечность.

maxSaveInteger представляет максимальное безопасное целочисленное значение в JavaScript (2^53 - 1). Под безопасностью в этом контексте понимается способность точно представлять целые числа и правильно их сравнивать.

Все операции с числами типа bigint возвращают bigint. В математических операциях мы не можем смешивать bigint и обычные числа.

*Обычные* и *bigint* числа принадлежат к разным типам, они могут быть равны только при нестрогом сравнении ==.



**DATE**

**Date –** объект, который используется для работы с датой и временем.

**new Date() –** создает объект с текущей датой и временем.

**new Date(milliseconds) –** создает объект на основе количества миллисекунд прошедших с первого января 1970 года. Передав отрицательное можно получить дату до 1970.

**new Date(datestring) –** почти то же что и Date.parse.

**new Date(year, month, date, hours, minutes, seconds, ms) –** создать объект по компонентам. Обязательными являются только год и месяц. date по умолчанию 1, остальные 0. Отсчет месяцев начинается с нуля.

**Date.UTC(year, month, date, hours, minutes, seconds, ms) –** параметры аналогичны предыдущему, но возвращает количество миллисекунд с января 1970 по UTC.

**Date.parse(str) –** разбирает строку в формате , где T – разделитель между между датой и временем, Z-временная зона.



**Date.now() –** возвращает текущую дату в виде миллисекунд. Аналогичен вызову +new Date() но не создает промежуточный объект даты, поэтому работает быстрее.

Для доступа к компонентам даты используются следующие методы:

* **getFullYear() –** получить год (из 4 цифр).
* **getMonth() –** получить месяц, от 0 до 11.
* **getDate() –** получить число месяца, от 1 до 31.
* **getHours() –** получить часы.
* **getMinutes() –** получить минуты
* **getSeconds() –** получить секунды
* **getMilliseconds() –** получить миллисекунды.
* **getDay() –** получить день недели. **от 0(воскресенье) до 6(суббота)**
* **getUTCFullYear(), getUTCMonth() и т. д. –** аналоги вышеуказанных методов, возвращающие время по нулевому мередиану.
* **getTime() –** число миллисекунд прошедших с 1 января 1970 года GMT+0.
* **getTimezoneOffset() –** возвращает разницу между местным и UTC временем в минутах.

Установить значения компонентов позволяют следующие методы:

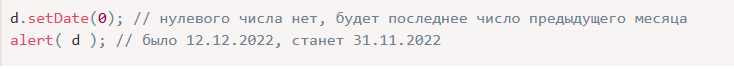
* **setFullYear(year [, month, date])**
* **setMonth(month [, date])**
* **setDate(date)**
* **setHours(hour [, min, sec, ms])**
* **setMinutes(min [, sec, ms])**
* **setSeconds(sec [, ms])**
* **setMilliseconds(ms)**
* **setTime(milliseconds)** (устанавливает всю дату по миллисекундам с 01.01.1970 UTC)

Все они, кроме **setTime(),** обладают также **UTC-вариантом**, например: **setUTCHours().**

Объект Date обладает **автоисправлением**. Неправильные компоненты даты автоматически распределяются по остальным.



Это удобно для нахождения даты, отдаленной на нужный промежуток времени. Можно устанавливать и нулевые, и отрицательные компоненты.





Когда Date используется в числовом контексте, он преобразуется в количество миллисекунд.

Важный побочный эффект: **даты можно вычитать**, результат вычитания объектов Date – их **временная разница, в миллисекундах**.

Вывести отформатированную дату можно методами: **date.toLocaleString(локаль, опции)**



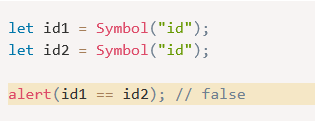
**toString() –** вернуть дату целиком**, toDateString() –** только дату**, toTimeString() –** только время**.** Эти методы возвращают стандартное строчное представление, не заданное жестко в стандарте а зависящее от браузера.

**toISOString()** - Возвращает дату в формате ISO.

**SYMBOL**

**Symbol** представляет собой уникальный идентификатор. Они могут иметь описание (имя). Создаются с помощью функции Symbol():

Символы гарантированно уникальны. Даже символы с одинаковым описанием все равно будут разными. Описание это просто метка, которая ни на что не влияет.



