Javascript

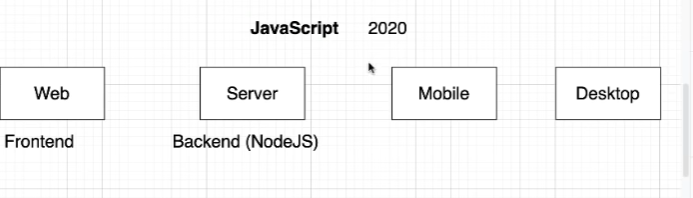
**Закончил на** [**Продвинутая работа с функциями**](https://learn.javascript.ru/advanced-functions)**, не начал**

# ВВедение

JavaScript – это нетипизированный язык, т. е. в нем не требуется определять типы переменных

* Изначально были только html and css, но они статические и хотели чтобы все двигалось, поэтому решили сделать 3й язык для динамики на пэйджах
* **ecmaScript** – этостандарт которая следит чтобы разные версии js yнормально работали в разных браузерах, это было потмоу что майкрософт сопротивлялся и говорил что у них все свое и не признавали js, но потом все таки пришлось. Так что **по сути это и есть js**
* **ES6** = ecmasctipt 2015 там появилось куча всего и с тех пор js стал оч популярным

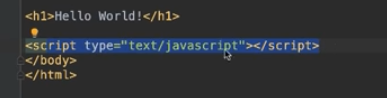
* **nodeJS** – в хроме есть движок транспиляции js, котороый называется chromewb8(или чето такое) его вытащили из хрома, написали над ним конструкции и сделали так кака бы серверный js и в итоге



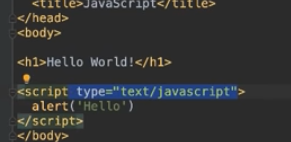
* js применяется как на фронте, так и **на бэке как nodejs**, потому что это производительная штука.
* с помощью js можно писать и на **мобилки**, например с помощью **react native**, это будет универсально и под андроид и под айос
* еще и десктоп можно,

Для подключения нескольких скриптов используйте несколько тегов:

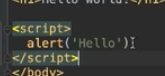
чтобы **соединить html и js** пишем в html тжг sctipt, лучше всегда в конце body



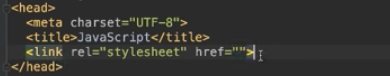
Говорим, что **в теге скрипт** мы будем хранить текст который написан на js



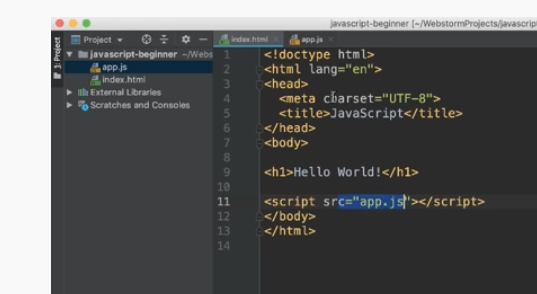
Но вообще **по умолчанию скрипт это и так js**, то можно не писать type.



* Чтобы **подключать css** в блоке можно юзаать



* Чтобы **подключить js** надо



## <Script>

### [Внешние скрипты](https://learn.javascript.ru/hello-world#vneshnie-skripty)

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл.

Файл скрипта можно подключить к HTML с помощью атрибута src:

<script src="/path/to/script.js"></script>

Здесь /path/to/script.js – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, src="script.js" будет означать, что файл "script.js" находится в текущей папке.

Можно указать и полный URL-адрес. Например:

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>

## Npx vs npm

Npm exists npx(Node Package Execute).

Npx - It is an npm package runner that can execute any package that you want from the npm registry **without even installing that package.** The npx is useful during a single time use package.

|  |  |
| --- | --- |
| npm | Npx |
| Если вы хотите запустить пакет через npm, вам необходимо указать этот пакет в своем package.json и установить его локально | Пакет может быть исполняемым без установки пакета, это средство запуска пакетов npm, поэтому, если какие-либо пакеты, которые еще не установлены, он будет установлен автоматически. |
| Npm - это инструмент, который используется для установки пакетов. | Npm - это инструмент, который используется для execute пакетов. |
| Пакеты, используемые npm, устанавливаются глобально, поэтому вы должны заботиться о загрязнении в долгосрочной перспективе. | Packages used by npx are not installed globally so you have to carefree for the pollution for the long term. |

## атрибут src

**Если атрибут src установлен, содержимое тега script будет игнорироваться.**

В одном теге <script> нельзя использовать одновременно атрибут src и код внутри.

Нижеприведённый пример не работает:

<script src="file.js">

alert(1); // содержимое игнорируется, так как есть атрибут src

</script>

Нужно выбрать: либо внешний скрипт <script src="…">, либо обычный код внутри тега <script>.

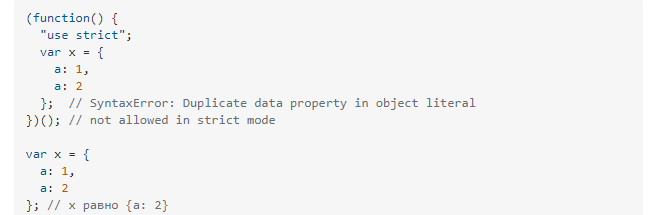
## Use strict

Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: "use strict".

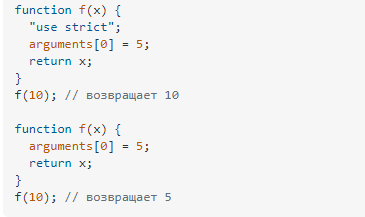
Директива выглядит как строка: "use strict" или 'use strict'. Когда она находится в начале скрипта, весь сценарий работает в «строгом» режиме.

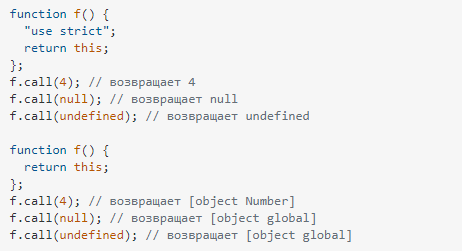
В строгом режиме:

* **нельзя присваивать значение в неопределённую переменную**
* **нельзя использовать инструкцию with**
* **в ES5 нельзя определить повторные свойства в литерале объекта**



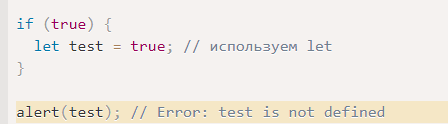
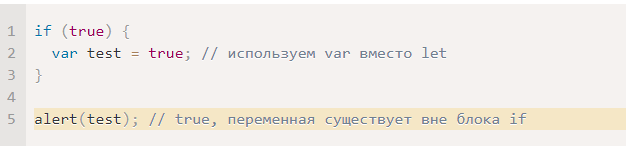
* **нельзя определить повторные формальные параметры функции**
* **изменения объекта arguments не изменяют аргументы**
* **delete приводит к ошибке, если аргумент - не изменяемое свойство объекта**



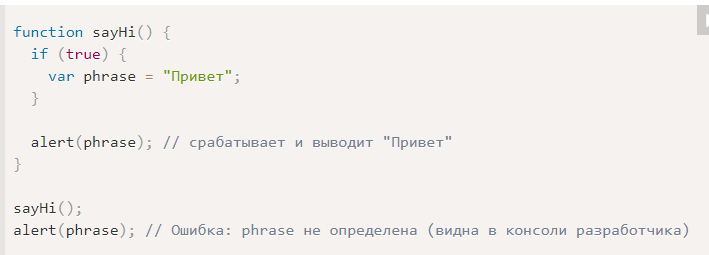
* **this не преобразуется в объект**, а **если значение this - undefined или null, то не преобразуется в глобальный объект**
* 
* **eval и arguments - нельзя изменить или использовать в качестве имени**
* **нельзя использовать argument.caller и arguments.callee**
* **больше слов, зарезервированных для использования в будущем**
* **нельзя использовать литералы восьмеричной СС**

## var

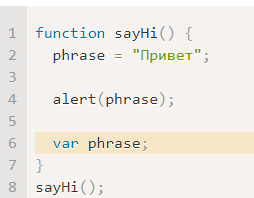
* [Для «var» не существует блочной области видимости](https://learn.javascript.ru/var" \l "dlya-var-ne-suschestvuet-blochnoy-oblasti-vidimosti)

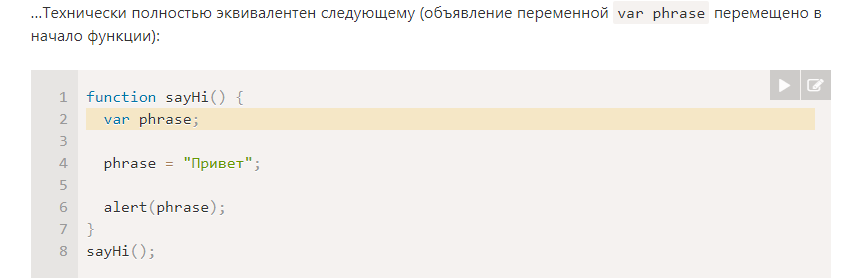
 

Но Если блок кода находится внутри функции, то var становится локальной переменной в этой функции:

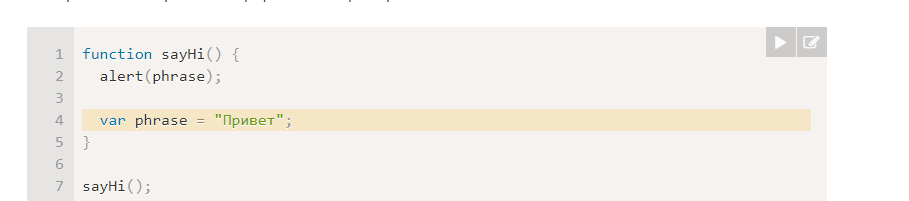


* [«var» обрабатываются в начале запуска функции](https://learn.javascript.ru/var" \l "var-obrabatyvayutsya-v-nachale-zapuska-funktsii)

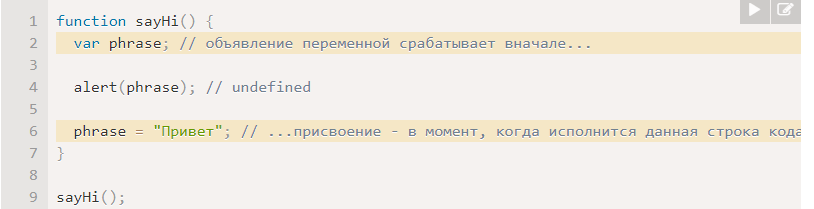




* Объявления переменных «всплывают», но присваивания значений – нет.



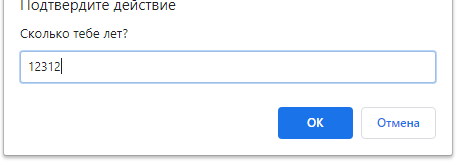
Выполняется следующим образом



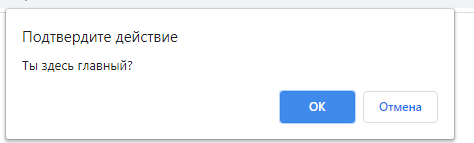
## [prompt](https://learn.javascript.ru/alert-prompt-confirm" \l "prompt)

      let age = prompt("Сколько тебе лет?", 100);

      alert(`Тебе ${age} лет!`); *// Тебе 100 лет*

вот такая штука

## confirm



let isBoss = confirm("Ты здесь главный?");

alert( isBoss ); // true, если нажата OK

# ФИЧИ синтаксиса JS

* чувствительный к **регистру**
* Необязательные **точки с запятой**
* **Комментарии** как в с++
* **Кавычки** не важно одинарные или нет
* В javascript есть ток int string bool, значения для них понятно какие + null, undefined
* **Фунции** еcть, классы все как в с++ c#
* Наконец, **значениями свойств в объектных литералах не обязательно должны быть константы** – это могут быть **произвольные JavaScript выражения**. Кроме того, **в качестве имен свойств** в объектных литералах допускается использовать **строковые значения**:

var square = { "upperLeft": { x:point.x, y:point.y },

'lowerRight': { x:(point.x + side), y:(point.y+side) }}

# Переменные

• Если переменная равна null, следовательно, в ней не содержится допустимого объекта, массива, числа, стро ки или логического значения

* if(a instanceof Array) проверка на тип
* { x:1, y:2 } // Инициализатор **объекта**

[1,2,3,4,5] // Инициализатор **массива**

* **Объекты в JavaScript** отображают имена свойств на произвольные значения.
* object.property и object["property"] – это одно и тоже

## Тип данных Symbol

«Символ» представляет собой уникальный идентификатор.

Символы гарантированно уникальны

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

let user = {

    name: "Вася"

};

let id = Symbol("id");

user[id] = 1;

document.write(JSON.stringify(user)); *//{"name":"Вася"}*

document.write("<br>", user[id]); *//1*

document.write("<br>", user.id); *//undefined*

for (let key in user) alert(key); // name, (свойства с ключом-символом нет среди перечисленных)

Это вызвано тем, что нам нужно использовать значение переменной id в качестве ключа, а не строку «id».

Object.keys(user) также игнорирует символы

Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство.

Для этого существует **глобальный реестр символов**. Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам гарантированно будет возвращаться один и тот же символ.

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

// читаем символ из глобального реестра и записываем его в переменную

let id = Symbol.for("id"); // если символа не существует, он будет создан

// читаем его снова в другую переменную (возможно, из другого места кода)

let idAgain = Symbol.for("id");

// проверяем -- это один и тот же символ

alert( id === idAgain ); // true

**Symbol.keyfor**

Для глобальных символов, кроме Symbol.for(key), который ищет символ по имени, существует обратный метод: Symbol.keyFor(sym), который, наоборот, принимает глобальный символ и возвращает его имя.

## Примитивы

Я особо ничо не поняял… поэтммоу лучше заново прочитай

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в [спецификации](https://tc39.github.io/ecma262/#sec-toprimitive): string, number, default

**default"**

Происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать с обоими типами: строками (объединять их) и числами (складывать).

**В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:**

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен "string"
   * пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он существует.
3. В случае, если хинт равен "number" или "default"
   * пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он существует

[**Symbol.toPrimitive**](https://learn.javascript.ru/object-toprimitive#symbol-toprimitive)

Начнём с универсального подхода – символа Symbol.toPrimitive: метод с таким названием (если есть) используется для всех преобразований:

obj[Symbol.toPrimitive] = function(hint) {

// должен вернуть примитивное значение

// hint равно чему-то одному из: "string", "number" или "default"

};

Для примера используем его в реализации объекта user:

let user = {

name: "John",

money: 1000,

[Symbol.toPrimitive](hint) {

alert(`hint: ${hint}`);

return hint == "string" ? `{name: "${this.name}"}` : this.money;

}

};

// демонстрация результатов преобразований:

alert(user); // hint: string -> {name: "John"}

alert(+user); // hint: number -> 1000

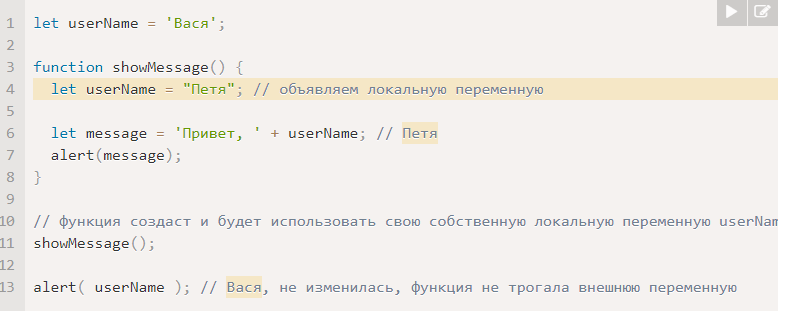
alert(user + 500); // hint: default -> 1500

Как мы видим из кода, user преобразовывается либо в информативную читаемую строку, либо в денежный счёт в зависимости от значения хинта. Единственный метод user[Symbol.toPrimitive] смог обработать все случаи преобразований.

## Глобальные переменные

* Global variables can be used (and changed) by all scripts in the page (and in the window).
* Variables created **without** a declaration keyword (var, let, or const) are always global, even if they are created inside a function.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).



## Map

это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

* new Map() – создаёт коллекцию.
* map.set(key, value) – записывает по ключу key значение value.
* map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* map.delete(key) – удаляет элемент по ключу key.
* map.clear() – очищает коллекцию от всех элементов.
* map.size – возвращает текущее количество элементов.
* map.keys() – возвращает итерируемый объект по ключам,
* map.values() – возвращает итерируемый объект по значениям,
* map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по элементам в формате [ключ, значение]

for (let entry of recipeMap) { // то же самое, что и recipeMap.entries()

alert(entry); // огурец,500 (и так далее)

}

**Object - > Map**

let obj = {

name: "John",

age: 30

};

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

Здесь Object.entries возвращает массив пар ключ-значение: [ ["name","John"], ["age", 30] ]. Это именно то, что нужно для создания Map.

**Map->Object**

let map = new Map();

map.set('banana', 1);

map.set('orange', 2);

map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // make a plain object (\*)// или Object.fromEntries(map);

// готово!

// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2

## WeakMap

Как мы знаем из главы [Сборка мусора](https://learn.javascript.ru/garbage-collection), движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы).

let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

// объект john хранится в массиве, поэтому он не будет удалён сборщиком мусора

// мы можем взять его значение как array[0]

Аналогично, если мы используем объект как ключ в Map, то до тех пор, пока существует Map, также будет существовать и этот объект. Он занимает место в памяти и не может быть удалён сборщиком мусора.

let john = { name: "John" };

let map = new Map();

map.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`,

// он доступен через map.keys()

WeakMap – принципиально другая структура в этом аспекте. Она не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей.

Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями:

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // работает (объект в качестве ключа)

// нельзя использовать строку в качестве ключа

weakMap.set("test", "Whoops"); // Ошибка, потому что "test" не объект

!!**В основном, WeakMap используется в качестве дополнительного хранилища данных.**

weakMap.set(john, "секретные документы");

// если john умрёт, "секретные документы" будут автоматически уничтожены

Другая частая сфера применения – это кеширование, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его.

## Set

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

* new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
* set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
* set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.
* set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.
* set.clear() – удаляет все имеющиеся значения.
* set.size – возвращает количество элементов в множестве.
* set.values() – возвращает перебираемый объект для значений,
* set.keys() – то же самое, что и set.values(), присутствует для обратной совместимости с Map,
* set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

! При повторных вызовах set.add() с одним и тем же значением ничего не происходит

let set = new Set();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз

set.add(john);

set.add(pete);

set.add(mary);

set.add(john);

set.add(mary);

// set хранит только 3 уникальных значения

alert(set.size); // 3

for (let user of set) {

alert(user.name); // John (потом Pete и Mary)

}

**[Перебор объекта Set](https://learn.javascript.ru/map-set" \l "perebor-obekta-set)**

let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

// то же самое с forEach:

set.forEach((value, valueAgain, set) => {

alert(value);

});

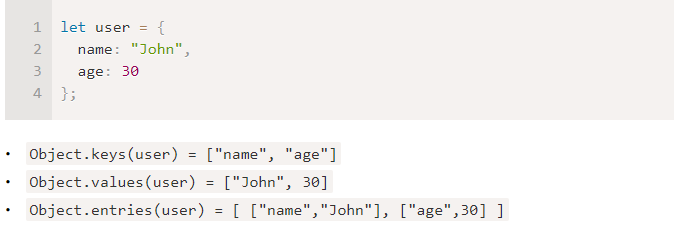
## [WeakSet](https://learn.javascript.ru/weakmap-weakset" \l "weakset)

Коллекция WeakSet ведёт себя похоже:

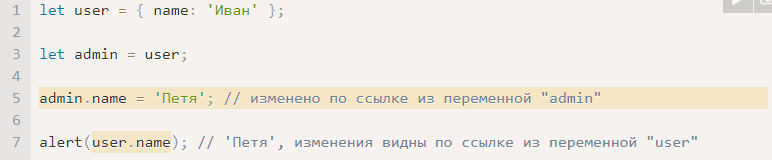
* Она аналогична Set, но мы можем добавлять в WeakSet только объекты (не примитивные значения).
* Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.
* Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой.

# Объекты

## Keys values entries



## Объекты хранят ссылки

Поэтому если копируем, то будет копироваться ссылка а не сам объект

### Операторы равенства

 == и строгого равенства === **для объектов** работают одинаково.

### ****Два объекта равны только в том случае, если это один и тот же объект.****

В примере ниже две переменные ссылаются на один и тот же объект, поэтому они равны друг другу:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true, т.к. обе переменные ссылаются на один и тот же объект

alert( a === b ); // true

В другом примере два разных объекта не равны, хотя оба пусты:

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

## Передача в функцию параметров как свойств из объекта

// мы передаём объект в функцию

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

// ...и она немедленно извлекает свойства в переменные

function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {

// title, items – взято из options,

// width, height – используются значения по умолчанию

alert( `${title} ${width} ${height}` ); // My Menu 200 100

alert( items ); // Item1, Item2

}

showMenu(options);

## Реальное копирование объектов

let user = {

name: "Иван",

age: 30

};

let clone = {}; // новый пустой объект

// скопируем все свойства user в него

for (let key in user) {

clone[key] = user[key];

}

// теперь в переменной clone находится абсолютно независимый клон объекта

clone.name = "Пётр"; // изменим в нём данные

alert( user.name ); // в оригинальном объекте значение свойства `name` осталось прежним – Иван.

Кроме того, для этих целей мы можем использовать метод [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign).

Синтаксис:

Object.assign(dest, [src1, src2, src3...])