Символы не преобразуются автоматически в строки (Например при вызове alert будет ошибка). Сделано в целях защиты , чтобы строки и символы не преобразовывались друг в друга там, где мы этого не ожидаем.

Символы позволяют создавать “скрытые” свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их в другом коде, в котором нет этого символа.

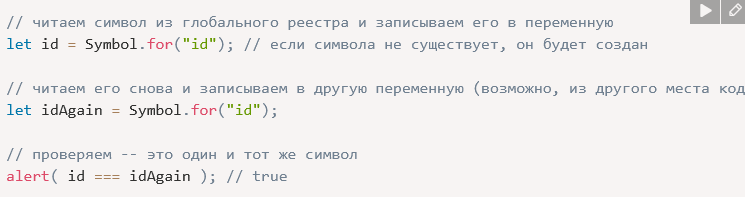
Свойства, чьи ключи символы, не перебираются через for..in. А вот **Object.assign**, копирует и строковые и символьные свойства.

В литеральном объекте нужно заключать символ в скобки, чтобы js не принимал его за строку.



Если же мы хотим, чтобы символы с одинаковыми именами были одной сущностью, можно использовать **глобальный реестр символов**. Мы можем создавать в нем символы, и обращаться к ним позже, при этом нам гарантированно будет возвращаться один и тот же символ.

Для **чтения** (или, при отсутствии, **создания**) символа из реестра используется вызов **Symbol.for(key)**.



**Symbol.keyFor(sym)** – принимает глобальный символ, и возвращает его имя. Если символ не содержится в глобальном реестре, вернется undefined.

Для любых символов доступно свойство **description**, который хранит имя символа.

Существует множество системных символов:

* **Symbol.hasInstance** - Метод, определяющий, распознает ли конструктор некоторый объект как свой экземпляр. Используется оператором **instanceof**.
* Symbol.isConcatSpreadable
* **Symbol.iterator –** возвращает итератор по умолчанию для объекта. Используется конструкцией **for .. of**.
* **Symbol.toPrimitive** - Метод, преобразующий объект в примитив (примитивное значение).

Технически символы не полностью скрыты.

* **Object.getOwnPropertySymbols(obj) –** позволяет получить все свойства объекта с ключами-символами.
* **Reflect.ownKeys(obj) –** возвращает все ключи объекта, включая символьные.

**OBJECT**

**object –** объект. Состоит из пар ключ-значение. Ключ (имя свойства) – строка, значение что угодно.

Имя свойства может состоять из нескольких слов через пробел, но тогда оно должно быть заключено в кавычки.

Пустой объект можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

Можем сразу создать объект из нескольких свойств:

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

};

**Конструктор Object** создаёт объект-обёртку для переданного значения. Если значением является null или undefined, создаёт и возвращает пустой объект, в противном случае возвращает объект такого типа, который соответствует переданному значению. Если значение уже является объектом, конструктор вернёт это значение.

Для обращения к свойствам используется запись через точку, либо через квадратные скобки и имя свойства в кавычках.

alert( user.name ); // John

alert( user.age ); // 30

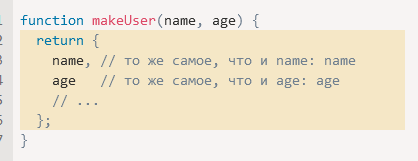
alert(user["likes birds"]); // true

Для удаления свойства используется оператор **delete**

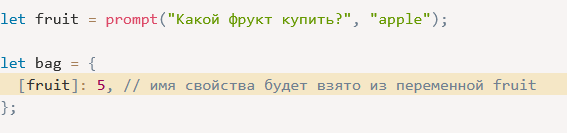
delete user.age;

**свойства упорядочены** особым образом: свойства **с целочисленными ключами сортируются по возрастанию**, **остальные** располагаются **в порядке создания.**

Если хотим значение переменной установить как свойство объекта, и использовать то же имя что и переменная, можно использовать короткую запись



С помощью квадратных скобок в литеральной нотации мы можем создать **вычисляемое свойство**:



Имя свойства может содержать зарезервированные слова (for, let, return).