* Первый аргумент dest — целевой объект.
* Остальные аргументы src1, ..., srcN (может быть столько, сколько нужно)) являются исходными объектами
* Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект.
* Возвращает объект dest.
* Например, объединим несколько объектов в один:
* let user = { name: "Иван" };
* let permissions1 = { canView: true };
* let permissions2 = { canEdit: true };
* // копируем все свойства из permissions1 и permissions2 в user
* Object.assign(user, permissions1, permissions2);
* // теперь user = { name: "Иван", canView: true, canEdit: true }

## Проверка существования свойств

## перебор всех свойств объекта

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

// ключи

alert( key ); // name, age, isAdmin

// значения ключей

alert( user[key] ); // John, 30, true

}

Важно что если ключи **целочисленные**, когда мы будем их перечислять, они будут в абсолютно случайном порядкеname

## [Рекурсивные обходы](https://learn.javascript.ru/recursion" \l "rekursivnye-obhody) объектов(поиск массива внутри)

Другим отличным применением рекурсии является рекурсивный обход.

Представьте, у нас есть компания. Структура персонала может быть представлена как объект:

let company = { // тот же самый объект, сжатый для краткости

sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600 }],

development: {

sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],

internals: [{name: 'Jack', salary: 1300}]

}

};

// Функция для подсчёта суммы зарплат

function sumSalaries(department) {

if (Array.isArray(department)) { // случай (1)

return department.reduce((prev, current) => prev + current.salary, 0); // сумма элементов массива

} else { // случай (2)

let sum = 0;

for (let subdep of Object.values(department)) {

sum += sumSalaries(subdep); // рекурсивно вызывается для подотделов, суммируя результаты

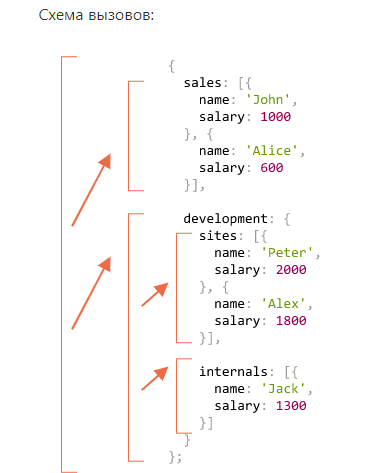
}

return sum;

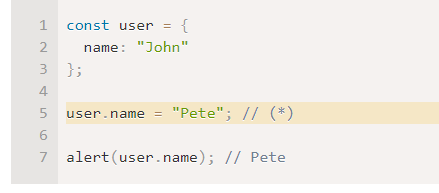
}

}

alert(sumSalaries(company)); // 6700

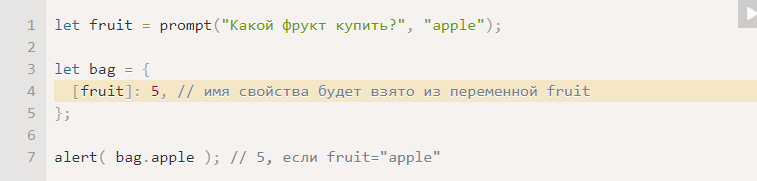


## const object может быть изменён



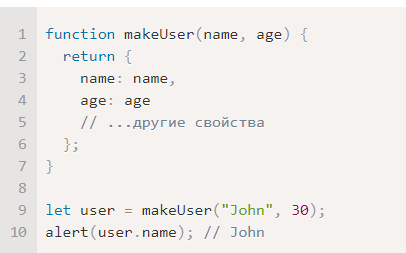
## Изменяемые свойства объекта

Мы можем подставлять в квадратные скобки в свои строки

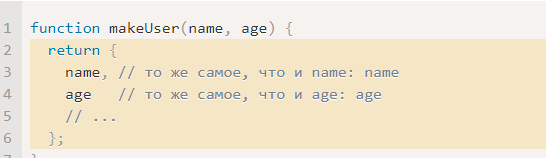


## «Конструктор» объекта через функцию

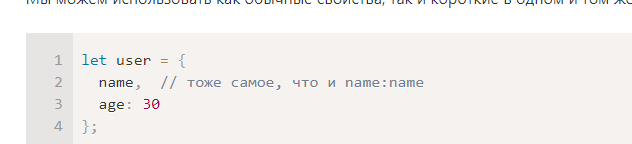
Мы можем через скобочки перечислять значения объекта, сделав функцию возращающую объект



Сокращенно:



Когда пишем объект тоже самое

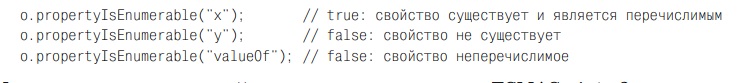


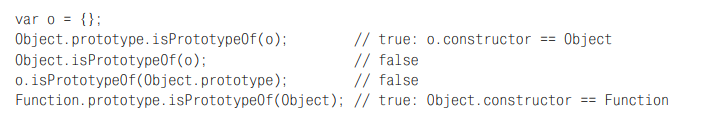
## Удаление свойств

delete book.chapter2;

причем он именно удаляет а не ставит undefined

## свойства объектов

* Метод **hasOwnProperty**() возвращает true, если для объекта определено не унасле дованное свойство с именем
* **toLocaleString**(). Получает **локализованный** строковое представление объекта. По умолчанию метод toLocaleString(), определяемый классом Object, никакой локализации не выполняет; он всегда возвращает в точности такую же строку, что и **toString**(). Однако под классы могут определять собственные версии метода toLocaleString().
* propertyIsEnumerable() возвращает true, если в объекте определено свойство с именем, указанным в единственном строковом аргументе метода, и это свой ство может быть перечислено циклом for/in. 
* isPrototypeOf() возвращает true, если объект, которому принадлежит ме тод, является прототипом объекта, передаваемого методу в качестве аргумента.



## Флаги и дескрипторы свойств

ВСЕ РАБОТАЕТ ТОЛЬКО В СТРОГОМ РЕЖИМЕ!!!!

### [Флаги свойств](https://learn.javascript.ru/property-descriptors#flagi-svoystv)

Помимо значения **value**, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

* **writable** – если true, свойство можно изменить, иначе оно только для чтения.
* **enumerable** – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
* **configurable** – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

### getOwnPropertyDescriptor

Метод Object.getOwnPropertyDescriptor позволяет получить полную информацию о свойстве.

Его синтаксис:

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

let user = {

name: "John"

};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\* дескриптор свойства:

{

"value": "John",

"writable": true,

"enumerable": true,

"configurable": true

}

\*/

### defineProperty

let user = {

name: "John",

toString() {

return this.name;

}

};

Object.defineProperty(user, "toString", {

enumerable: false

});

// Теперь наше свойство toString пропало из цикла:

for (let key in user) alert(key); // name

### Другие редкие свойства

Но ещё есть методы, которые ограничивают доступ ко всему объекту:

[**Object.preventExtensions(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions)

Запрещает добавлять новые свойства в объект.

[**Object.seal(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/seal)

Запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.

[**Object.freeze(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze)

Запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть методы для их проверки:

[**Object.isExtensible(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible)

Возвращает false, если добавление свойств запрещено, иначе true.

[**Object.isSealed(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed)

Возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.

[**Object.isFrozen(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isFrozen)

Возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

## [Геттеры и сеттеры](https://learn.javascript.ru/property-accessors" \l "gettery-i-settery)

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

let obj = {

get propName() {

// геттер, срабатывает при чтении obj.propName

},

set propName(value) {

// сеттер, срабатывает при записи obj.propName = value

}

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение присваивается.

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

}

};

alert(user.fullName); // John Smith

Мы не вызываем user.fullName как функцию, а читаем как обычное свойство: геттер выполнит всю работу за кулисами.

У них еще есть как и флагов свойств:

* **enumerable** – то же самое, что и для свойств-данных,
* **configurable** – то же самое, что и для свойств-данных

# Опциональная цепочка '?.'

Опциональная цепочка ?. — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует.

Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если часть перед ?. имеет значение undefined или null.

**Для краткости в этой статье мы будем говорить о значении, что оно «существует», если оно отличается от null или undefined.**

Вот безопасный способ обратиться к свойству user.address.street:

let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)

Чтение адреса с помощью конструкции user?.address выполняется без ошибок, даже если объекта user не существует:

let user = null;

alert( user?.address ); // undefined

alert( user?.address.street ); // undefined

Обратите внимание, что синтаксис ?. делает необязательным только свойство перед ним, а не какое-либо последующее.

# Массив

* **Массив** может быть создан с помощью функцииконструктора Array(). Созданному массиву допустимо присваивать любое количество индексированных элементов: var a = new Array(); a[0] = 1.2; a[1] = "JavaScript"; a[2] = true; a[3] = { x:1, y:3 };

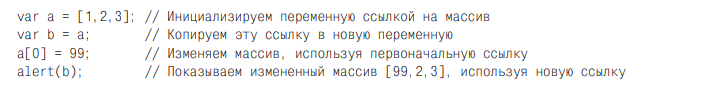
Литерал, или **инициализатор, массива** – это список разделенных запятыми значений, заключенных в квадратные скобки

var a = [1.2, "JavaScript", true, { x:1, y:3 }];

Для того чтобы включить **в литерал массива неопределенный элемент**, достаточно пропустить значение между запятыми. Следующий массив содержит пять элементов, в том числе три неопределенных: var sparseArray = [1,,,,5];

* Массивы и функции всегда передаются по ссылке

Во второй строчке присваивается по ссылке а не копируется



* Массивы в JavaScript могут быть **разреженными**

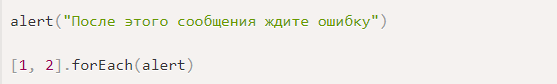
a[0] = 1;

a[10000] = "это элемент 10,000";

-и между ними ничего нетт

* Оператор **delete** записывает в элемент массива значение **undefined**, при этом сам элемент массива продолжает свое существование. Чтобы удалить полностью надо юзать pop\shift

## Методы массивов

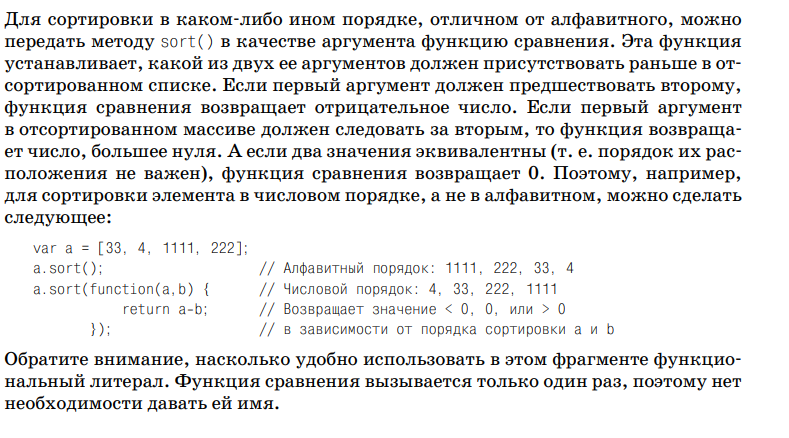


* Array.join() преобразует все элементы массива в строки и объединяет их.
* Array.reverse() ясно
* Sort()

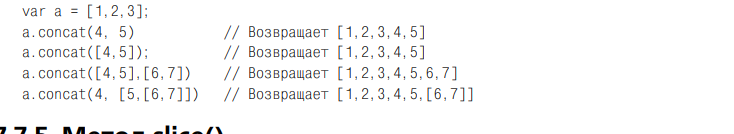
var a = new Array("banana", "cherry", "apple");

a.sort();

var s = a.join(", "); // s == "apple, banana, cherry"



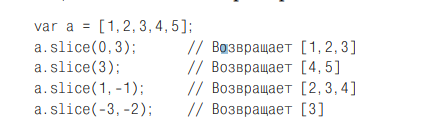
* Метод Array.**concat**() создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод concat()



* **slice**() возвращает фрагмент, или подмассив, указанного массива.

Два аргумента метода определяют начало и конец возвращаемого фрагмента.

аргумент, равный –1, задает последний элемент массива



* **Array.splice**() – это универсальный метод для вставки или удаления элементов массив

Он изменяет массив на месте, а не возвращает новый массив, как это делают методы slice() и concat().

# Деструктуризация

/ у нас есть массив с именем и фамилией

let arr = ["Ilya", "Kantor"]

// деструктурирующее присваивание

// записывает firstName=arr[0], surname=arr[1]

let [firstName, surname] = arr;

alert(firstName); // Ilya

alert(surname); // Kantor

**Пропускайте элементы, используя запятые**

// второй элемент не нужен

let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul

**Работает с любым перебираемым объектом с правой стороны**

let [a, b, c] = "abc";

let [one, two, three] = new Set([1, 2, 3]);

**Присваивайте чему угодно с левой стороны**

Мы можем использовать что угодно «присваивающее» с левой стороны.

Например, можно присвоить свойству объекта:

let user = {};

[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya

**Цикл с .entries()**

let user = {

name: "John",

age: 30

};

// цикл по ключам и значениям

for (let [key, value] of Object.entries(user)) {

alert(`${key}:${value}`); // name:John, затем age:30

}

…то же самое для map:

let user = new Map();

user.set("name", "John");

user.set("age", "30");

for (let [key, value] of user) {

alert(`${key}:${value}`); // name:John, затем age:30

}

Передача в функцию параметров как свойств из объекта

// мы передаём объект в функцию

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

// ...и она немедленно извлекает свойства в переменные

function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {

// title, items – взято из options,

// width, height – используются значения по умолчанию

alert( `${title} ${width} ${height}` ); // My Menu 200 100

alert( items ); // Item1, Item2

}

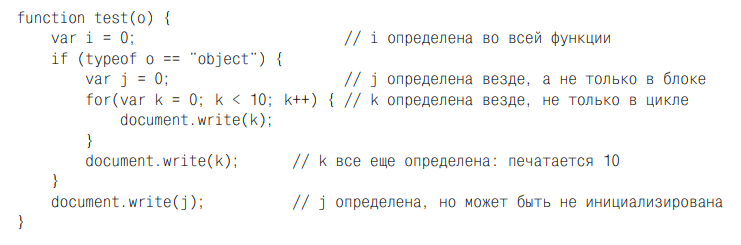
showMenu(options);

# **errors**

в js не exceptions a errors, основные:

Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError и URIError

# Функции

* In JavaScript all functions are object methods.
* If a function is not a method of a JavaScript object, it is a function of the global object (see previous chapter).
* Массивы и функции всегда передаются по ссылке
* в отличие от C, C++ и Java, в JavaScript нет области видимости на уровне блоков. Все переменные, объявленные внутри функции, независимо от того, где именно это сделано, определены во всей функции 

function User(name) {

// this = {}; (неявно)

// добавляет свойства к this

this.name = name;

this.isAdmin = false;

// return this; (неявно)

}

## Объект функции

Объект функции содержит несколько полезных свойств.

### свойство «name»:

Например, имя функции нам доступно как свойство «**name**»:

function sayHi() {

alert("Hi");

}

alert(sayHi.name); // sayHi

или

let sayHi = function() {

alert("Hi");

};

alert(sayHi.name); // sayHi (есть имя!)

### свойство «length»

Ещё одно встроенное свойство «length» содержит количество параметров функции в её объявлении. Например:

function f1(a) {}

function f2(a, b) {}

function many(a, b, ...more) {}

alert(f1.length); // 1

alert(f2.length); // 2

alert(many.length); // 2

### Свойства функции(не переменные)

function sayHi() {

alert("Hi");

// давайте посчитаем, сколько вызовов мы сделали

sayHi.counter++;

}

sayHi.counter = 0; // начальное значение

sayHi(); // Hi

sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза

## JavaScript Closures(замыкание)

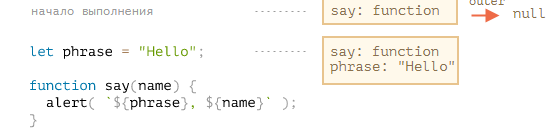
функция может получить доступ к переменным из внешнего окружения

Превью про окружение в конце…

Превью про окружение:

### [Внутреннее и внешнее лексическое окружение](https://learn.javascript.ru/closure" \l "vnutrennee-i-vneshnee-leksicheskoe-okruzhenie)

**Function declaration - В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение.**



В течение вызова say() использует внешнюю переменную phrase. Давайте разберёмся подробно, что происходит.

При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова.

Например, для say("John") это выглядит так (выполнение находится на строке, отмеченной стрелкой):

**Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.**

**! Функция получает текущее значение внешних переменных, то есть, их последнее значение**

**Один вызов – одно лексическое окружение**

Пожалуйста, обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется.

И, если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

### Возращаемая функция

function makeCounter() {

let count = 0;

return function() {

return count++; // есть доступ к внешней переменной "count"

};

}

let counter = makeCounter();

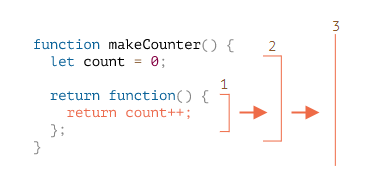
alert( counter() ); // 0

alert( counter() ); // 1

alert( counter() ); // 2

Как же это работает изнутри?

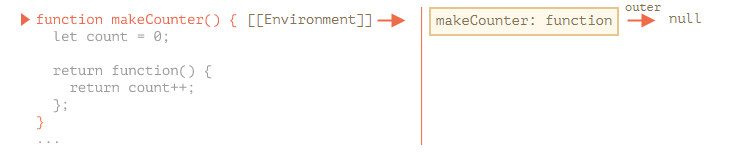
Когда внутренняя функция начинает выполняться, начинается поиск переменной count++ изнутри-наружу. Для примера выше порядок будет такой:



В этом примере count будет найден на шаге 2. Когда внешняя переменная модифицируется, она изменится там, где была найдена. Значит, count++ найдёт внешнюю переменную и увеличит её значение в лексическом окружении, которому она принадлежит. Как если бы у нас было let count = 1.

Если степ-ву-степ:

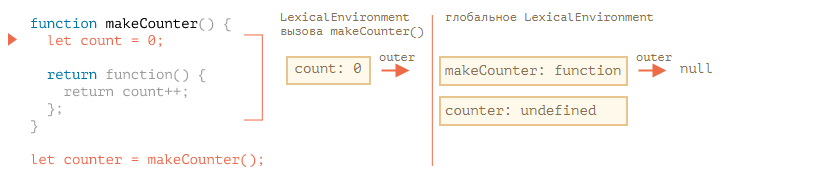
* 1. Когда скрипт только начинает выполняться, есть только глобальное лексическое окружение:

В этот начальный момент есть только функция makeCounter, потому что это Function Declaration. Она ещё не выполняется.

**Все функции «при рождении» получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, где они были созданы.**

В данном случае, makeCounter создан в глобальном лексическом окружении, так что [[Environment]] содержит ссылку на него.

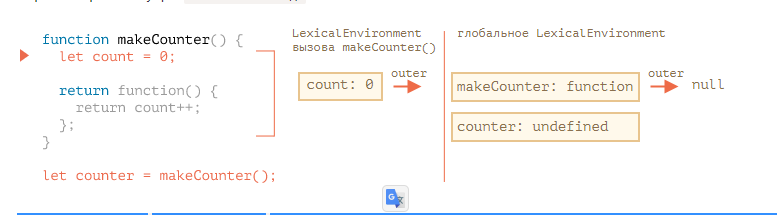
* 1. Код продолжает выполняться, объявляется новая глобальная переменная counter, которой присваивается результат вызова makeCounter. Вот снимок момента, когда интерпретатор находится на первой строке внутри makeCounter():\



1. В момент вызова makeCounter() создаётся лексическое окружение, для хранения его переменных и аргументов.

Как и все лексические окружения, оно содержит две вещи:

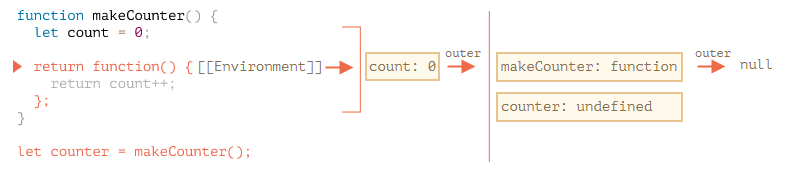
* 1. Environment Record с локальными переменными. В нашем случае count – единственная локальная переменная (появляющаяся, когда выполняется строчка с let count).
  2. Ссылка на внешнее окружение, которая устанавливается в значение [[Environment]] функции. В данном случае, [[Environment]] функции makeCounter ссылается на глобальное лексическое окружение.



Итак, теперь у нас есть два лексических окружения: первое – глобальное, второе – для текущего вызова makeCounter, с внешней ссылкой на глобальный объект.

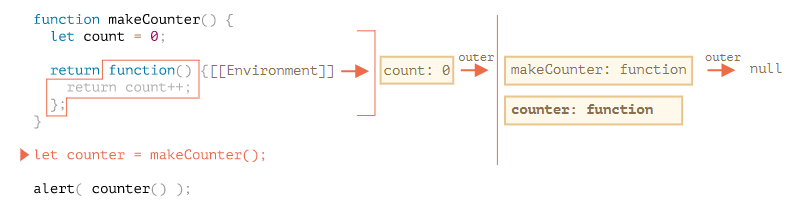
* 1. В процессе выполнения makeCounter() создаётся небольшая вложенная функция

Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство [[Environment]], которое ссылается на лексическое окружение, в котором они были созданы. То же самое происходит и с нашей новой маленькой функцией.

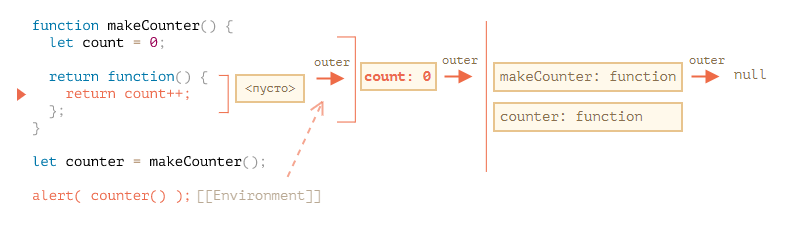


Пожалуйста, обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри function() { return count++ } не выполняется.