При обращении к свойству, которого нет в объекте возвращается underfind. Чтобы проверить, есть ли у объекта такое свойство можно использовать оператор **in**:

let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age существует

alert( "blabla" in user ); // false, user.blabla не существует

Мы можем менять объект, объявленый с *const*(добавлять, удалять поля и менять их значение). Мы не можем изменить только саму переменную.

Для перебора всех свойств объекта и его цепочки прототипа используется цикл **for..in**:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

**Object.keys(x) –** возвращает массив с именами свойств объекта. (Не ищет по прототипу).

Для проверки типа используется оператор **typeof**. Возвращает строку с именем типа.

Оператор **instanceof** проверяет, принадлежит ли объект к определённому классу. Другими словами, object instanceof constructor проверяет, присутствует ли объект constructor.prototype в цепочке прототипов object.

Объект хранится и копируется по ссылке.

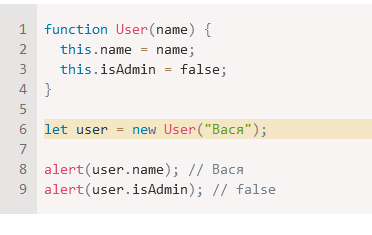
Для **клонирования** придется создать новый объект и переписать в него свойства. Либо



Он копирует все свойства объектов src в объект dest.

Синтаксис {…} позволяет создавать один объект. Создавать множество однотипных объектов можно с помощью **функции-конструктора** и **оператора new**.

Это обычные функции, и любую функцию можно вызвать через оператор new. Но их обычно называют с большой буквы.



Конструктор работает следующим образом:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется тело функции. Обычно оно модифицирует this, добавляя туда новые свойства.
3. Возвращается значение this.

Технически, любая функция (кроме стрелочных) может использоваться в качестве конструктора.

Используя специальное свойство **new.target** внутри функции, мы можем проверить, **вызвана ли функция при помощи оператора new или без него**.

В случае **обычного вызова** функции new.target будет **undefined**. Если же она была вызвана при помощи new, new.target будет **равен самой функции**.

Если функция **содержит** оператор **return**, то при вызове с new:

* При вызове return с объектом, вместо this вернётся объект.
* При вызове return с примитивным значением, оно игнорируется.

**Object.create(proto [, descriptors]) –** создает пустой объект со свойством [[Prototype]], указанным как proto (может быть null), и необязательными дескрипторами свойств.

**При использовании объектов с операторами, они преобразуются в примитивы.**

* В **логическом** контексте все объекты являются **true**.

Преобразование Объекта к примитивам осуществляется с помощью хинтов:

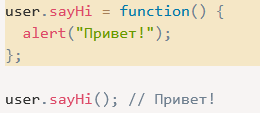
* **“string”** - для преобразования объекта к строке, когда мы выполняем операцию, которая ожидает строку
* **“number”** – для преобразования к числу в случае математических операций, и большинстве математических функций, операторами сравнения.
* **“default”** – происходит редко, когда оператор не уверен какой тип ожидать. Например бинарный плюс может работать как со строками (конкатенация), так и с числами (сложение). Поэтому при сложении + использует хинт “default”. Так же с операторам ==.

Все встроенные объекты, за исключением Date, реализуют "default" преобразование тем же способом, что и "number".

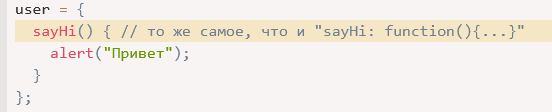
Для преобразования используется метод **Symbol.toPrimitive**, он принимает название хинта и выполняет соответствующее преобразование. Он обрабатывает все случаи преобразования.

Если нет Symbol.toPrimitive то JS ищет методы **toString** для строковых преобразований, и **valueOf** для остальных. Если valueOf нет, либо он не возвращает примитив, то везде используется toString.

Примитивы имеют **объекты-обертки.** String, Number, Boolean, Symbol и BigInt (null и undefind не имеют). Они могут автоматически оборачиваться в момент вызова метода на примитиве. Затем вызывается метод и обертка удаляется.