* 1. Выполнение продолжается, вызов makeCounter() завершается, и результат (небольшая вложенная функция) присваивается глобальной переменной counter:



1. При вызове counter() для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом counter локальных переменных нет. Но [[Environment]] counter используется, как ссылка на внешнее лексическое окружение outer, которое даёт доступ к переменным предшествующего вызова makeCounter, где counter был создан.



Теперь, когда вызов ищет переменную count, он сначала ищет в собственном лексическом окружении (пустое), а затем в лексическом окружении предшествующего вызова makeCounter(), где и находит её.

1. Вызов counter() не только возвращает значение count, но также увеличивает его. Обратите внимание, что модификация происходит «на месте». Значение count изменяется конкретно в том окружении, где оно было найдено.
2. Можем ли мы каким-нибудь образом сбросить счётчик count из кода, который не принадлежит makeCounter? Например, после вызова alert в коде выше.
3. Если мы вызываем makeCounter несколько раз – нам возвращается много функций counter. Они независимы или разделяют одну и ту же переменную count?

ответы

1. Такой возможности нет: count – локальная переменная функции, мы не можем получить к ней доступ извне.
2. Для каждого вызова makeCounter() создаётся новое лексическое окружение функции, со своим собственным count. Так что, получившиеся функции counter – независимы.

let counter1 = makeCounter();

let counter2 = makeCounter();

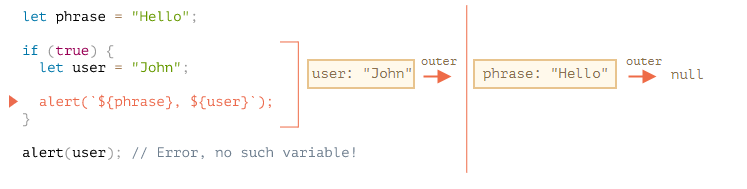
alert( counter1() ); // 0

alert( counter1() ); // 1

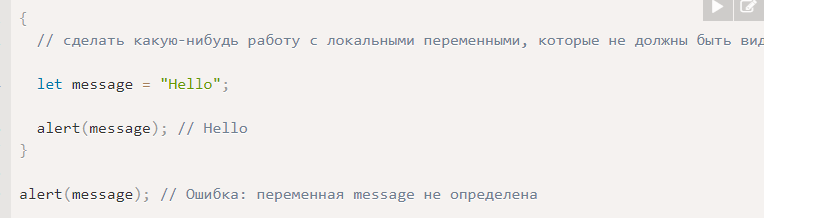
alert( counter2() ); // 0 (независимо)

### лексическое окружение существует для любых блоков кода {...}.

if, while, for …



Мы также можем использовать «простые» блоки кода {...}, чтобы изолировать переменные в «локальной области видимости».



### [IIFE](https://learn.javascript.ru/closure" \l "iife)

В прошлом в JavaScript не было лексического окружения на уровне блоков кода.

Так что программистам пришлось что-то придумать. И то, что они сделали, называется «immediately-invoked function expressions» (аббревиатура IIFE), что означает функцию, запускаемую сразу после объявления.



Function Expression обёрнуто в скобки (function {...}), потому что, когда JavaScript встречает "function" в основном потоке кода, он воспринимает это как начало Function Declaration. Но у Function Declaration должно быть имя, так что такой код вызовет ошибку:

// Попробуйте объявить и сразу же вызвать функцию

function() { // <-- Error: Unexpected token (

let message = "Hello";

alert(message); // Hello

}();

скобки вокруг функции – это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения, и, таким образом, это функциональное выражение: ей не нужно имя и её можно вызвать немедленно.

### [Сборка мусора](https://learn.javascript.ru/closure" \l "sborka-musora)

**Все как обычно** есть вложенная функция, которая всё ещё доступна после выполнения f, то у неё есть свойство [[Environment]], которое ссылается на внешнее лексическое окружение, тем самым оставляя его достижимым, «живым»:

function f() {

let value = 123;

function g() { alert(value); }

return g;

}

let g = f(); // g доступно и продолжает держать внешнее лексическое окружение в памяти

**НО**

**Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке.**

function f() {

let value = Math.random();

function g() {

debugger; // в консоли: напишите alert(value); Такой переменной нет!

}

return g;

}

let g = f();

g();

**м**

## [Named Function Expression](https://learn.javascript.ru/function-object" \l "named-function-expression)

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя.

let sayHi = function(who) {

alert(`Hello, ${who}`);

};

И присвоим ему имя:

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

let sayHi = function func(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

func("Guest"); // использует func, чтобы снова вызвать себя же

}

};

sayHi(); // Hello, Guest

// А вот так - не cработает:

func(); // Ошибка, func не определена (недоступна вне функции)

Почему мы используем func? Почему просто не использовать sayHi для вложенного вызова?

Вообще, обычно мы можем так поступить:

let sayHi = function(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

sayHi("Guest");

}

};

Однако, у этого кода есть проблема, которая заключается в том, что значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнёт выдавать ошибки:

let sayHi = function(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

sayHi("Guest"); // Ошибка: sayHi не является функцией

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не работает!

Так происходит, потому что функция берёт sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null.

Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы.

Давайте используем его, чтобы исправить наш код:

let sayHi = function func(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

func("Guest"); // Теперь всё в порядке

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)

## .map

let prices = {

banana: 1,

orange: 2,

meat: 4,

};

let doublePrices = Object.fromEntries(

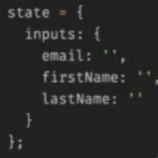
// преобразовать в массив, затем map, затем fromEntries обратно объект

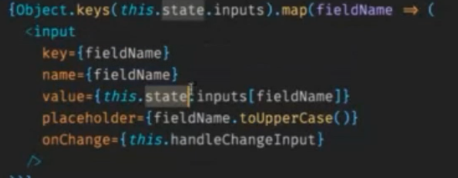
Object.entries(prices).map(([key, value]) => [key, value \* 2])

);

alert(doublePrices.meat); // 8

Пример:





## call

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the **fullName** method of person, using it on **person2**:

var person = {  
  **fullName**: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
var person2 = {  
  firstName:"Mary",  
  lastName: "Doe"  
}  
person.fullName.call(**person2**);  // Will return "Mary Doe"

The call() method can accept arguments:

var person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
person.fullName.call(person1, "Oslo", "Norway");

### call для передачи контекста

синтаксис:

func.call(context, arg1, arg2, ...)

----

function sayHi() {

alert(this.name);

}

let user = { name: "John" };

let admin = { name: "Admin" };

// используем 'call' для передачи различных объектов в качестве 'this'

sayHi.call( user ); // John

sayHi.call( admin ); // Admin

### Заимствование метода

допустим надо сделать функцию и вернуть строку из N аргументов

function hash(args) {

return args.join();

}

…К сожалению, это не сработает, потому что мы вызываем hash(arguments), а объект arguments является перебираемым и псевдомассивом, но не реальным массивом.

function hash() {

alert( arguments.join() ); // Ошибка: arguments.join не является функцией

}

hash(1, 2);

Тем не менее, есть простой способ использовать соединение массива:

function hash() {

alert( [].join.call(arguments) ); // 1,2

}

hash(1, 2);

Мы берём (заимствуем) метод join из обычного массива [].join. И используем [].join.call, чтобы выполнить его в контексте arguments.

## Apply

func.apply(context, args) – вызывает func, передавая context как this и псевдомассив args как список аргументов.

The apply() method is similar to the call() method (previous chapter).

**The Difference Between call() and apply()**

The difference is:

* The call() method takes arguments **separately**.
* The apply() method takes arguments as an **array**.

The apply() method is very handy if you want to use an array instead of an argument list.

**Example 0:**

Since JavaScript **arrays** do not have a max() method, you can apply the Math.max() method instead.

Math.max.apply(null, [1,2,3]); // Will also return 3

[default](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math_apply) max method: Math.max(1,2,3);  // Will return 3

**Example 1:**

var person = {  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
var person1 = {  
  firstName: "Mary",  
  lastName: "Doe"  
}  
person.fullName.apply(person1);  // Will return "Mary Doe"

**Example 2**

var person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
person.fullName.apply(person1, ["Oslo", "Norway"]);

## Invoking a Function as a Method

In JavaScript you can define functions as object methods. То есть у объектов можно просто объявлять методы. Все.

The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

var myObject = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
}  
myObject.fullName();         // Will return "John Doe"

## New Function

let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3

А вот функция без аргументов, в этом случае достаточно указать только тело:

let sayHi = new Function('alert("Hello")');

sayHi(); // Hello

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

### [Замыкание](https://learn.javascript.ru/new-function" \l "zamykanie)

Но когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на внешнее лексическое окружение, в котором она была создана, а на глобальное. Поэтому такая функция имеет доступ только к глобальным переменным.

function getFunc() {

let value = "test";

let func = new Function('alert(value)');

return func;

}

getFunc()(); // ошибка: value не определено

Сравним это с обычным объявлением:

function getFunc() {

let value = "test";

let func = function() { alert(value); };

return func;

}

getFunc()(); // "test", из лексического окружения функции getFunc

Эта особенность new Function выглядит странно, но оказывается очень полезной на практике.

Представьте, что нужно создать функцию из строки. Код этой функции неизвестен во время написания скрипта (поэтому не используем обычные функции), а будет определён только в процессе выполнения. Мы можем получить код с сервера или с другого ресурса.

Наша новая функция должна взаимодействовать с основным скриптом.

Что если бы она имела доступ к внешним переменным?

Проблема в том, что перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью минификатора – специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена.

Например, если в функции объявляется переменная let userName, то минификатор изменяет её на let a (или другую букву, если она не занята) и изменяет её везде. Обычно так делать безопасно, потому что переменная является локальной, и никто снаружи не имеет к ней доступ. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы достаточно умные. Они не просто осуществляют «тупой» поиск-замену, они анализируют структуру кода, и поэтому ничего не ломается.

Так что если бы даже new Function и имела доступ к внешним переменным, она не смогла бы найти переименованную userName.

**Если бы new Function имела доступ к внешним переменным, при этом были бы проблемы с минификаторами.**

## [Функция-конструктор](https://learn.javascript.ru/constructor-new#funktsiya-konstruktor)

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

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

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

// this = {}; (неявно)

// добавляет свойства к this

this.name = name;

this.isAdmin = false;

// return this; (неявно)

}

То есть, результат вызова new User("Вася") – это тот же объект, что и:

let user = {

name: "Вася",

isAdmin: false

};

**new function() { … }**

Если в нашем коде большое количество строк, создающих один сложный объект, мы можем обернуть их в функцию-конструктор следующим образом:

let user = new function() {

this.name = "Вася";

this.isAdmin = false;

// ...другой код для создания пользователя

// возможна любая сложная логика и выражения

// локальные переменные и т. д.