**Методы** – свойства объекта, являющиеся функциями. 

Есть **сокращенная запись** для метода:



Методы:

* **Object.assign** – копирует свойства всех указанных объектов в указанный объект и возвращает этот объект. Он не делает глубокое клонирование.
* **Create** – создает новый объект с указанным прототипом и свойствами.
* **defineProperties** – добавляет к объекту новое свойство (enumerable…). Изменить флаги свойства.
* **Entries/fromEntries** – создает вложенный массив пар ключ-значение объекта (наоборот).
* **Freeze** – Запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.
* **Seal** – Запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств..
* **preventExtensions** – Запрещает добавлять новые свойства в объект.
* **hasOwnProperty** – возвращает логическое значение, указывающее, содержит ли объект указанное свойство (Не проверяет прототипы, то есть именно в этом объекте).
* **Is** – являются ли 2 значения одинаковыми. Сравнивает даже null, NaN и т.д.
* **Values** – возвращает массив значений перечисляемых свойств объекта объекта, не включая цепочку прототипов.
* **Keys** – возвращает массив из собственных перечисляемых свойств переданного объекта.
* **ValueOf** – возвращает примитивное значение указанного объекта.
* **setPropertyOf** – устанавливает прототип.
* **getPropertyOf** – возвращает прототип.
* **Prototype** – позволяет добавлять свойства к объекту.

**Свойства объектов**

Свойства имеют 3 специальных атрибута:

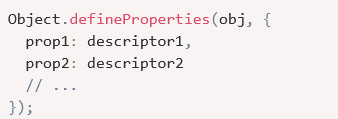
* **writable** – если true, свойство можно изменить, иначе оно только для чтения.
* **enumerable** – если true, свойство перечисляется в циклах (for…in, Object.keys()), в противном случае циклы его игнорируют.
* **configurable** – если false нельзя удалять свойство и менять его флаги. При этом можно менять его значение. Сделать его обратно true нельзя.

**Object.getOwnPropertyDescriptor(obj, propertyName)** позволяет получить полную информацию о свойстве.



**Object.defineProperty(obj, propertyName, descriptor) –** Изменить флаги свойства. Если свойство существует, его флаги обновляться. В противном случае метод создаст новое свойство с указанным значением и флагами. Если какой-то флаг не указан явно, ему присваивается значение false.

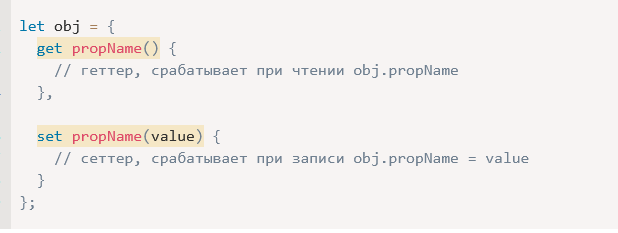
**Object.defineProperties(obj, descriptors) –** позволяет определить множество свойств сразу:



**Object.getOwnPropertyDescriptors(obj) –** получить все дескрипторы свойств сразу.

Свойству можно назначить **геттеры и сеттеры**.

Они вызываются не как функции, а как обычные свойства. Геттер – при получении, сеттер – при записи.

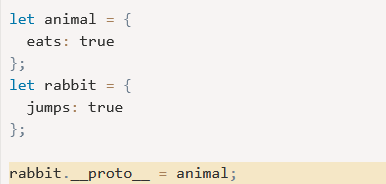


**Прототипы**

**Prototype** – специальное скрытое свойство, которое содержит другой объект, либо null. Это объект, на основе которого создан другой объект.

Если свойство отсутствует в объекте, js ищет его в прототипе. Это называется **“**Прототипным наследованием**”**. При этом **запись свойств происходит непосредственно в объект**, а не в прототип. **Исключение** составляют **свойства-аксессоры**. Но this все равно содержит ссылка на вызываемый объект.

Задать его можно **с помощью геттера-сеттера \_\_proto\_\_ для внутреннего свойства [[Prototype]]:**

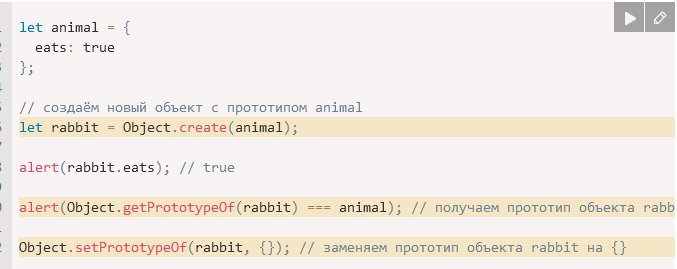


\_\_proto\_\_ может быть только объектом, или null**.** Другие значения игнорируются. Нельзя зацикливать прототипы.

По умолчанию **ptototype** – это объект с единственным свойством constructor, которое ссылается на функцию конструктор. Присвоив конструктору свойство prototype конструктор устанавливает его для новых объектов, соpданных с помощью new.

Современные же методы это:

* **Object.create(proto, [descriptors])** – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.
* **Object.getPrototypeOf(obj)** – возвращает свойство [[Prototype]] объекта obj.
* **Object.setPrototypeOf(obj, proto)** – устанавливает свойство [[Prototype]] объекта obj как proto.



Object.create можно использовать **для** «продвинутого» **клонирования объекта** (с поверхностным копированием свойств), вместо for in:

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

Движок JS оптимизирован под изменение прототипов только при создании объекта. Последующие изменения могут негативно сказаться на производительности.

**МАССИВЫ**

**Массив –** упорядоченная коллекция данных, в которой каждый элемент определяется своим индексом. В одном массиве могут хранится элементы разных типов.

Массив – это объект, где в качестве ключа выступает числовое значение.

Движок js старается хранить элементы массива в непрерывной области памяти, один за другим. Это ускоряет работу массивов. Поэтому желательно создавать массивы однотипных элементов, не создавать «дыр» и не добавлять нечисловых свойств.

Объявление массива:

* let arr = new Array(1, 2, 3);
* let arr = new Array(5); - пустой массив длины 5
* let arr = [1, 2, 3];

Получить элемент можно по его индексу (нельзя использовать отрицательные):



Таким же образом можно заменять и даже добавлять элементы.

**Свойства:**

* **length –** размер массива (число элементов). Его можно перезаписать вручную.

**Методы:**

* **arr.at(index)** – новый метод, который поддерживает отрицательные индексы.
* **arr.push(val…)**  - добавить элемент в конец.
* **arr.pop() –** удаляет последний элемент и возвращает его.
* **arr.shift() –** удаляет первый элемент массива, возвращает его, и сдвигает остальные элементы на одну ячейку влево.
* **arr.unshift(val…) –** добавляет элемент в начало, остальные элементы смещаются вправо.
* **arr.toString() –** возвращает строку со списком элементов, разделенных запятыми. Пустой массив возвращает пустую строку. Этот метод используется когда нужно привести массив к примитиву.
* **arr.splice(index[, deleteCount, elem1, ..., elemN]) –** начиная с позиции index, удаляет deleteCount элементов и вставляет elem1, …, elemN на их место. Возвращает массив из удаленных элементов.
* **arr.slice([start], [end]) –** возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Вызвав его без аргументов скопируется весь массив.
* **arr.concat(arg1, arg2...)** – создает новый массив, в который добавляет дргуие элементы. Добавлять можно как обычные элементы, так и массивы. Из массива копируются все элементы. Обычный объект может копироваться как массив (поэлементно) если у него есть свойство **Symbol.isConcatSpreadable**.
* **Arr.forEach(func) –** позволяет перебрать все элементы и выполнить над ними указанное действие.
* **arr.indexOf(item, from) -** ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* **arr.lastIndexOf(item, from) –** то же самое, но ищет последнее вхождение.
* **arr.includes(item, from) –** ищет item, начиная с индекса from, и возвращает true, если поиск успешен.
* **arr.find(func) –** найти первый элемент, удовлетворяющий условию. Если не найдено, возвращает undefind/
* **arr.findIndex(func) –** как и find, только возвращает индекс. Если не найден -1.
* **arr.filter(func) –** возвращает массив элементов, удовлетворяющих условию.
* **arr.map(func) –** вызывает функцию для каждого элемента и возвращает массив результатов выполнения этой функции.
* **arr.sort(func) –** сортирует массив. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как сортируется сам arr. По умолчанию элементы сортируются как строки.
* **arr.reverse(func) –** меняет порядок элементов в массиве на обратный.
* **str.split(delim) –** разбивает строку на массив по заданному разделителю
* **arr.join(glue) –** создает строку из элементов массива вставляя между ними указанный разделитель.
* **arr.reverse() –** меняет порядок элементов на обратный.
* **arr.join(glue) –** создает строку из элементов массива с разделителем glue.
* **arr.reduce() –** используется для вычисления единого значения на основе массива. Передаваемая функция принимает 2 параметра, аккумулятор и очередной элемент массива.  
    