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается

## [Возврат значения из конструктора return](https://learn.javascript.ru/constructor-new" \l "vozvrat-znacheniya-iz-konstruktora-return)

Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в this, который в итоге станет результатом.

Но если return всё же есть, то применяется простое правило:

* При вызове return с объектом, будет возвращён объект, а не this.
* При вызове return с примитивным значением, примитивное значение будет отброшено.

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this.

В примере ниже return возвращает объект вместо this:

function BigUser() {

this.name = "Вася";

return { name: "Godzilla" }; // <-- возвращает этот объект

}

alert( new BigUser().name ); // Godzilla, получили этот объект

## [Функции-«колбэки»](https://learn.javascript.ru/function-expressions#funktsii-kolbeki)

Давайте напишем функцию ask(question, yes, no) с тремя параметрами:

**question**

Текст вопроса

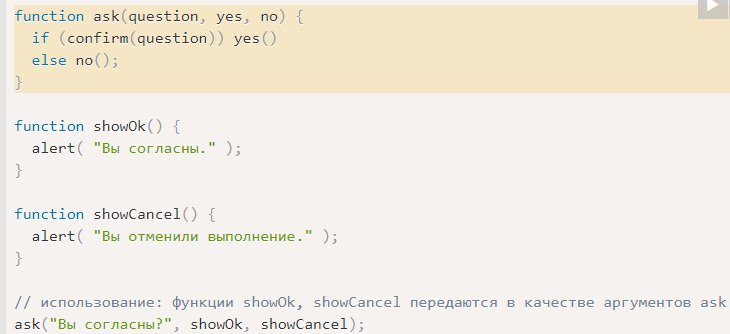
**yes**

Функция, которая будет вызываться, если ответ будет «Yes»

**no**

Функция, которая будет вызываться, если ответ будет «No»

Наша функция должна задать вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no():



## [Создание методов в конструкторе](https://learn.javascript.ru/constructor-new" \l "sozdanie-metodov-v-konstruktore)

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В this мы можем добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi:

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Меня зовут: " + this.name );

};

}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/\*

vasya = {

name: "Вася",

sayHi: function() { ... }

}

\*/

## return

**Никогда не добавляйте перевод строки между return и его значением**

Для длинного выражения в return может быть заманчиво разместить его на нескольких отдельных строках, например так:

return

(some + long + expression + or + whatever \* f(a) + f(b))

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после return. Для него это будет выглядеть так:

return;

(some + long + expression + or + whatever \* f(a) + f(b))

## Списки аргументов переменной длины

arguments – это специальное свойство объекта вызова, ссылающееся на объект, известный как объект Arguments(чето типо массива)

function sumAll(...args) { // args — имя массива

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

* Мы можем положить первые несколько параметров в переменные, а остальные – собрать в массив: function showName(firstName, lastName, ...titles)

### [Переменная "arguments"](https://learn.javascript.ru/rest-parameters-spread-operator" \l "the-arguments-variable)

**Стрелочные функции не имеют "arguments"**

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

// Объект arguments можно перебирать

// for (let arg of arguments) alert(arg);

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined (второго аргумента нет)

showName("Илья");

* Хотя arguments похож на массив, и его тоже можно перебирать, это всё же не массив. Он не поддерживает методы массивов, поэтому мы не можем, например, вызвать arguments.map(...).

### [Оператор расширения](https://learn.javascript.ru/rest-parameters-spread-operator" \l "spread-operator)

Мы узнали, как получить массив из списка параметров.

Но иногда нужно сделать в точности противоположное.

Например, есть встроенная функция [Math.max](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max). Она возвращает наибольшее число из списка:

alert( Math.max(3, 5, 1) ); // 5

Допустим, у нас есть массив чисел [3, 5, 1]. Как вызвать для него Math.max?

let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN

alert( Math.max(...arr) ); // 5 (оператор "раскрывает" массив в список аргументов)

Оператор расширения можно использовать и для **слияния массивов**:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

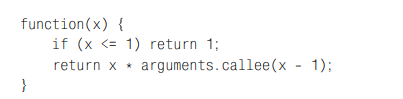
alert(merged); // 0,3,5,1,2,8,9,15

**превратить строку в массив символов:**

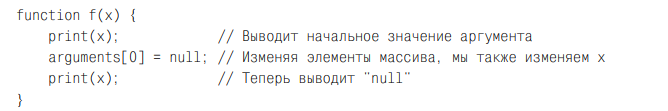
let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

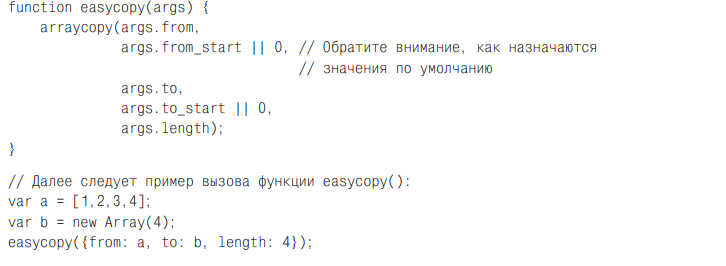
* Объект Arguments также определяет дополнительное свойство **callee**,

callee, ссылающееся на исполняемую в данный момент функцию. Пример рекурсии: 

* функции с переменным числом аргументов не должны допускать возможность вызова с пустым списком аргументов. Будет вполне разум ным использовать объект arguments[] при написании функции
* У объекта Arguments есть одна очень необычная особенность.



## Использование свойств объекта в качестве аргументов



## Prototype

Любая функция имеет свойство prototype, ссылающееся на предопределенный объект'прототип. Этот объект, который вступает в игру, когда функция ис пользуется в качестве конструктора с оператором new, играет важную роль в про цессе определения новых типов объектов

## Function expression vs declaration

Отличие в объявление от vs function declaration



Более тонкое отличие состоит, в том, когда создаётся функция движком JavaScript.

**Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.**

После того, как поток выполнения достигнет правой части выражения присваивания let sum = function… – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д. ).

С Function Declaration всё иначе.

**Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке).**

Другими словами, когда движок JavaScript готовится выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Можно считать этот процесс «стадией инициализации».

И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение.

В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений.

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости.

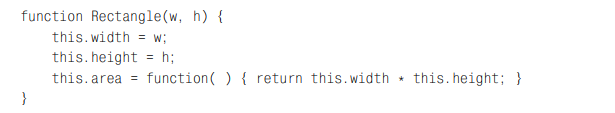
**В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него.**



## This in functions

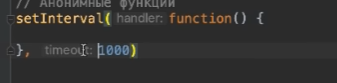
Когда функция вызывается таким способом, объект, посредством ко торого производится вызов, становится значением ключевого слова this.

## Присваивания свойств через функцию

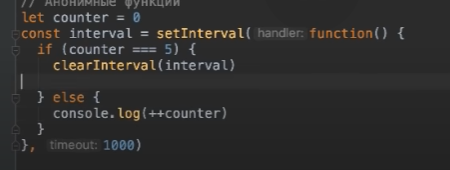


## Анонимные функции

Допустим, есть встроеная функция setInterval, которая будет выполнять нашу функцию Без названия каждые 1000мс. Вот она и анонимная



Пс чтобы остановить интервал можно юзать clearInterval. Типо мы сказали что у нас есть объект-интервал и в моменте просто останавливает его с помощью встроеной функции.



## Стрелочные функции

### [У стрелочных функций нет «this»](https://learn.javascript.ru/object-methods#u-strelochnyh-funktsiy-net-this), берется из внешней

Стрелочные функции особенные: у них **нет своего «собственного**» this. Если мы используем this внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Пример:

let group = {

title: "Our Group",

students: ["John", "Pete", "Alice"],

showList() {

this.students.forEach(

student => alert(this.title + ': ' + student)

);

}

};

group.showList();

**Если бы мы использовали «обычную» функцию, была бы ошибка:**

let group = {

title: "Our Group",

students: ["John", "Pete", "Alice"],

showList() {

this.students.forEach(function(student) {

// Error: Cannot read property 'title' of undefined

alert(this.title + ': ' + student)

});

}

};

group.showList();

Ошибка возникает потому, что forEach по умолчанию выполняет функции с this, равным undefined, и в итоге мы пытаемся обратиться к undefined.title.

Это не влияет на стрелочные функции, потому что у них просто нет this.

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение this, а хотим брать его из внешнего контекста.

**!!! Существует тонкая разница между стрелочной функцией => и обычной функцией, вызванной с .bind(this):**

* .bind(this) создаёт «связанную версию» функции.
* Стрелка => ничего не привязывает. У функции просто нет this. При получении значения this – оно, как обычная переменная, берётся из внешнего лексического окружения.

Также свойства:

* Не имеют arguments.
* Не могут быть вызваны с new.
* (У них также нет super

## bind - Привязка контекста к функции

пример ошибки

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(user.sayHi, 1000); // Привет, undefined!

Описание БИНД

let boundFunc = func.bind(context);

Например, здесь funcUser передаёт вызов в func, фиксируя this=user:

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

**Удобный метод: bindAll**

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

for (let key in user) {

if (typeof user[key] == 'function') {

user[key] = user[key].bind(user);

}

}

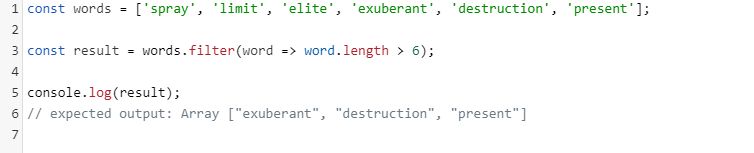
**Мы можем привязать не только this, но и аргументы**. Это делается редко, но иногда может быть полезно.

Полный синтаксис bind:

let bound = func.bind(context, [arg1], [arg2], ...);

## filter()

The **filter()** method **creates a new array** with all elements that pass the test implemented by the provided function.



## Символ \_ в параметрах функции

A single underscore is a convention used by some javascript programmers to indicate to other programmers that they should "ignore this binding/parameter". Since JavaScript doesn't do parameter-count checking the parameter could have been omitted entirely.

This symbol is often used (by convention again) in conjunction with fat-arrow functions to make them even terser and readable, like this:

const fun = \_ => console.log('Hello, World!')

fun()

In this case, the function needs no params to run, so the developer has used the underscore as a convention to indicate this. The same thing could be written like this:

const fun = () => console.log('Hello, World!')

fun()

The difference is that the second version is a function with no parameters, but the first version has a parameter called \_ that is ignored. These are different though and the second version is safer, if slightly more verbose (1 extra character).

Also, consider a case like

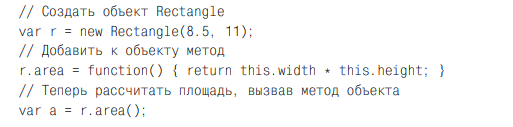
arr.forEach(function (\_, i) {..})

Where \_ indicates the first parameter is not to be used.

The use of underscores like this can get very confusing when using the popular lodash or underscore libraries.

# Классы

## Методы по ходу выполнения программы



# [«this»](https://learn.javascript.ru/object-methods" \l "this-ne-yavlyaetsya-fiksirovannym)

* The this keyword in the constructor does not have a value.