  Если не указать начальное значения аккумулятора, то в качестве него будет взят первый элемент массива. Начальное значение следует указывать если массив может быть пустой, чтобы не произошло ошибки.
* **arr.some(fn) –** возвращает true если хотя бы один из элементов массива соответствует условию.
* **arr.every(fn) –** возвращает true если все элементы массива соответствуют условию.
* **arr.flatMap(fn) –** аналогичен map, но одно значение может быть преобразовано в несколько.
* **arr.fill(value, start, end) –** заполняет массив повторяющимися value, начиная с индекса start и заканчивая end.
* **arr.copyWithin(target, start, end) –** копирует свои элементы, начиная со start и заканчивая end в собственную позицию **target**.
* **Array.isArray(arr) –** проверить, является ли переданное значение массивом.
* **Array.from –** создает новый массив из массивоподобного или итерируемого объекта.

Для перебора массива можно использовать обычный for, либо специальный вариант – **for..of**.

**for..of** в отличии от for..in выполняет перебор только числовых свойств.

**SET**

**Set -** это множество уникальных значений без ключей. Создать Set можно через new Set() и если в него будет передан какой-то итерируемый объект (обычно это массив), то копирует его значение в новый Set.

* Add(value) – добавляет новое значение, но если оно уже есть, то ничего не делает. Возвращает тот же объект, к которому был применен метод.
* Delete(value) – удаляет значение и возвращает true/false в зависимости от успешности выполнения.
* Has(value) – проверяет на наличие значения.
* Clear() – очищает от всех элементов.
* Size – возвращает текущее количество элементов.

Перебрать можно с помощью for..of, forEach.

Set имеет все те же встроенные методы, что и Map.

WeakSet, также как и WeakMap является дополнительным хранилищем данных и имеет те же ограничения.

**MAP**

Это коллекция ключ-значение, при этом ключи могут быть любого типа. Создать новую коллекцию можно через new Map();

* Set(key, value) – записывает по key значение value.
* Get(key) – возвращает значение по ключу или undefined, если ключа нет.
* Has(key) – проверяет на наличие ключа.
* Delete(key) – удаляет элемент по ключу.
* Clear() – очищает коллекцию от всех элементов.
* Size – возвращает текущее количество элементов.

Map может использовать объекты в качестве ключей. Также при сравнении ключей, коллекция использует алгоритм, похожий на ===, но NaN = NaN.

Для перебора Map используются 3 метода:

* Keys() – возвращает итерируемый объект по ключам.
* Values() – возвращает итерируемый объект по значениям.
* Entries() - возвращает итерируемый объект по парам ключ-значение (используется по умолчанию в for..of).

В отличии от object, в map перебор происходит в том же порядке, в каком происходило добавление элементов. Также map имеет встроенный метод forEach. https://learn.javascript.ru/array-methods#perebor-foreach

Получение map из object:

let map = new Map(Object.entries(obj));

Получение object из map:

let obj = Object.fromEntries(map.entries());

В WeakMap ключи должны быть объектами, а не примитивами. Если на такой ключ нет ссылок извне, то он удалится сборщиком мусора. Он не поддерживает keys(), values(), entries().

В основном, WeakMap используется как дополнительное хранилище данных или для кэширования.

**ЗАМЫКАНИЯ. ЛЕКСИЧЕСКОЕ ОКРУЖЕНИЕ**

У каждой выполняемой функции, блока кода и скрипта есть связанный с ним объект, называемый **лексическим окружением** (**LexicalEnviroment**).

Оно состоит из двух частей:

* 1. **Enviroment Record -**  объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).
  2. **Ссылка на внешнее лексическое окружение.**

"Переменная" – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».

Вручную мы не можем изменять этот объект с помощью кода. Этим занимается сам движок js, а также оптимизацией этого объекта (удалением неиспользуемых переменных и т.п.)

**Глобальное лексическое окружение –** связанно со всем скриптом. Ссылка на внешнее окружение у него null.

Переменные попадают в лексическое окружение, когда выполнение скрипта доходит до их инициализации. Функции объявленные с помощью function declaration попадают в лексическое окружение сразу, после его создания, независимо от того где они объявлены (поэтому мы можем вызывать функцию до ее определения).

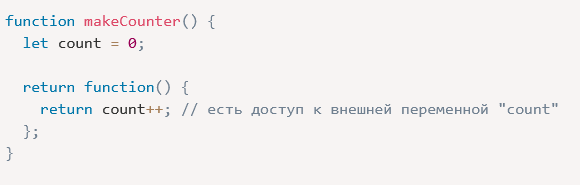
**При запуске функции** для нее автоматически **создается лексическое окружение**, для хранения локальных переменных и параметров вызова.

**Поиск переменной** поочередно выполняется **сначала во внутреннем, потом во внешнем** окружении и т.д. до глобального.

Каждый раз **при вызове функции создается новое лексическое окружение**, со своими, специфичными для этого вызова переменными и параметрами.

**Вложенная функция** – функция, которая создается внутри другой функции. Она имеет доступ к лексическому окружению внешней.

Все функции при создании получают скрытое свойство **[[Environment]]**, которое ссылается на лексическое окружение места, где они были созданы. Таким образом при вызове функции и создании ее лексического окружения, ссылка на внешнее окружение берется из переменной [[Environment]].



В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать.

**Замыкание –** это функция, которая запоминает внешние переменные и может получать к ним доступ. В JS все функции изначально являются замыканиями (кроме созданных с помощью new Function())

В функциях созданных через new Function() внешнее окружение всегда является глобальным.

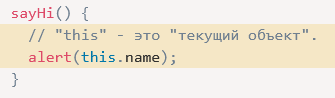
Лексическое окружение создается для любого блока кода: if, for, while, любой блок кода в {}.

У каждой итерации цикла свое лексическое окружение.

**КОНТЕКСТ THIS**

Для доступа к информации внутри объекта метод может использовать ключевое слово **this.**

Грубо говоря **this** – это объект перед точкой, который используется для вызова метода.



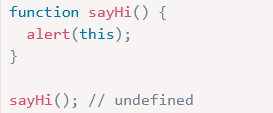
This **не является фиксированным.**  Значение this вычисляется во время выполнения кода, в зависимости от контекста.

Его можно использовать в любой функции, даже если это не метод объекта.

В строгом режиме в таком коде **this** будет являться **undefind.**

В нестрогом режиме значением **this** в таком случае будет **глобальный объект** (window в браузерe).

У **стрелочных функций нет this.** Он берется из внешней функции (в принципе как и любая переменная которой нет в локальном окружении). При вызове стрелочной функции можно сказать что контекст перебрасывается на ступеньку выше.



**func.call(context, …args)** – специальный встроенный метод функции, который позволяет вызывать функцию, явно устанавливая this.

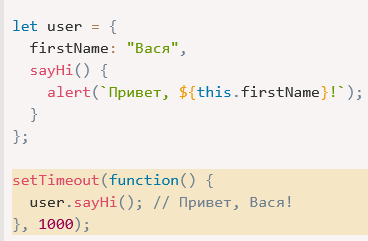
Он запускает функцию func, используя первый аргумент в качестве this, а последующие как ее аргументы.

**func.apply(context, args) –** аналогичен call, но принимает аргументы не по одному, а в виде псевдомассива.

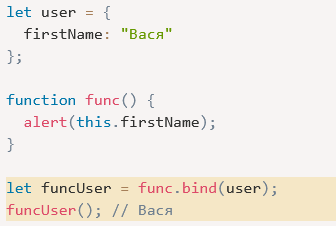
При передаче методов объекта в качестве колбэков, например для setTimeout, возникает известная проблема – **потеря this.**

Метод setTimeout в браузере имеет особенность: он устанавливает this=window для вызова функции.