## Bind – В разделе с функциями

## [У стрелочных функций нет «this»](https://learn.javascript.ru/object-methods#u-strelochnyh-funktsiy-net-this)

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы используем this внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

let user = {

firstName: "Илья",

sayHi() {

let arrow = () => alert(this.firstName);

arrow();

}

};

user.sayHi(); // Илья

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение this, а хотим брать его из внешнего контекста. Позднее в главе [Повторяем стрелочные функции](https://learn.javascript.ru/arrow-functions) мы увидим больше примеров на эту тему.

## different values depending on where it is used:

* In a method, this refers to the **owner object**.
* Alone, this refers to the **global object**.
* In a function, this refers to the **global object**.
* In a function, in strict mode, this is undefined.
* In an event, this refers to the **element** that received the event.
* Methods like call(), and apply() can refer this to **any object**.

### this in a Method

In an object method, this refers to the "**owner**" of the method.

In the example on the top of this page, this refers to the **person** object.

The **person** object is the **owner** of the **fullName** method.

fullName : function() {  
  return **this**.firstName + " " + **this**.lastName;  
}

### this in a function(default)

In a JavaScript function, the owner of the function is the **default** binding for this.

So, in a function, this refers to the Global object [object Window].

function myFunction() {  
  return this;  
}

### this in a function(strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, this is undefined.

"use strict";  
function myFunction() {  
  return this;  
}

### this Alone

When used alone, the **owner** is the Global object, so this refers to the Global object.

In a browser window the Global object is [object Window]:

var x = this;

 In **strict mode**, when used alone, this also refers to the Global object [object Window]:

"use strict";  
var x = this;

### this in Event Handlers

In HTML event handlers, this refers to the HTML element that received the event:

<button onclick="this.style.display='none'">  
  Click to Remove Me!  
</button>

Кновочка исчезнет если на нее нажать

### Object Method Binding

In these examples, this is the person object (The person object is the "owner" of the function):

var **person** = {  
  firstName  : "John",  
  lastName   : "Doe",  
  id         : 5566,  
  myFunction : function() {  
    return **this**;  
  }  
};

var **person** = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return **this**.firstName + " " + **this**.lastName;  
  }  
};

Другими словами, this.firstName означает свойство firstName этого объекта.

## Explicit Function Binding

The call() and apply() methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument

## Вообще в js THIS не является фиксированным

Если вы до этого изучали другие языки программирования, тогда вы, скорее всего, привыкли к идее "фиксированного this" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод).

В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Эта идея вычисления this в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Здесь мы не будем судить о том, является ли это решение в языке хорошим или плохим. Мы должны понимать, как с этим работать, чтобы получать выгоды и избегать проблем.

Пример

В этом коде нет синтаксической ошибки:

function sayHi() {

alert( this.name );

}

Значение this вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

let user = { name: "Джон" };

let admin = { name: "Админ" };

function sayHi() {

alert( this.name );

}

// используем одну и ту же функцию в двух объектах

user.f = sayHi;

admin.f = sayHi;

// вызовы функции, приведённые ниже, имеют разное значение this

// "this" внутри функции является ссылкой на объект, который указан "перед точкой"

user.f(); // Джон (this == user)

admin.f(); // Админ (this == admin)

admin['f'](); // Админ (неважен способ доступа к методу - через точку или квадратные

## Вызов без объекта: this == undefined

Мы даже можем вызвать функцию вовсе без использования объекта:

function sayHi() {

alert(this);

}

sayHi(); // undefined

В строгом режиме ("use strict") в таком коде значением this будет являться undefined. Если мы попытаемся получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет глобальный объект (window для браузера, мы вернёмся к этому позже в главе [Глобальный объект](https://learn.javascript.ru/global-object)). Это – исторически сложившееся поведение this, которое исправляется использованием строгого режима ("use strict").

Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта

потеря значения this:

let user = {

name: "Джон",

hi() { alert(this.name); },

bye() { alert("Пока"); }

};

user.hi(); // Джон (простой вызов метода работает хорошо)

// теперь давайте попробуем вызывать user.hi или user.bye

// в зависимости от имени пользователя user.name

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

(user.name == "Джон" ? user.hi() : user.bye()); *//нет ошибки*

В 9 строчке кода используется условный оператор ?, который определяет, какой будет вызван метод (user.hi или user.bye) в зависимости от выполнения условия. В данном случае будет выбран user.hi.

Затем метод тут же вызывается с помощью скобок (). Но вызов не работает как положено!

Вы можете видеть, что при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим).

**Еще один пример:**

let user = {

name: "Джон",

hi() { alert(this.name); }

};

// разделим получение метода объекта и его вызов в разных строках

let hi = user.hi;

hi(); // Ошибка, потому что значением this является undefined

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

**Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого**[**Reference Type**](https://tc39.github.io/ecma262/#sec-reference-specification-type)**.**

## Ссылочный тип

Этот ссылочный тип (Reference Type) является внутренним типом. Мы не можем явно использовать его, но он используется внутри языка.

Значение ссылочного типа – это «триплет»: комбинация из трёх значений (base, name, strict), где:

* base – это объект.
* name – это имя свойства объекта.
* strict – это режим исполнения. Является true, если действует строгий режим (use strict).

// значение ссылочного типа (Reference Type)

(user, "hi", true)

Когда скобки () применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки . до вызывающих скобок ().

При любой другой операции(предыдущий пример), например, присваивании hi = user.hi, ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без this.

Таким образом, значение **this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки** obj.method() или квадратных скобок obj['method']() (они делают то же самое). Позднее в этом учебнике мы изучим различные варианты решения проблемы потери значения this. Например, такие как [func.bind()](https://learn.javascript.ru/bind#solution-2-bind).

# Callbacks

функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Если после вызова loadScript(…) есть какой-то код, то он не будет ждать, пока скрипт загрузится.

loadScript('/my/script.js');

// код, написанный после вызова функции loadScript,

// не будет дожидаться полной загрузки скрипта

// ...

Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать её, когда скрипт загрузится:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

loadScript('/my/script.js', function() {

// эта функция вызовется после того, когда загрузится скрипт

newFunction(); // теперь всё работает

...

});

Возьмём для **примера** реальный скрипт с библиотекой функций:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Здорово, скрипт ${script.src} загрузился`);

alert( \_ ); // функция, объявленная в загруженном скрипте

});

# Promise

Универсальный метод для навешивания обработчиков:

promise.then(onFulfilled, onRejected)

* onFulfilled – функция, которая будет вызвана с результатом при resolve.
* onRejected – функция, которая будет вызвана с ошибкой при reject.

С его помощью можно назначить как оба обработчика сразу, так и только один:

// onFulfilled сработает при успешном выполнении

promise.then(onFulfilled)

// onRejected сработает при ошибке

promise.then(null, onRejected)

**.catch**

Для того, чтобы поставить обработчик только на ошибку, вместо .then(null, onRejected) можно написать .catch(onRejected) – это то же самое.

**Синхронный throw – то же самое, что reject**

Если в функции промиса происходит синхронный throw (или иная ошибка), то вызывается reject:

'use strict';

let p = new Promise((resolve, reject) => {

// то же что reject(new Error("o\_O"))

throw new Error("o\_O");

})

p.catch(alert); // Error: o\_O

Посмотрим, как это выглядит вместе, на простом примере.

На примере клиент-сервер: 

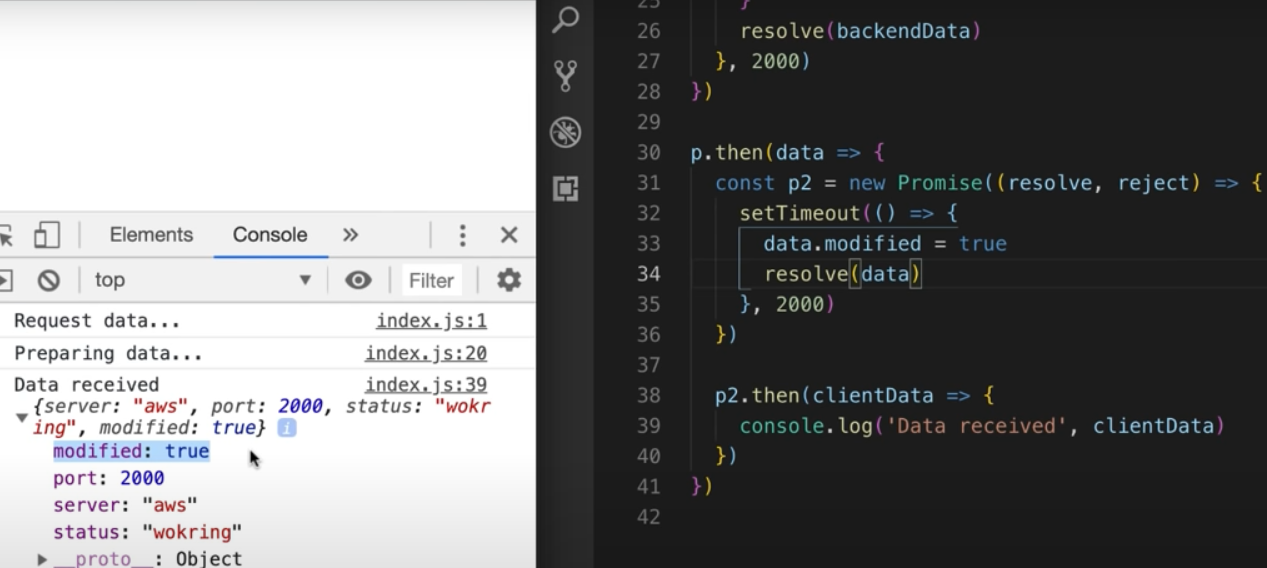
Допустим мы хотим сделать ассинхронную операцию: Подготовка данных на серве(какое то время) и потом вывести что все готово.

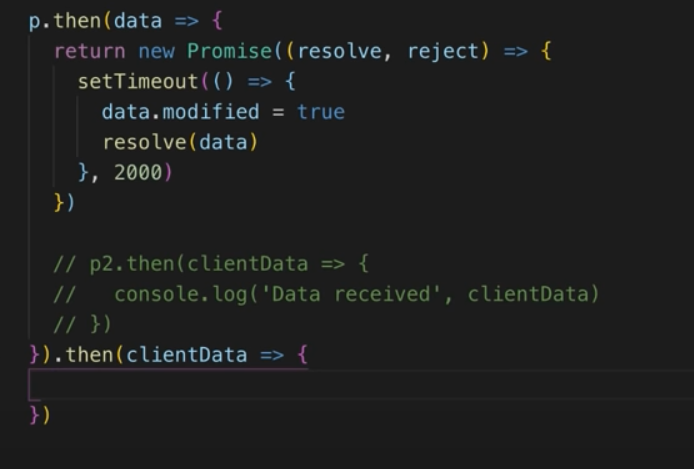
Пишем промис:

18: передаем функцию как параметр, можно юзать и стрелочную функцию

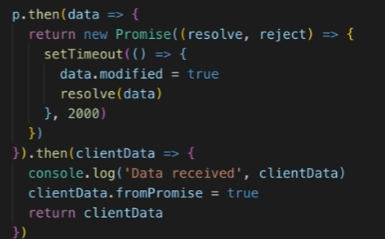
Когда все готово пишем resolve как выход из функции, в нее передаем то что хотим передать наружу из промиса(у нас бэкэнд дата)

Потом, то что нужно сделать за промисом, пишем через then, и передаем еще одну функцию. В нем можно писать еще промисы

чтобы не растягивать и не создавать кучу промисов, мы можем вернуть промис и сразу написать .then:

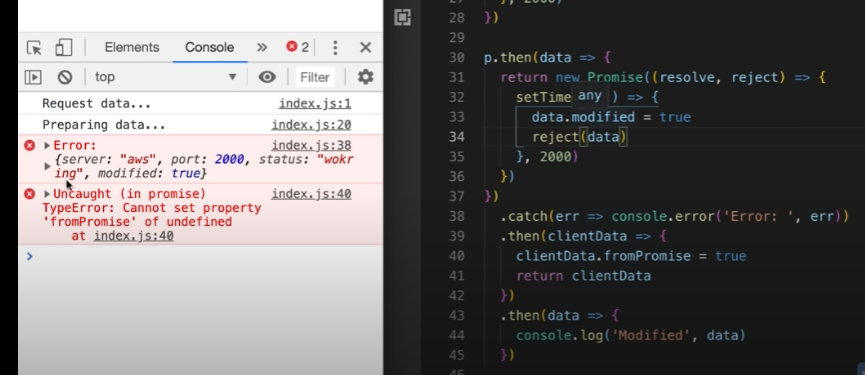


Потом, мы можем изменять данные внутри, например мы чтото сделали с ClientData и зотим его вернуть, мы можем протсто вернуть из промиса объект



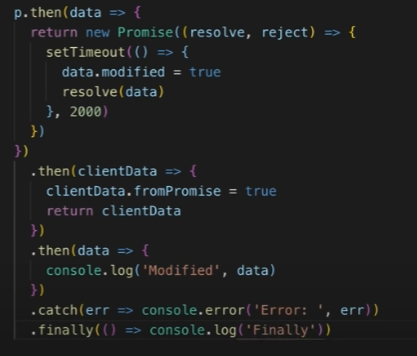
То есть не обязательно возращать промисы

**Про Reject + Catch:**

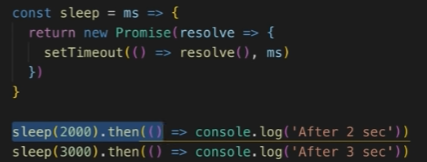


Здесь мы сказали что чтото не так в reject и потом поймали B catch

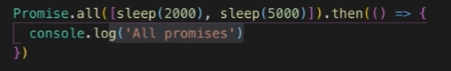
**Finally** вывзывается всегда, есть ошибка или нет



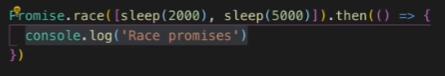
Ну еще один пример как писать красивее



Есть y promise еще метод **All,** который выполняет все промисы переданные в параметры в массив параллельно

****

Также есть Promise.race, который срабатывает когда выполнится первый



# Async/await

Начнём с ключевого слова async. Оно ставится перед функцией, вот так:

async function f() {

return 1;

}

У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

Например, эта функция возвратит выполненный промис с результатом 1:

async function f() {

return 1;

}

f().then(alert); // 1

Можно и явно вернуть промис, результат будет одинаковым:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

### await

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

В этом примере промис успешно выполнится через 1 секунду

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("готово!"), 1000)

});

let result = await promise; // будет ждать, пока промис не выполнится (\*)

alert(result); // "готово!"

}

f();

**await нельзя использовать в обычных функциях**

**await нельзя использовать на верхнем уровне вложенности**

Программисты, узнав об await, часто пытаются использовать эту возможность на верхнем уровне вложенности (вне тела функции). Но из-за того, что await работает только внутри async–функций, так сделать не получится:

// SyntaxError на верхнем уровне вложенности

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

Можно обернуть этот код в анонимную async–функцию, тогда всё заработает:

(async () => {

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

...

})();

# fetch

let promise = fetch(url, [options])

* **url** – URL для отправки запроса.
* **options** – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

Браузер сразу же начинает запрос и возвращает **промис**, который внешний код использует для получения результата.

**Во-первых, promise выполняется с объектом встроенного класса**[**Response**](https://fetch.spec.whatwg.org/#response-class)**в качестве результата, как только сервер пришлёт заголовки ответа.**

let response = await fetch(url);

if (response.ok) { // если HTTP-статус в диапазоне 200-299

// получаем тело ответа (см. про этот метод ниже)

let json = await response.json();

} else {

alert("Ошибка HTTP: " + response.status);

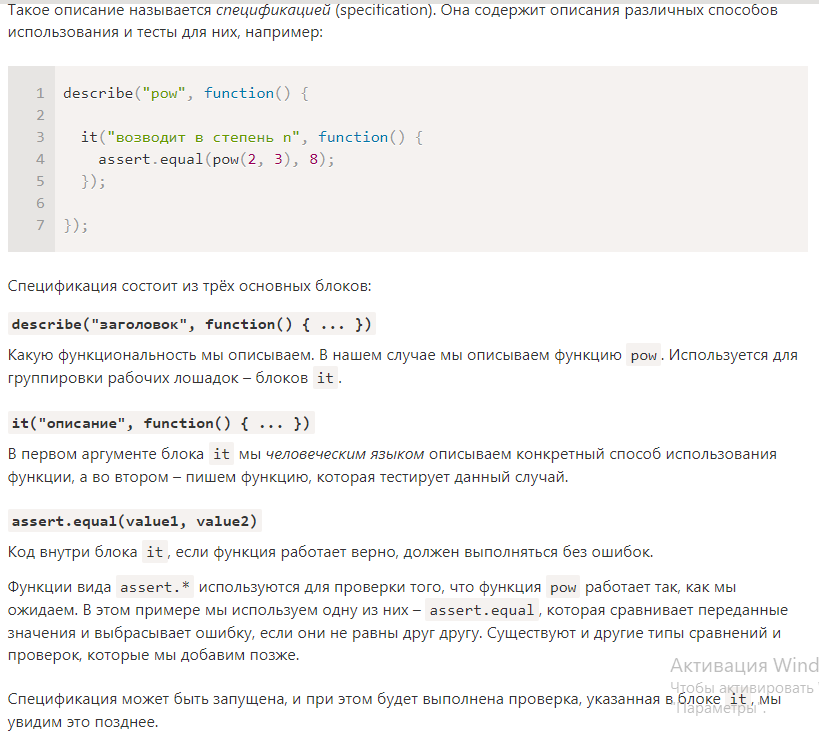
}

**Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.**

Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

* **response.text()** – читает ответ и возвращает как обычный текст,
* **response.json()** – декодирует ответ в формате JSON,
* **response.formData()** – возвращает ответ как объект FormData (разберём его в [следующей главе](https://learn.javascript.ru/formdata)),

# Testing



# [Babel](https://learn.javascript.ru/polyfills" \l "babel)

Когда мы используем современные возможности JavaScript, некоторые движки могут не поддерживать их. Как было сказано выше, не везде реализованы все функции.

И тут приходит на помощь Babel.

[Babel](https://babeljs.io/) – это [транспилер](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%B0%D0%B9%D0%BB%D0%B5%D1%80). Он переписывает современный JavaScript-код в предыдущий стандарт.

На самом деле, есть две части Babel:

1. Во-первых, транспилер, который переписывает код. Разработчик запускает Babel на своём компьютере. Он переписывает код в старый стандарт. И после этого код отправляется на сайт. Современные сборщики проектов, такие как [webpack](http://webpack.github.io/) или [brunch](http://brunch.io/), предоставляют возможность запускать транспилер автоматически после каждого изменения кода, что позволяет экономить время.
2. Во-вторых, полифил.

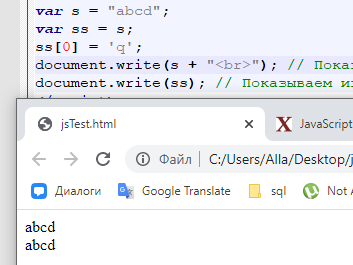
Новые возможности языка могут включать встроенные функции и синтаксические конструкции. Транспилер переписывает код, преобразовывая новые синтаксические конструкции в старые. Но что касается новых встроенных функций, нам нужно их как-то реализовать. JavaScript является высокодинамичным языком, скрипты могут добавлять/изменять любые функции, чтобы они вели себя в соответствии с современным стандартом.

Термин «полифил» означает, что скрипт «заполняет» пробелы и добавляет современные функции.

# ссылки

• Если начальное значение не задано в инструкции var, то переменная объявляет ся, но ее начальное значение остается неопределенным (undefined), пока не будет изменено программой.

* ! если не объявить(var i) переменную то она сама объявится интерпритатором js
* Переменные, объявленные с помощью инструкции var, называются долговре' менными (permanent): попытка удалить их с помощью оператора delete приведет к ошибке.
* С помощью инструкции var можно объявить одну и ту же переменную несколько раз
* Чтобы объявить переменную как глобальную, надо написать без var, в функциях она будет изменяться
* Строки неизменны



* При определении идентичности двух значений **оператор ===** руководствуется следующими правилами:
  1. Если два значения имеют различные типы, они не идентичны.
  2. Два значения идентичны, только если оба они представляют собой числа, имеют одинаковые значения и не являются значением NaN (в этом, последнем случае они не идентичны). Значение NaN никогда не бывает идентичным ни какому значению, даже самому себе! Чтобы проверить, является ли значение значением NaN, следует использовать глобальную функцию isNaN().
  3. Если оба значения представляют собой строки и содержат одни и те же симво лы в тех же позициях, они идентичны. Если строки отличаются по длине или содержимому, они не идентичны. Обратите внимание, что в некоторых случа ях стандарт Unicode допускает несколько способов кодирования одной и той же строки. Однако для повышения эффективности сравнение строк в Java Script выполняется строго посимвольно, при этом предполагается, что все строки перед сравнением преобразованы в «нормализованную форму». Другой способ сравнения строк обсуждается в части III книги при описании мето да String.localeCompare().
  4. Если оба значения представляют собой логические значения true или false, то они идентичны.
  5. Если оба значения ссылаются на один и тот же объект, массив или функцию, то они идентичны. Если они ссылаются на различные объекты (массивы или функции), они не идентичны, даже если оба имеют идентичные свойства или идентичные элементы.
  6. Если оба значения равны null или undefined, то они идентичны.
* Оператор in

var point = { x:1, y:1 }; // Определяем объект

var has\_x\_coord = "x" in point; // Равно true

var has\_y\_coord = "y" in point; // Равно true

var has\_z\_coord = "z" in point; // Равно false; это не трехмерная точка

var ts = "toString" in point; // Унаследованное свойство; равно true

* Instanceof()