**Решения:**

1. **Обернуть вызов в анонимную функцию, сделав замыкание**
2. **Привязать контекст с помощью bind**

**let boundFunc = func.bind(context)** – является особый экзотический объект, который вызывается как функция и прозрачно передает вызов в func, при этом устанавливая this=context. Вызов boundFunc подобен вызову func с фиксированным this.  
При этом this не изменится, даже если выполнит функцию с помощью call, и передать в качестве контекста другой объект. Такая привязка называется **жесткой**.



C помощью bind можно так же **привязать аргументы**.



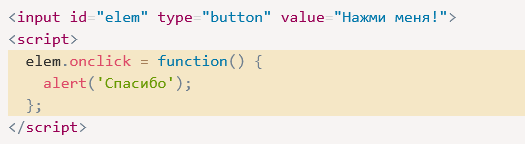
Если указать не все аргументы, то оставшиеся не будут привязаны.

**Назначение обработчиков**

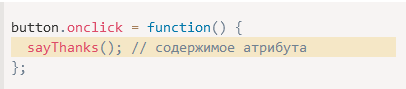
**прямо в разметке**, в атрибуте **on<событие>**:



**Свойство DOM-объекта, on<событие>**:



Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство.



Этот способ, по сути, аналогичен предыдущему. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Так как у Элемента DOM может быть только одно свойство с определенным именем, то **таким способом** **назначить более одного обработчика нельзя**.

**addEventListener** – позволяет назначать несколько обработчиков.



**event** – имя события, например “click”;

**handler –** ссылка на функцию обработчик;

**options** - Дополнительный объект со свойствами:

* **once**: если true, тогда обработчик будет автоматически удалён после выполнения.
* **capture**: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}. Если false (по умолчанию), событие будет поймано при всплытии. True – при погружении.
* **passive**: если true, то указывает, что обработчик никогда не вызовет preventDefault(). Есть некоторые события, как touchmove на мобильных устройствах, которое по умолчанию начинает прокрутку, но мы можем отменить это действие в обработчике. Браузер запустит прокрутку только после выполнения всех обработчиков, если никакой обработчик ее не отменит. Указав passive: true мы сообщим браузеру, что не собираемся отменять обработку по умолчанию, и браузер сразу начнет прокрутку.

Для удаления обработчика следует использовать **removeEventListener**



Для удаления нужно передать именно ту функцию, которая была назначена. Если функцию обработчик не сохранить, мы не сможем ее удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener.

Есть события, которые можно назначить только через addEventListener например DOMContentLoaded.

Внутри обработчика события this ссылается на элемент, на котором висит обработчик.

Обработчики вызываются в порядке их назначения.

**Объект-обработчик** также может быть назначен в качестве обработчика с помощью addEventListener. В этом случае, когда происходит событие, вызывается метод объекта **handleEvent**.

**Объект события –** объект, который содержит детали события. Браузер создает его, когда происходит событие, и передает его в качестве аргумента функции обработчику.

Свойства:

* event.type – тип события
* event.currentTarget – (= this) элемент до которого дошло всплытие, на нем сейчас выполняется обработчик.
* event.target – целевой элемент, на котором произошло событие.
* event.eventPhase – номер фазы, на которой поймано событие.
* event.clientX / event.clientY – координаты курсора в момент клика относительно окна, для событий мыши.

target – объект, на котором произошло событие

current-target – объект на который наложен обработчик

**3 фазы прохода события**

**Фаза погружения** – событие идет сверху вниз

**Фаза цели** – событие достигло целевого (исходного) элемента.

**Всплытие -** событие начинает всплывать.  
Когда на элементе происходит событие, обработчики сначала срабатывают на нем, потом на его родителе и так далее, вверх по цепочке предков до объекта document(иногда window). Всплывают не все события (focus).

Любой промежуточный обработчик может остановить всплытие, вызвав ***event.stopPropagation()***

Если у элемента есть несколько обработчиков, то даже при прекращении всплытия будут выполнены все обработчики на текущем элементе. Метод **stopImmediatePropagation()** предотвращает всплытие и останавливает обработку событий на текущем элементе.

**Делегирование событий**

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

В нем мы можем получить целевой элемент event.target, чтобы понять, на каком элементе произошло событие.